PostScript®
LANGUAGE REFERENCE
third edition

Adobe Systems Incorporated

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Harlow, England • Amsterdam • Bonn • Sydney • Singapore • Tokyo
Madrid • San Juan • Paris • Seoul • Milan • Mexico City • Taipei

GOOGLE EXHIBIT 1053

Library of Congress Cataloging-in-Publication Data
PostScript language reference manual / Adobe Systems Incorporated. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-37922-8
1. PostScript (Computer program language) I. Adobe Systems.
QA76.73.P67 P67 1999
005.13'3—dc21 98-55489
CIP

© 1985–1999 Adobe Systems Incorporated. All rights reserved.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated.

No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher.

PostScript is a registered trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Except as otherwise stated, any mention of a "PostScript printer," "PostScript software," or similar item refers to a product that contains PostScript technology created or licensed by Adobe Systems Incorporated, not to one that purports to be merely compatible.

Adobe, Adobe Illustrator, Adobe Type Manager, Chameleon, Display PostScript, FrameMaker, Minion, Myriad, Photoshop, PostScript, PostScript 3, and the PostScript logo are trademarks of Adobe Systems Incorporated. LocalTalk, QuickDraw, and TrueType are trademarks and Mac OS is a registered trademark of Apple Computer, Inc. Helvetica and Times are registered trademarks of Linotype-Hell AG and/or its subsidiaries. Times New Roman is a trademark of The Monotype Corporation registered in the U.S. Patent and Trademark Office and may be registered in certain other jurisdictions. Unicode is a registered trademark of Unicode, Inc. PANTONE is a registered trademark and Hexachrome is a trademark of Pantone, Inc. Windows is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners.

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

ISBN 0-201-37922-8
1 2 3 4 5 6 7 8 9 CRS 03 02 01 00 99
First printing February 1999

Contents

Preface xiii

Chapter 1: Introduction 1

1.1 About This Book 3
1.2 Evolution of the PostScript Language 5
1.3 LanguageLevel 3 Overview 6
1.4 Related Publications 7
1.5 Copyrights and Trademarks 9

Chapter 2: Basic Ideas 11

2.1 Raster Output Devices 11
2.2 Scan Conversion 12
2.3 Page Description Languages 13
2.4 Using the PostScript Language 15

Chapter 3: Language 23

3.1 Interpreter 24
3.2 Syntax 25
3.3 Data Types and Objects 34
3.4 Stacks 45
3.5 Execution 46
3.6 Overview of Basic Operators 51
3.7 Memory Management 56
3.8 File Input and Output 73
3.9 Named Resources 87
3.10 Functions 106
3.11 Errors 114
3.12 Early Name Binding 117
3.13 Filtered Files Details 123
3.14 Binary Encoding Details 156

Chapter 4: Graphics 175

4.1 Imaging Model 176
4.2 Graphics State 178
4.3 Coordinate Systems and Transformations 182

4.4 Path Construction 189
4.5 Painting 193
4.6 User Paths 197
4.7 Forms 206
4.8 Color Spaces 210
4.9 Patterns 248
4.10 Images 288

Chapter 5: Fonts 313

5.1 Organization and Use of Fonts 313
5.2 Font Dictionaries 321
5.3 Character Encoding 328
5.4 Glyph Metric Information 331
5.5 Font Cache 333
5.6 Unique ID Generation 335
5.7 Type 3 Fonts 337
5.8 Additional Base Font Types 343
5.9 Font Derivation and Modification 348
5.10 Composite Fonts 357
5.11 CID-Keyed Fonts 364

Chapter 6: Device Control 391

6.1 Using Page Devices 393
6.2 Page Device Parameters 398
6.3 In-RIP Trapping 439
6.4 Output Device Dictionary 455

Chapter 7: Rendering 457

7.1 CIE-Based Color to Device Color 459
7.2 Conversions among Device Color Spaces 473
7.3 Transfer Functions 478
7.4 Halftones 480
7.5 Scan Conversion Details 501

Chapter 8: Operators 505

8.1 Operator Summary 508
8.2 Operator Details 524

Appendix A: LanguageLevel Feature Summary 725

A.1 LanguageLevel 3 Features 725
A.2 LanguageLevel 2 Features 731
A.3 Incompatibilities 735

Appendix B: Implementation Limits 737

B.1 Typical Limits 738
B.2 Virtual Memory Use 742

Appendix C: Interpreter Parameters 745

C.1 Properties of User and System Parameters 746
C.2 Defined User and System Parameters 749
C.3 Details of User and System Parameters 753
C.4 Device Parameters 760

Appendix D: Compatibility Strategies 761

D.1 The LanguageLevel Approach 761
D.2 When to Provide Compatibility 763
D.3 Compatibility Techniques 765
D.4 Installing Emulations 769

Appendix E: Character Sets and Encoding Vectors 773

E.1 Times Family 775
E.2 Helvetica Family 776
E.3 Courier Family 777
E.4 Symbol 778
E.5 Standard Latin Character Set 779
E.6 StandardEncoding Encoding Vector 784
E.7 ISOLatin1Encoding Encoding Vector 785
E.8 CE Encoding Vector 786
E.9 Expert Character Set 787
E.10 Expert Encoding Vector 790
E.11 ExpertSubset Encoding Vector 791
E.12 Symbol Character Set 792
E.13 Symbol Encoding Vector 794

Appendix F: System Name Encodings 795

Appendix G: Operator Usage Guidelines 801

Bibliography 811

INDEX 817

Figures

2.1 How the PostScript interpreter and an application interact 16

3.1 Mapping with the **Decode** array 112
3.2 Homogeneous number array 161
3.3 Binary object sequence 164

4.1 The two squares produced by Example 4.1 186
4.2 Effects of coordinate transformations 188
4.3 Nonzero winding number rule 195
4.4 Even-odd rule 196
4.5 Color specification 212
4.6 Color rendering 213
4.7 Component transformations in the **CIEBasedABC** color space 222
4.8 Component transformations in the **CIEBasedA** color space 229
4.9 **CIEBasedDEFG** pre-extension to the **CIEBasedABC** color space 232
4.10 Output from Example 4.21 256
4.11 Output from Example 4.23 259
4.12 Starting a new triangle in a free-form Gouraud-shaded triangle mesh 272
4.13 Connecting triangles in a free-form Gouraud-shaded triangle mesh 272
4.14 Varying the value of the edge flag to create different shapes 273
4.15 Lattice-form triangular meshes 275
4.16 Coordinate mapping from a unit square to a four-sided Coons patch 277
4.17 Painted area and boundary of a Coons patch 279
4.18 Color values and edge flags in Coons patch meshes 281
4.19 Edge connections in a Coons patch mesh 282
4.20 Control points in a tensor-product mesh 284
4.21 Typical sampled image 288
4.22 Image data organization and processing 293
4.23 Source image coordinate system 294
4.24 Mapping the source image 295

5.1 Results of Example 5.2 317
5.2 Glyphs painted in 50% gray 318
5.3 Glyph outlines treated as a path 319
5.4 Graphics clipped by a glyph path 320
5.5 Encoding scheme for Type 1 fonts 329
5.6 Glyph metrics 331
5.7 Relationship between two sets of metrics 333
5.8 Output from Example 5.6 341
5.9 Composite font mapping example 359
5.10 CID-keyed font basics 367
5.11 Type 0 CIDFont character processing 372

6.1 Trapping example 440
6.2 Sliding trap 452

7.1 Various halftoning effects 486
7.2 Halftone cell with a nonzero angle 493
7.3 Angled halftone cell divided into two squares 493
7.4 Halftone cell and two squares tiled across device space 494
7.5 Tiling of device space in a type 16 halftone dictionary 497
7.6 Rasterization without stroke adjustment 504

8.1 **arc** operator 530
8.2 **arc** operator example 531
8.3 **arcn** operator example 532
8.4 **arct** operator 533
8.5 **arct** operator example 533
8.6 **curveto** operator 565
8.7 **imagemask** example 609
8.8 **setflat** operator 669
8.9 Line cap parameter shapes 673
8.10 Line join parameter shapes 674
8.11 Miter length 676

Tables

2.1 Control characters for the interactive executive 21

3.1 White-space characters 27
3.2 Types of objects 34
3.3 Standard local dictionaries 65
3.4 Standard global dictionaries 66
3.5 Access strings 79
3.6 Standard filters 85
3.7 Regular resources 91
3.8 Resources whose instances are implicit 91
3.9 Resources used in defining new resource categories 92
3.10 Standard procedure sets in LanguageLevel 3 96
3.11 Entries in a category implementation dictionary 101
3.12 Entries common to all function dictionaries 108
3.13 Additional entries specific to a type 0 function dictionary 109
3.14 Additional entries specific to a type 2 function dictionary 113
3.15 Additional entries specific to a type 3 function dictionary 114
3.16 Entries in the **$error** dictionary 116
3.17 Entries in an **LZWEncode** or **LZWDecode** parameter dictionary 133
3.18 Typical LZW encoding sequence 135
3.19 Entries in a **FlateEncode** or **FlateDecode** parameter dictionary 138
3.20 Predictor-related entries in an LZW or Flate filter parameter dictionary 141
3.21 Entries in a **CCITTFaxEncode** or **CCITTFaxDecode** parameter dictionary 144
3.22 Entries in a **DCTEncode** parameter dictionary 148
3.23 Entries in a **SubFileDecode** parameter dictionary *(LanguageLevel 3) 152*
3.24 Entries in a **ReusableStreamDecode** parameter dictionary 155
3.25 Binary token interpretation 158
3.26 Number representation in header for a homogeneous number array 162
3.27 Object type, length, and value fields 166

4.1 Device-independent parameters of the graphics state 179
4.2 Device-dependent parameters of the graphics state 180
4.3 Operation codes for encoded user paths 201
4.4 Entries in a type 1 form dictionary 208
4.5 Entries in a **CIEBasedABC** color space dictionary 223

4.6 Entries in a **CIEBasedA** color space dictionary 229
4.7 Additional entries specific to a **CIEBasedDEF** color space dictionary 233
4.8 Additional entries specific to a **CIEBasedDEFG** color space dictionary 235
4.9 Entries in a type 1 pattern dictionary 251
4.10 Entries in a type 2 pattern dictionary 260
4.11 Entries common to all shading dictionaries 262
4.12 Additional entries specific to a type 1 shading dictionary 265
4.13 Additional entries specific to a type 2 shading dictionary 266
4.14 Additional entries specific to a type 3 shading dictionary 268
4.15 Additional entries specific to a type 4 shading dictionary 270
4.16 Additional entries specific to a type 5 shading dictionary 275
4.17 Additional entries specific to a type 6 shading dictionary 279
4.18 Data values in a Coons patch mesh 282
4.19 Data values in a tensor-product patch mesh 287
4.20 Entries in a type 1 image dictionary 298
4.21 Typical **Decode** arrays 300
4.22 Entries in a type 3 image dictionary 304
4.23 Entries in an image data dictionary 305
4.24 Entries in a mask dictionary 306
4.25 Entries in a type 4 image dictionary 307

5.1 Font types 322
5.2 Entries common to all font dictionaries 324
5.3 Additional entries common to all base fonts 325
5.4 Additional entries specific to Type 1 fonts 326
5.5 Entries in a **FontInfo** dictionary 327
5.6 Additional entries specific to Type 3 fonts 338
5.7 Additional entries specific to Type 42 fonts 346
5.8 Additional entries specific to Type 0 fonts 357
5.9 **FMapType** mapping algorithms 360
5.10 Entries in a **CIDSystemInfo** dictionary 368
5.11 **CIDFontType** and **FontType** values 370
5.12 Entries common to all **CIDFont** dictionaries 370
5.13 Additional entries specific to Type 0 **CIDFont** dictionaries 373
5.14 Entries in a dictionary in **FDArray** 374
5.15 Entries replacing **Subrs** in the **Private** dictionary of an **FDArray** dictionary 375
5.16 Additional entry specific to Type 1 **CIDFont** dictionaries 377
5.17 Additional entries specific to Type 2 **CIDFont** dictionaries 378
5.18 Entries in a **CMap** dictionary 383

6.1 Categories of page device parameters 399
6.2 Page device parameters related to media selection 400

6.3 Page device parameters related to roll-fed media 412
6.4 Page device parameters related to page image placement 414
6.5 Page device parameters related to page delivery 417
6.6 Page device parameters related to color support 420
6.7 Page device parameters related to device initialization and page setup 426
6.8 Page device parameter related to recovery policies 433
6.9 Entries in the **Policies** dictionary 433
6.10 Entries in a Type 1001 trapping details dictionary 442
6.11 Entries in a colorant details dictionary 443
6.12 Entries in a colorant subdictionary 444
6.13 Entries in a trapping parameter dictionary 447
6.14 Example of normal trapping rule 451
6.15 Entries in a **ColorantZoneDetails** dictionary 454
6.16 Entries in an output device dictionary 455

7.1 Entries in a type 1 CIE-based color rendering dictionary 463
7.2 Rendering intents 470
7.3 Types of halftone dictionaries 485
7.4 Entries in a type 1 halftone dictionary 487
7.5 Entries in a type 3 halftone dictionary 490
7.6 Entries in a type 6 halftone dictionary 491
7.7 Entries in a type 10 halftone dictionary 495
7.8 Entries in a type 16 halftone dictionary 496
7.9 Entries in a proprietary halftone dictionary 500

8.1 Operand and result types 506

A.1 LanguageLevel 3 operators defined in procedure sets 726
A.2 New resource categories 727
A.3 New resource instances 727
A.4 New page device and interpreter parameters 728

B.1 Architectural limits 739
B.2 Typical memory limits in LanguageLevel 1 741

C.1 User parameters 749
C.2 System parameters 751

E.1 Encoding vectors 773

G.1 Guidelines summary 802

Preface

IN THE 1980S, ADOBE DEVISED a powerful graphics imaging model that over time has formed the basis for the Adobe PostScript technologies. These technologies—a combination of the PostScript language and PostScript language–based graphics and text-formatting applications, drivers, and imaging systems—have forever changed the printing and publishing world by sparking the desktop and digital publishing revolutions. Since their inception, PostScript technologies have enabled unprecedented control of the look and feel of printed documents and have changed the overall process for designing and printing them as well. The capabilities PostScript makes possible have established it as the industry page description language standard.

Today, as never before, application developers and imaging systems vendors support the PostScript language as the industry standard. We at Adobe accept our responsibility as stewards of this standard to continually advance the standard in response to the creative needs of the industry.

With this third advance of the language, which we call LanguageLevel 3, Adobe has greatly expanded the boundaries of imaging capabilities made possible through the PostScript language. This most recent advance has yielded significant improvements in the efficiency and performance of the language as well as in the quality of final output.

To complement the strengths of LanguageLevel 3, Adobe PostScript 3 imaging system technologies have been engineered to exploit the new LanguageLevel 3 constructs to the fullest extent, fulfilling the Adobe commitment to provide printing solutions for the broad spectrum of users.

No significant change comes without the concerted effort of many individuals. The work to advance the PostScript language and to create Adobe PostScript 3 imaging system technologies is no exception. Our goal since the introduction of the first Adobe imaging model has been nothing less than to provide the most innovative, meaningful imaging solutions in the industry. Dedicated Adobe employees and many industry partners have striven to make that goal a reality. We take this opportunity to thank all those who contributed to this effort.

John Warnock and Chuck Geschke
February 1999

CHAPTER 1

Introduction

THE POSTSCRIPT® LANGUAGE is a simple interpretive programming language with powerful graphics capabilities. Its primary application is to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages, according to the Adobe imaging model. A program in this language can communicate a description of a document from a composition system to a printing system or control the appearance of text and graphics on a display. The description is high-level and device-independent.

The page description and interactive graphics capabilities of the PostScript language include the following features, which can be used in any combination:

- Arbitrary shapes made of straight lines, arcs, rectangles, and cubic curves. Such shapes may self-intersect and have disconnected sections and holes.
- Painting operators that permit a shape to be outlined with lines of any thickness, filled with any color, or used as a clipping path to crop any other graphic. Colors can be specified in a variety of ways: grayscale, RGB, CMYK, and CIE-based. Certain other features are also modeled as special kinds of colors: repeating patterns, smooth shading, color mapping, and spot colors.
- Text fully integrated with graphics. In the Adobe imaging model, text characters in both built-in and user-defined fonts are treated as graphical shapes that may be operated on by any of the normal graphics operators.
- Sampled images derived from natural sources (such as scanned photographs) or generated synthetically. The PostScript language can describe images sampled at any resolution and according to a variety of color models. It provides a number of ways to reproduce images on an output device.

- A general coordinate system that supports all combinations of linear transformations, including translation, scaling, rotation, reflection, and skewing. These transformations apply uniformly to all elements of a page, including text, graphical shapes, and sampled images.

A PostScript page description can be rendered on a printer, display, or other output device by presenting it to a PostScript interpreter controlling that device. As the interpreter executes commands to paint characters, graphical shapes, and sampled images, it converts the high-level PostScript description into the low-level raster data format for that particular device.

Normally, application programs such as document composition systems, illustrators, and computer-aided design systems generate PostScript page descriptions automatically. Programmers generally write PostScript programs only when creating new applications. However, in special situations a programmer can write PostScript programs to take advantage of capabilities of the PostScript language that are not accessible through an application program.

The extensive graphics capabilities of the PostScript language are embedded in the framework of a general-purpose programming language. The language includes a conventional set of data types, such as numbers, arrays, and strings; control primitives, such as conditionals, loops, and procedures; and some unusual features, such as dictionaries. These features enable application programmers to define higher-level operations that closely match the needs of the application and then to generate commands that invoke those higher-level operations. Such a description is more compact and easier to generate than one written entirely in terms of a fixed set of basic operations.

PostScript programs can be created, transmitted, and interpreted in the form of ASCII source text as defined in this book. The entire language can be described in terms of printable characters and white space. This representation is convenient for programmers to create, manipulate, and understand. It also facilitates storage and transmission of files among diverse computers and operating systems, enhancing machine independence.

There are also binary encoded forms of the language for use in suitably controlled environments—for example, when the program is assured of a fully transparent communications path to the PostScript interpreter. Adobe recommends strict adherence to the ASCII representation of PostScript programs for document interchange or archival storage.

1.1 About This Book

This is the programmer's reference for the PostScript language. It is the definitive documentation for the syntax and semantics of the language, the imaging model, and the effects of the graphics operators.

- Chapter 2, "Basic Ideas," is an informal presentation of some basic ideas underlying the more formal descriptions and definitions to come in later chapters. These include the properties and capabilities of raster output devices, requirements for a language that effectively uses those capabilities, and some pragmatic information about the environments in which the PostScript interpreter operates and the kinds of PostScript programs it typically executes.
- Chapter 3, "Language," introduces the fundamentals of the PostScript language: its syntax, semantics, data types, execution model, and interactions with application programs. This chapter concentrates on the conventional programming aspects of the language, ignoring its graphical capabilities and use as a page description language.
- Chapter 4, "Graphics," introduces the Adobe imaging model at a device-independent level. It describes how to define and manipulate graphical entities—lines, curves, filled areas, sampled images, and higher-level structures such as patterns and forms. It includes complete information on the color models that the PostScript language supports.
- Chapter 5, "Fonts," describes how the PostScript language deals with text. Characters are defined as graphical shapes, whose behavior conforms to the imaging model presented in Chapter 4. Because of the importance of text in most applications, the PostScript language provides special capabilities for organizing sets of characters as fonts and for painting characters efficiently.
- Chapter 6, "Device Control," describes how a page description communicates its document processing requirements to the output device. These include page size, media selection, finishing options, and in-RIP trapping.
- Chapter 7, "Rendering," details the device-dependent aspects of rendering page descriptions on raster output devices (printers and displays). These include color rendering, transfer functions, halftoning, and scan conversion, each of which is device-dependent in some way.

- Chapter 8, "Operators," describes all PostScript operators and procedures. The chapter begins by categorizing operators into functional groups. Then the operators appear in alphabetical order, with complete descriptions of their operands, results, side effects, and possible errors.

The appendices contain useful tables and other auxiliary information.

- Appendix A, "LanguageLevel Feature Summary," summarizes the ways the PostScript language has been extended with new operators and other features over time.
- Appendix B, "Implementation Limits," describes typical limits imposed by implementations of the PostScript interpreter—for example, maximum integer value and maximum stack depth.
- Appendix C, "Interpreter Parameters," specifies various parameters to control the operation and behavior of the PostScript interpreter. Most of these parameters have to do with allocation of memory and other resources for specific purposes.
- Appendix D, "Compatibility Strategies," helps PostScript programmers take advantage of newer PostScript language features while maintaining compatibility with the installed base of older PostScript interpreter products.
- Appendix E, "Character Sets and Encoding Vectors," describes the organization of common fonts that are built into interpreters or are available as separate software products.
- Appendix F, "System Name Encodings," assigns numeric codes to standard names, for use in binary-encoded PostScript programs.
- Appendix G, "Operator Usage Guidelines," provides guidelines for PostScript operators whose use can cause unintended side effects, make a document device-dependent, or inhibit postprocessing of a document by other programs.

The book concludes with a Bibliography and an Index.

The enclosed CD-ROM contains the entire text of this book in Portable Document Format (PDF).

1.2 Evolution of the PostScript Language

Since its introduction in 1985, the PostScript language has been considerably extended for greater programming power, efficiency, and flexibility. Typically, these language extensions have been designed to adapt the PostScript language to new imaging technologies or system environments. While these extensions have introduced significant new functionality and flexibility to the language, the basic imaging model remains unchanged.

Extensions are organized into major groups, called *LanguageLevels*. Three LanguageLevels have been defined, numbered 1, 2, and 3. Each LanguageLevel encompasses all features of previous LanguageLevels as well as a significant number of new features. A PostScript interpreter claiming to support a given LanguageLevel must implement all features defined in that LanguageLevel and lower. Thus, for example, a feature identified in this book as "LanguageLevel 2" is understood to be available in all LanguageLevel 3 implementations as well.

This book documents the entire PostScript language, which consists of three distinct groups of features. Features that are part of the LanguageLevel 2 or LanguageLevel 3 additions are clearly identified as such. Features that are not otherwise identified are LanguageLevel 1.

A PostScript interpreter can also support extensions that are not part of its base LanguageLevel. Some such extensions are specialized to particular applications, while others are of general utility and are candidates for inclusion in a future LanguageLevel.

The most significant special-purpose extension is the set of features for the Display PostScript® system. Those features enable workstation applications to use the PostScript language and the Adobe imaging model for managing the appearance of the display and for interacting with the workstation's windowing system. The Display PostScript extensions were documented in the second edition of this book but have been removed for this edition. Further information is available in the *Display PostScript System* manuals.

Appendix D describes strategies for writing PostScript programs that can run compatibly on interpreters supporting different LanguageLevels. With some care, a program can take advantage of features in a higher LanguageLevel when available but will still run acceptably when those features are not available.

1.3 LanguageLevel 3 Overview

In addition to unifying many previous PostScript language extensions, LanguageLevel 3 introduces a number of new features. This section summarizes those features, for the benefit of readers who are already familiar with LanguageLevel 2.

- *Functions*. A PostScript function is a self-contained, static description of a mathematical function having one or more arguments and one or more results.
- *Filters*. Three filters have been added, named **FlateDecode**, **FlateEncode**, and **ReusableStreamDecode**. Some existing filters accept additional optional parameters.
- *Idiom recognition*. The **bind** operator can find and replace certain commonly occurring procedures, called *idioms*, typically appearing in application prologs. The substituted procedure achieves equivalent results with significantly improved performance or quality. This enables LanguageLevel 3 features to work in applications that have not yet been modified to use those features directly.
- *Clipping path stack*. The **clipsave** and **cliprestore** operators save and restore just the clipping path without affecting the rest of the graphics state.
- *Color spaces*. Three color spaces have been added: **CIEBasedDEF** and **CIEBased–DEFG** provide increased flexibility for specifying device-independent colors; **DeviceN** provides a means of specifying high-fidelity and multitone colors.
- *Color space substitution*. Colors that have been specified in **DeviceGray**, **DeviceRGB**, or **DeviceCMYK** color spaces can be remapped into CIE-based color spaces. This capability can be useful in a variety of circumstances, such as for redirecting output intended for one device to a different one or for producing CIE-based colors from an application that generates LanguageLevel 1 output only (and thus is unable to specify them directly).
- *Smooth shading*. It is now possible to paint with a color that varies smoothly over the object or region being painted.
- *Masked images*. A sampled image can be clipped by a mask as it is painted. The mask can be represented explicitly or encoded with a color key in the image data. This enables the background to show through parts of the image.
- *CID-keyed fonts*. This font organization provides a convenient and efficient means for defining multiple-byte character encodings and for creating base fonts containing a very large number of character descriptions.

- *Font formats*. Support has been added for additional types of base fonts, including CFF (Compact Font Format), Chameleon®, TrueType™, and bitmap fonts.
- *Device setup*. There are many additional page device parameters to control colorant selection, finishing options, and other features. Any device can now produce arbitrary separations, even in a monochrome printing system (which can mark only one colorant at a time).
- *In-RIP trapping*. Certain products support *trapping*, which is the automatic generation of overlaps to correct for colorant misregistration during the printing process.
- *Color rendering intent*. A PostScript program can specify a *rendering intent* for color reproduction, causing automatic selection of an appropriate CIE-based color rendering dictionary.
- *Halftones*. Several standard halftone types have been added. They include 16-bit threshold arrays and more flexible tiling organizations for improved color accuracy on high-resolution devices. Halftone supercells increase the number of gray levels achievable on low-resolution devices.

1.4 Related Publications

A number of publications related to this book are listed in the Bibliography; some notable ones are mentioned here. For more details, see the Bibliography.

1.4.1 The Supplement

The *PostScript Language Reference Supplement* documents PostScript language extensions that are available in certain releases of Adobe PostScript® software. A new edition of the *Supplement* is published along with each major release of Adobe PostScript software.

The *Supplement* documents three major classes of extensions:

- New PostScript language features that have been introduced since the most recent LanguageLevel and that are candidates for inclusion in the next LanguageLevel.
- Extensions for controlling unique features of products, such as communication parameters, print engine options, and so on. Certain PostScript language features, such as **setdevparams**, **setpagedevice**, and the named resource facility,

are designed to be extended in this way. Although the framework for this is a standard part of the PostScript language, the specific extensions are product-dependent.

- LanguageLevel 1 compatibility operators, principally in the **statusdict** dictionary. Those features were the LanguageLevel 1 means for controlling unique features of products, but they have been superseded. They are not formally a part of the PostScript language, but many of them are still supported in Adobe PostScript interpreters as a concession to existing applications that depend on them.

1.4.2 Font Formats

PostScript interpreters support several standard formats for font programs, including Adobe Type 1, CFF (Compact Font Format), TrueType, and CID-keyed fonts. The PostScript language manifestations of those fonts are documented in this book. However, the specifications for the font files themselves are published separately, because they are highly specialized and are of interest to a different user community. A variety of Adobe publications are available on the subject of font formats, most notably the following:

- *Adobe Type 1 Font Format* and Adobe Technical Note #5015, *Type 1 Font Format Supplement*
- Adobe Technical Note #5176, *The Compact Font Format Specification*
- Adobe Technical Note #5012, *The Type 42 Font Format Specification*
- Adobe Technical Note #5014, *Adobe CMap and CID Font Files Specification*

1.4.3 Document Structure

Some conventions have been established for the structure of PostScript programs that are to be treated as documents. Those conventions, while not formally part of the PostScript language, are highly recommended, since they enable interoperability with applications that pay attention to them.

- Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*, describes a convention for structuring PostScript page descriptions to facilitate their handling and processing by other programs.

- Adobe Technical Note #5002, *Encapsulated PostScript File Format Specification*, describes a format that enables applications to treat each other's output as included illustrations.

1.4.4 Portable Document Format (PDF)

Adobe has specified another format, PDF, for portable representation of electronic documents. PDF is documented in the *Portable Document Format Reference Manual.*

PDF and the PostScript language share the same underlying Adobe imaging model. A document can be converted straightforwardly between PDF and the PostScript language; the two representations produce the same output when printed. However, PDF lacks the general-purpose programming language framework of the PostScript language. A PDF document is a static data structure that is designed for efficient random access and includes navigational information suitable for interactive viewing.

1.5 Copyrights and Trademarks

The general *idea* of using a page description language is in the public domain. Anyone is free to devise his or her own set of unique commands that constitute a page description language. However, Adobe Systems Incorporated owns the copyright for the list of operators and the written specification for Adobe's PostScript language. Thus, these elements of the PostScript language may not be copied without Adobe's permission. Additionally, Adobe owns the trademark "PostScript," which is used to identify both the PostScript language and Adobe's PostScript software.

Adobe will enforce its copyright and trademark rights. Adobe's intentions are to:

- Maintain the integrity of the PostScript language standard. This enables the public to distinguish between the PostScript language and other page description languages.
- Maintain the integrity of "PostScript" as a trademark. This enables the public to distinguish between Adobe's PostScript interpreter and other interpreters that can execute PostScript language programs.

However, Adobe desires to promote the use of the PostScript language for information interchange among diverse products and applications. Accordingly, Adobe gives permission to anyone to:

- Write programs in the PostScript language.
- Write drivers to generate output consisting of PostScript language commands.
- Write software to interpret programs written in the PostScript language.
- Copy Adobe's copyrighted list of commands to the extent necessary to use the PostScript language for the above purposes.

The only condition of such permission is that anyone who uses the copyrighted list of commands in this way must include an appropriate copyright notice. This limited right to use the copyrighted list of commands does not include a right to copy this book, other copyrighted publications from Adobe, or the software in Adobe's PostScript interpreter, in whole or in part.

The trademark PostScript® (or a derivative trademark, such as PostScript® 3™) may not be used to identify any product not originating from or licensed by Adobe. However, it is acceptable for a non-Adobe product to be described as being PostScript-compatible and supporting a specific LanguageLevel, assuming that the claim is true.

CHAPTER 2

Basic Ideas

OBTAINING A COMPLETE UNDERSTANDING of the PostScript language requires considering it from several points of view:

- As a general-purpose programming language with powerful built-in graphics primitives
- As a page description language that includes programming features
- As an interactive system for controlling raster output devices (printers and displays)
- As an application- and device-independent interchange format for page descriptions

This chapter presents some basic ideas that are essential to understanding the problems the PostScript language is designed to solve and the environments in which it is designed to operate. Terminology introduced here appears throughout the manual.

2.1 Raster Output Devices

Much of the power of the PostScript language derives from its ability to deal with the general class of *raster output devices.* This class encompasses such technology as laser, dot-matrix, and ink-jet printers, digital imagesetters, and raster scan displays.

The defining property of a raster output device is that a printed or displayed image consists of a rectangular array of dots, called *pixels* (picture elements), that can be addressed individually. On a typical black-and-white output device, each pixel can be made either black or white. On certain devices, each pixel can be set

to an intermediate shade of gray or to some color. The ability to individually set the colors of pixels means that printed or displayed output can include text, arbitrary graphical shapes, and reproductions of sampled images.

The *resolution* of a raster output device is a measure of the number of pixels per unit of distance along the two linear dimensions. Resolution is typically—but not necessarily—the same horizontally and vertically.

Manufacturers' decisions on device technology and price/performance tradeoffs create characteristic ranges of resolution:

- Computer displays have relatively low resolution, typically 75 to 110 pixels per inch.
- Dot-matrix printers generally range from 100 to 250 pixels per inch.
- Ink-jet and laser-scanned xerographic printing technologies are capable of medium-resolution output of 300 to 1400 pixels per inch.
- Photographic technology permits high resolutions of 2400 pixels per inch or more.

Higher resolution yields better quality and fidelity of the resulting output, but is achieved at greater cost. As the technology improves and computing costs decrease, products evolve to higher resolutions.

2.2 Scan Conversion

An abstract graphical element (for example, a line, a circle, a text character, or a sampled image) is rendered on a raster output device by a process known as *scan conversion*. Given a mathematical description of the graphical element, this process determines which pixels to adjust and what values to assign those pixels to achieve the most faithful rendition possible at the device resolution.

The pixels on the page can be represented by a two-dimensional array of pixel values in computer memory. For an output device whose pixels can be only black or white, a single bit suffices to represent each pixel. For a device whose pixels can reproduce gray shades or colors, multiple bits per pixel are required.

Note: *Although the ultimate representation of a printed or displayed page is logically a complete array of pixels, its actual representation in computer memory need not*

consist of one memory cell per pixel. Some implementations use other representations, such as display lists. The Adobe imaging model has been carefully designed not to depend on any particular representation of raster memory.

For each graphical element that is to appear on the page, the scan converter sets the values of the corresponding pixels. When the interpretation of the page description is complete, the pixel values in memory represent the appearance of the page. At this point, a raster output process can make this representation visible on a printed page or a display.

Scan-converting a graphical shape, such as a rectangle or a circle, involves determining which device pixels lie "inside" the shape and setting their values appropriately (for example, setting them to black). Because the edges of a shape do not always fall precisely on the boundaries between pixels, some policy is required for deciding which pixels along the edges are considered to be "inside." Scan-converting a text character is conceptually the same as scan-converting an arbitrary graphical shape; however, characters are much more sensitive to legibility requirements and must meet more rigid objective and subjective measures of quality.

Rendering grayscale elements on a *bilevel* device—one whose pixels can only be black or white—is accomplished by a technique known as *halftoning*. The array of pixels is divided into small clusters according to some pattern (called the *halftone screen*). Within each cluster, some pixels are set to black and some to white in proportion to the level of gray desired at that place on the page. When viewed from a sufficient distance, the individual dots become unnoticeable and the result is a shade of gray. This enables a bilevel raster output device to reproduce shades of gray and to approximate natural images, such as photographs. Some color devices use a similar technique.

2.3 Page Description Languages

Theoretically, an application program could describe any page as a full-page pixel array. But this would be unsatisfactory, because the description would be bulky, the pixel array would be device-dependent, and memory requirements would be beyond the capacity of many personal computers.

A page description language should enable applications to produce files that are relatively compact for storage and transmission, and independent of any particular output device.

2.3.1 Imaging Model

In today's computer printing industry, raster output devices with different properties are proliferating, as are the applications that generate output for those devices. Meanwhile, expectations are also rising; typewriter emulation (text-only output in a single typeface) is no longer adequate. Users want to create, display, and print documents that combine sophisticated typography and graphics.

A high-level *imaging model* enables an application to describe the appearance of pages containing text, graphical shapes, and sampled images in terms of abstract graphical elements rather than in terms of device pixels. Such a description is economical and *device-independent*. It can be used to produce high-quality output on many different printers and displays.

A page description language is a language for expressing an imaging model. An application program produces printed output through a two-stage process:

1. The application generates a device-independent description of the desired output in the page description language.
2. A program controlling a specific raster output device interprets the description and renders it on that device.

The two stages may be executed in different places and at different times; the page description language serves as an *interchange standard* for transmission and storage of printable or displayable documents.

2.3.2 Static versus Dynamic Formats

A page description language may have either a *static* or a *dynamic* format.

- A *static format* provides some fixed set of operations and a syntax for specifying the operations and their arguments. Static formats have been in existence since computers first used printers; classic examples are format control codes for line printers and "format effector" codes in standard character sets. Historically, static formats have been designed to capture the capabilities of a specific class of printing device and have evolved to include new features as needed.
- A *dynamic format* allows more flexibility than a static format. The operator set may be extensible and the exact meaning of an operator may not be known until it is actually encountered. A page described in a dynamic format is a pro-

gram to be executed, rather than data to be consumed. Dynamic page description languages contain elements of programming languages, such as procedures, variables, and control constructs.

The PostScript language design is dynamic. The language includes a set of primitive graphics operators that can be combined to describe the appearance of any printed or displayed page. It has variables and allows arbitrary computations while interpreting the page description. It has a rich set of programming-language control structures for combining its elements.

2.4 Using the PostScript Language

It is important to understand the PostScript interpreter and how it interacts with applications using it.

2.4.1 The Interpreter

The PostScript *interpreter* controls the actions of the output device according to the instructions provided in a PostScript program generated by an application. The interpreter executes the program and produces output on a printer, display, or other raster device.

There are three ways the PostScript interpreter and the application interact (Figure 2.1 illustrates these scenarios):

- In the conventional output-only printing model, the application creates a *page description*—a self-contained PostScript language description of a document. The page description can be sent to the PostScript interpreter immediately or stored for transmission at some other time (via an intermediate print manager or spooler, for example). The interpreter consumes a sequence of page descriptions as "print jobs" and produces the requested output. The output device is typically a printer, but it can be a preview window on a workstation's display. The PostScript interpreter is often implemented on a dedicated processor that has direct control over the raster output device.
- In the integrated display model, an application interacts with the PostScript interpreter controlling a display or windowing system. Instead of a one-way transmission of a page description, a two-way *interactive session* takes place between the application and the interpreter. In response to user actions, the

application issues commands to the PostScript interpreter and sometimes reads information back from it.

- In the interactive programming language model, an interactive session takes place directly between a programmer and the PostScript interpreter; the programmer issues PostScript commands for immediate execution. Many PostScript interpreters (for both printers and displays) have a rudimentary interactive executive to support this mode of use; see Section 2.4.4, "Using the Interpreter Interactively."

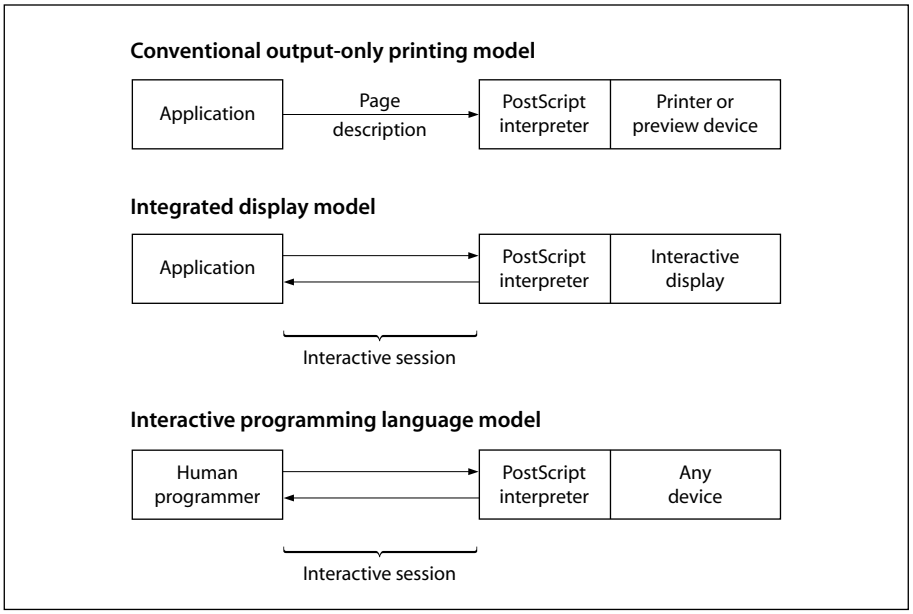


FIGURE 2.1 *How the PostScript interpreter and an application interact*

Even when a PostScript interpreter is being used noninteractively to execute page descriptions prepared previously, there may be some dynamic interactions between the print manager or spooler and the PostScript interpreter. For example, the sender may ask the PostScript interpreter whether certain fonts referenced by a document are available. This is accomplished by sending the interpreter a short program to read and return the information. The PostScript interpreter makes no distinction between a page description and a program that makes environmental queries or performs other arbitrary computations.

To facilitate document interchange and document management, a page description should conform to the structuring conventions discussed below. The structuring conventions do not apply in an interactive session, since there is no notion that the information being communicated represents a document to be preserved for later execution; a session has no obvious overall structure.

2.4.2 Program Structure

A well-structured PostScript page description generally consists of two parts: a prolog followed by a script. There is nothing in the PostScript language that formally distinguishes the prolog from the script or imposes any overall document structure. Such structuring is merely a convention, but one that is quite useful and is recommended for most applications.

- The *prolog* is a set of application-specific procedure definitions that an application may use in the execution of its script. It is included as the first part of every PostScript file generated by the application. It contains definitions that match the output functions of the application with the capabilities supported by the PostScript language.
- The *script* is generated automatically by the application program to describe the specific elements of the pages being produced. It consists of references to PostScript operators and to procedure definitions in the prolog, together with operands and data. The script, unlike the prolog, is usually very stylized, repetitive, and simple.

Dividing a PostScript program into a prolog and a script reduces the size of each page description and minimizes data communication and disk storage. An example may help explain the purpose of a separate prolog and script. One of the most common tasks in a PostScript program is placing text at a particular location on the current page. This is really *two* operations: "moving" the current point to a specific location and "showing" the text. A program is likely to do this often, so it is useful for the prolog to define a procedure that combines the operations:

```
/ms {moveto show} bind def
```

Later, the script can call the ms procedure instead of restating the individual operations:

```
(some text)  100  200  ms
```

The script portion of a printable document ordinarily consists of a sequence of separate pages. The description of an individual page should stand by itself, depending only on the definitions in the prolog and not on anything in previous pages of the script. The language includes facilities (described in Section 3.7, "Memory Management") that can be used to guarantee page independence.

Adobe has established conventions to make document structure explicit. These document structuring conventions appear in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*. Document structure is expressed in PostScript comments; the interpreter pays no attention to them. However, there are good reasons to adhere to the conventions:

- Utility programs can operate on structured documents in various ways: change the order of pages, extract subsets of pages, embed individual pages within other pages, and so on. This is possible only if the original document maintains page independence.
- Print managers and spoolers can obtain useful information from a properly structured document to determine how the document should be handled.
- The structuring conventions serve as a good basis for organizing printing from an application.

An application has its own model of the appearance of printable output that it generates. Some parts of this model are fixed for an entire document or for all documents; the application should incorporate their descriptions into the prolog. Other parts vary from one page to another; the application should produce the necessary descriptions of these as they appear. At page boundaries, the application should generate commands to restore the standard environment defined by the prolog and then explicitly reestablish nonstandard portions of the environment for the next page. This technique ensures that each page is independent of any other.

The structuring conventions also include standard methods for performing environmental queries. These conventions ensure consistent and reliable behavior in a variety of system environments, including those with print spoolers.

2.4.3 Translating from Other Print Formats

Many existing applications generate printable documents in some other print file format or in some intermediate representation. It is possible to print such docu-

ments by translating them into PostScript page descriptions. There are two scenarios in which this need arises:

- An application describes its printable output by making calls to an application programming interface, such as GDI in Microsoft Windows® or QuickDraw™ in the Apple Mac® OS. A software component called a *printer driver* interprets these calls and produces a PostScript page description.
- An application produces printable output directly in some other file format, such as PCL, HPGL, or DVI. A separate program must then translate this file into a PostScript page description.

Implementing a driver or translator is often the least expensive way to interface an existing application to a PostScript printer. Unfortunately, while such translation is usually straightforward, a translator may not be able to generate page descriptions that make the best use of the high-level Adobe imaging model. This is because the information being translated often describes the desired results at a level that is too low; any higher-level information maintained by the original application has been lost and is not available to the translator.

While direct PostScript output from applications is most desirable, translation from another print format may be the only choice available for some applications. A translator should do the best it can to produce output that conforms to the document structuring conventions (see Technical Note #5001). This ensures that such output is compatible with the tools for manipulating PostScript page descriptions.

2.4.4 Using the Interpreter Interactively

Normally, the interpreter executes PostScript programs generated by application programs; a user does not interact with the PostScript interpreter directly. However, many PostScript interpreters provide an *interactive executive* that enables a user to control the interpreter directly. That is, from a terminal or terminal emulator connected directly to the PostScript interpreter, you can issue commands for immediate execution and control the operation of the interpreter in limited ways. This is useful for experimentation and debugging.

To use the interpreter this way, you must first connect your keyboard and display directly to the standard input and output channels of the PostScript interpreter, so that characters you type are sent directly to the interpreter and characters the

interpreter sends appear on the screen. How to accomplish this depends on the product. A typical method is to connect a personal computer running terminal emulation software to a PostScript printer, either by direct physical connection or by establishing communication over a network.

Once the input and output connections are established, you can invoke the interactive executive by typing

```
executive
```

(all lowercase) and pressing the Return key. The interpreter responds with a herald, such as

```
PostScript(r) Version 3010.106
Copyright (c) 1984-1998 Adobe Systems Incorporated.
All Rights Reserved.
PS>
```

The PS> prompt is an indication that the PostScript interpreter is waiting for a command.

Each time you type a complete PostScript statement followed by the Return key, the interpreter executes that statement and then sends another PS> prompt. If the statement causes the interpreter to send back any output (produced by execution of the **print** or **=** operator, for example), that output appears before the PS> prompt. If the statement causes an error to occur, an error message appears before the PS> prompt; control remains in the interactive executive, whereas errors normally cause a job to terminate. The interactive executive remains in operation until you invoke the **quit** operator or enter a channel-dependent end-of-file indication (for example, Control-D for a serial connection).

The interactive executive provides a few simple amenities. While you are typing, the interpreter ordinarily "echoes" the typed characters (sends them back to your terminal so that you can see them). You can use the control characters in Table 2.1 to make corrections while entering a statement.

TABLE 2.1 Control characters for the interactive executive

CHARACTER	FUNCTION
Backspace (BS)	Backs up and erases one character.
Delete (DEL)	Same as backspace.
Control-U	Erases the current line.
Control-R	Redisplays the current line.
Control-C	Aborts the entire statement and starts over. Control-C can also abort a statement that is executing and force the executive to revert to a PS> prompt.

There are several important things to understand about the interactive executive:

- It is intended solely for direct interaction with the user; an application that is generating PostScript programs should never invoke **executive**. In general, a PostScript program will behave differently when sent through the interactive executive than when executed directly by the PostScript interpreter. For example, the executive produces extraneous output such as echoes of the input characters and PS> prompts. Furthermore, a program that explicitly reads data embedded in the program file will malfunction if invoked via the executive, since the executive itself is interpreting the file.
- The user amenities are intentionally minimal. The executive is not a full-scale programming environment; it lacks a text editor and other tools required for program development and it does not keep a record of your interactive session. The executive is useful mainly for experimentation and debugging.
- The **executive** operator is not necessarily available in all PostScript interpreters. Its behavior may vary among different products.

CHAPTER 3

Language

SYNTAX, DATA TYPES, AND EXECUTION SEMANTICS are essential aspects of any PostScript program. Later chapters document the graphics and font capabilities that specialize PostScript programs to the task of controlling the appearance of a printed page. This chapter explains the PostScript language as a *programming* language.

Like all programming languages, the PostScript language builds on elements and ideas from several of the great programming languages. The syntax most closely resembles that of the programming language FORTH. It incorporates a *postfix* notation in which operators are preceded by their operands. The number of special characters is small and there are no reserved words.

Note: *Although the number of built-in operators is large, the names that represent operators are not reserved by the language. A PostScript program may change the meanings of operator names.*

The data model includes elements, such as numbers, strings, and arrays, that are found in many modern programming languages. It also includes the ability to treat programs as data and to monitor and control many aspects of the language's execution state; these notions are derived from programming languages such as LISP.

The PostScript language is relatively simple. It derives its power from the ability to combine these features in unlimited ways without arbitrary restrictions. Though you may seldom fully exploit this power, you can design sophisticated graphical applications that would otherwise be difficult or impossible.

Because this is a reference book and not a tutorial, this chapter describes each aspect of the language systematically and thoroughly before moving on to the next.

It begins with a brief overview of the PostScript interpreter. The following sections detail the syntax, data types, execution semantics, memory organization, and general-purpose operators of the PostScript language (excluding those that deal with graphics and fonts). The final sections cover file input and output, named resources, function dictionaries, errors, how the interpreter evaluates name objects, and details on filtered files and binary encoding.

3.1 Interpreter

The PostScript interpreter executes the PostScript language according to the rules in this chapter. These rules determine the order in which operations are carried out and how the pieces of a PostScript program fit together to produce the desired results.

The interpreter manipulates entities called PostScript *objects*. Some objects are data, such as numbers, boolean values, strings, and arrays. Other objects are elements of programs to be executed, such as names, operators, and procedures. However, there is not a distinction between data and programs; any PostScript object may be treated as data or be executed as part of a program.

The interpreter operates by executing a sequence of objects. The effect of executing a particular object depends on that object's *type*, *attributes*, and *value*. For example, executing a number object causes the interpreter to push a copy of that object on the operand stack (to be described shortly). Executing a name object causes the interpreter to look up the name in a dictionary, fetch the associated value, and execute it. Executing an operator object causes the interpreter to perform a built-in action, such as adding two numbers or painting characters in raster memory.

The objects to be executed by the interpreter come from two principal sources:

- A character stream may be scanned according to the syntax rules of the PostScript language, producing a sequence of new objects. As each object is scanned, it is immediately executed. The character stream may come from an external source, such as a file or a communication channel, or it may come from a string object previously stored in the PostScript interpreter's memory.
- Objects previously stored in an array in memory may be executed in sequence. Such an array is known as a *procedure*.

The interpreter can switch back and forth between executing a procedure and scanning a character stream. For example, if the interpreter encounters a name in a character stream, it executes that name by looking it up in a dictionary and retrieving the associated value. If that value is a procedure object, the interpreter suspends scanning the character stream and begins executing the objects in the procedure. When it reaches the end of the procedure, it resumes scanning the character stream where it left off. The interpreter maintains an *execution stack* for remembering all of its suspended execution contexts.

3.2 Syntax

As the interpreter scans the text of a PostScript program, it creates various types of PostScript objects, such as numbers, strings, and procedures. This section discusses only the *syntactic* representation of such objects. Their internal representation and behavior are covered in Section 3.3, "Data Types and Objects."

There are three encodings for the PostScript language: *ASCII*, *binary token*, and *binary object sequence*. The ASCII encoding is preferred for expository purposes (such as this book), for archiving documents, and for transmission via communications facilities, because it is easy to read and does not rely on any special characters that might be reserved for communications use. The two binary encodings are usable in controlled environments to improve the efficiency of representation or execution; they are intended exclusively for machine generation. Detailed information on the binary encodings is provided in Section 3.14, "Binary Encoding Details."

3.2.1 Scanner

The PostScript language differs from most other programming languages in that it does not have any syntactic entity for a "program," nor is it necessary for an entire "program" to exist in one place at one time. There is no notion of "reading in" a program before executing it. Instead, the PostScript interpreter *consumes* a program by reading and executing one syntactic entity at a time. From the interpreter's point of view, the program has no permanent existence. Execution of the program may have side effects in the interpreter's memory or elsewhere. These side effects may include the creation of procedure objects in memory that are intended to be invoked later in the program; their execution is *deferred*.

It is not correct to think that the PostScript interpreter "executes" the character stream directly. Rather, a scanner groups characters into *tokens* according to the PostScript language syntax rules. It then assembles one or more tokens to create a PostScript *object*—in other words, a data value in the interpreter's memory. Finally, the interpreter executes the object.

For example, when the scanner encounters a group of consecutive digits surrounded by spaces or other separators, it assembles the digits into a token and then converts the token into a number object represented internally as a binary integer. The interpreter then executes this number object; in this case, it pushes a copy of the object on the operand stack.

3.2.2 ASCII Encoding

The standard character set for ASCII-encoded PostScript programs is the visible printable subset of the ASCII character set, plus characters that appear as "white space," such as space, tab, and newline characters. ASCII is the American Standard Code for Information Interchange, a widely used convention for encoding characters as binary numbers. ASCII encoding does not prohibit the use of characters outside this set, but such use is not recommended, because it impairs portability and may make transmission and storage of PostScript programs more difficult.

Note: *Control characters are often usurped by communications functions. Control codes are device-dependent—not part of the PostScript language. For example, the serial communication protocol supported by many products uses the Control-D character as an end-of-file indication. In such cases, Control-D is a communications function and should not be part of a PostScript program.*

White-space characters (Table 3.1) separate syntactic constructs such as names and numbers from each other. The interpreter treats any number of consecutive white-space characters as if there were just one. All white-space characters are equivalent, except in comments and strings.

The characters carriage return (CR) and line feed (LF) are also called *newline* characters. The combination of a carriage return followed immediately by a line feed is treated as one newline.

TABLE 3.1 White-space characters

OCTAL	HEXADECIMAL	DECIMAL	NAME
000	00	0	Null (nul)
011	09	9	Tab (tab)
012	0A	10	Line feed (LF)
014	0C	12	Form feed (FF)
015	0D	13	Carriage return (CR)
040	20	32	Space (SP)

The characters (,), <, >, [,], {, }, /, and % are special. They delimit syntactic entities such as strings, procedure bodies, name literals, and comments. Any of these characters terminates the entity preceding it and is not included in the entity.

All characters besides the white-space characters and delimiters are referred to as *regular characters*. These include nonprinting characters that are outside the recommended PostScript ASCII character set.

Comments

Any occurrence of the character % outside a string introduces a *comment*. The comment consists of all characters between the % and the next newline or form feed, including regular, delimiter, space, and tab characters.

The scanner ignores comments, treating each one as if it were a single white-space character. That is, a comment separates the token preceding it from the one following. Thus the ASCII-encoded program fragment

```
abc% comment {/%) blah blah blah
123
```

is treated by the scanner as just two tokens: abc and 123.

Numbers

Numbers in the PostScript language include:

- Signed integers, such as

  ```
  123 –98 43445 0 +17
  ```

- Real numbers, such as

  ```
  –.002 34.5 –3.62 123.6e10 1.0E–5 1E6 –1. 0.0
  ```

- Radix numbers, such as

  ```
  8#1777 16#FFFE 2#1000
  ```

An integer consists of an optional sign followed by one or more decimal digits. The number is interpreted as a signed decimal integer and is converted to an integer object. If it exceeds the implementation limit for integers, it is converted to a real object. (See Appendix B for implementation limits.)

A real number consists of an optional sign and one or more decimal digits, with an embedded period (decimal point), a trailing exponent, or both. The exponent, if present, consists of the letter `E` or `e` followed by an optional sign and one or more decimal digits. The number is interpreted as a real number and is converted to a real (floating-point) object. If it exceeds the implementation limit for real numbers, a **limitcheck** error occurs.

A radix number takes the form *base*#*number*, where *base* is a decimal integer in the range 2 through 36. *number* is interpreted in this base; it must consist of digits ranging from 0 to *base* – 1. Digits greater than 9 are represented by the letters `A` through `Z` (or `a` through `z`). The number is treated as an unsigned integer and is converted to an integer object having the same twos-complement binary representation. This notation is intended for specifying integers in a nondecimal radix, such as binary, octal, or hexadecimal. If the number exceeds the implementation limit for integers, a **limitcheck** error occurs.

Strings

There are three conventions for quoting a literal string object:

- As literal text, enclosed in (and)
- As hexadecimal data, enclosed in < and >
- As ASCII base-85 data, enclosed in <~ and ~> *(LanguageLevel 2)*

Literal Text Strings

A literal text string consists of an arbitrary number of characters enclosed in (and). Any characters may appear in the string other than (,), and \, which must be treated specially. Balanced pairs of parentheses in the string require no special treatment.

The following lines show several valid strings:

```
(This is a string)
(Strings may contain newlines
and such.)
(Strings may contain special characters *!&}^% and
balanced parentheses ( ) (and so on).)
(The following is an empty string.)
()
(It has 0 (zero) length.)
```

Within a text string, the \ (backslash) character is treated as an "escape" for various purposes, such as including newline characters, unbalanced parentheses, and the \ character itself in the string. The character immediately following the \ determines its precise interpretation.

\n	line feed (LF)
\r	carriage return (CR)
\t	horizontal tab
\b	backspace
\f	form feed
\\	backslash
\(	left parenthesis
\)	right parenthesis
ddd	character code *ddd* (octal)

If the character following the \ is not in the preceding list, the scanner ignores the \. If the \ is followed immediately by a newline (CR, LF, or CR-LF pair), the scanner ignores both the initial \ and the newline; this breaks a string into multiple lines without including the newline character as part of the string, as in the following example:

```
(These \
two strings \
are the same.)
(These two strings are the same.)
```

But if a newline appears without a preceding \, the result is equivalent to \n. For example:

```
(This string has a newline at the end of it.
)
(So does this one.\n)
```

For more information about end-of-line conventions, see Section 3.8, "File Input and Output."

The *ddd* form may be used to include any 8-bit character constant in a string. One, two, or three octal digits may be specified, with high-order overflow ignored. This notation is preferred for specifying a character outside the recommended ASCII character set for the PostScript language, since the notation itself stays within the standard set and thereby avoids possible difficulties in transmitting or storing the text of the program. It is recommended that three octal digits always be used, with leading zeros as needed, to prevent ambiguity. The string (\0053), for example, contains two characters—an ASCII 5 (Control-E) followed by the digit 3—whereas the strings (\53) and (\053) contain one character, the ASCII character whose code is octal 53 (plus sign).

Hexadecimal Strings

A hexadecimal string consists of a sequence of hexadecimal digits (0–9 and either A–F or a–f) enclosed within < and >. Each pair of hexadecimal digits defines one character of the string. White-space characters are ignored. If a hexadecimal string contains characters outside the allowed character set, a **syntaxerror** occurs. Hexadecimal strings are useful for including arbitrary binary data as literal text.

If the final digit of a given hexadecimal string is missing—in other words, if there is an odd number of digits—the final digit is assumed to be 0. For example, <901fa3> is a 3-character string containing the characters whose hexadecimal codes are 90, 1f, and a3, but <901fa> is a 3-character string containing the characters whose hexadecimal codes are 90, 1f, and a0.

ASCII Base-85 Strings

An ASCII base-85 string *(LanguageLevel 2)* consists of a sequence of printable ASCII characters enclosed in <~ and ~>. This notation represents arbitrary binary data using an encoding technique that produces a 4:5 expansion as opposed to the 1:2 expansion for hexadecimal. The ASCII base-85 encoding algorithm is described under "ASCII85Encode Filter" on page 131. If an ASCII base-85 string is malformed, a **syntaxerror** occurs.

Names

Any token that consists entirely of regular characters and cannot be interpreted as a number is treated as a *name* object (more precisely, an *executable* name). All characters except delimiters and white-space characters can appear in names, including characters ordinarily considered to be punctuation.

The following are examples of valid names:

```
abc Offset $$ 23A 13–456 a.b $MyDict @pattern
```

Use care when choosing names that begin with digits. For example, while 23A is a valid name, 23E1 is a real number, and 23#1 is a radix number token that represents an integer.

A / (slash—*not* backslash) introduces a *literal* name. The slash is not part of the name itself, but is a prefix indicating that the following sequence of zero or more regular characters constitutes a literal name object. There can be no white-space characters between the / and the name. The characters // (two slashes) introduce an *immediately evaluated name*. The important properties and uses of names and the distinction between executable and literal names are described in Section 3.3, "Data Types and Objects"; immediately evaluated names are discussed in Section 3.12.2, "Immediately Evaluated Names."

Note: *The token / (a slash followed by no regular characters) is a valid literal name.*

Arrays

The characters [and] are self-delimiting tokens that specify the construction of an array. For example, the program fragment

```
[ 123 /abc (xyz) ]
```

results in the construction of an array object containing the integer object 123, the literal name object abc, and the string object xyz. Each token within the brackets is executed in turn.

The [and] characters are special syntax for names that, when executed, invoke PostScript operators that collect objects and construct an array containing them. Thus the example

```
[ 123 /abc (xyz) ]
```

contains these five tokens:

- The name object [
- The integer object 123
- The literal name object abc
- The string object xyz
- The name object]

When the example is executed, a sixth object (the array) results from executing the [and] name objects.

Procedures

The special characters { and } delimit an *executable array*, otherwise known as a *procedure*. The syntax is superficially similar to that for the array construction operators **[** and **]**; however, the semantics are entirely different and arise as a result of *scanning* the procedure rather than *executing* it.

Scanning the program fragment

```
{add 2 div}
```

produces a single procedure object that contains the name object **add**, the integer object 2, and the name object **div**. When the scanner encounters the initial {, it continues scanning and creating objects, but the interpreter does not execute them. When the scanner encounters the matching }, it puts all the objects created since the initial { into a new executable array (procedure) object.

The interpreter does not execute a procedure immediately, but treats it as data; it pushes the procedure on the operand stack. Only when the procedure is explicitly invoked (by means yet to be described) will it be executed. Execution of the procedure—and of all objects within the procedure, including any embedded procedures—has been *deferred*. The matter of immediate versus deferred execution is discussed in Section 3.5, "Execution."

The procedure object created by { and } is either an array or a packed array, according to the current setting of a mode switch. The distinction between these array types is discussed in Section 3.3, "Data Types and Objects."

Dictionaries

The special character sequences << and >> *(LanguageLevel 2)* are self-delimiting tokens that denote the construction of a dictionary, much the same as [and] denote the construction of an array. They are intended to be used as follows:

<< key_1 $value_1$ key_2 $value_2$ … key_n $value_n$ >>

This creates a dictionary containing the bracketed key-value pairs and pushes it on the operand stack. Dictionaries are introduced in Section 3.3, "Data Types and Objects."

<< and >> are merely special names for operators that, when executed, cause a dictionary to be constructed. They are like the **[** and **]** array construction operators, but unlike the { and } delimiters for procedure literals.

The << and >> tokens are self-delimiting, so they need not be surrounded by white-space characters or other delimiters. Do not confuse these tokens with < and >, which delimit a hexadecimal string literal, or <~ and ~>, which delimit an ASCII base-85 string literal. The << and >> tokens are objects in their own right (specifically, name objects), whereas in < … > and <~ … ~> the delimiting characters are merely punctuation for the enclosed literal string objects.

3.3 Data Types and Objects

All data accessible to PostScript programs, including procedures that are part of the programs themselves, exists in the form of *objects*. Objects are produced, manipulated, and consumed by the PostScript operators. They are also created by the scanner and executed by the interpreter.

Each object has a *type,* some *attributes,* and a *value*. Objects contain their own dynamic types; that is, an object's type is a property of the object itself, not of where it is stored or what it is called. Table 3.2 lists all the object types supported by the PostScript language. Extensions to the language may introduce additional object types. The distinction between simple and composite objects is explained below.

TABLE 3.2 Types of objects

SIMPLE OBJECTS	COMPOSITE OBJECTS
boolean	array
fontID	dictionary
integer	file
mark	gstate *(LanguageLevel 2)*
name	packedarray *(LanguageLevel 2)*
null	save
operator	string
real	

3.3.1 Simple and Composite Objects

Objects of most types are simple, atomic entities. An atomic object is always constant—a 2 is always a 2. There is no visible substructure in the object; the type, attributes, and value are irrevocably bound together and cannot be changed.

However, objects of certain types indicated in Table 3.2 are *composite*. Their values are separate from the objects themselves; for some types of composite object, the values have internal substructure that is visible and can sometimes be

modified selectively. The details of the substructures are presented later in the descriptions of these individual types.

An important distinction between simple and composite objects is the behavior of operations that *copy* objects. *Copy* refers to any operation that transfers the contents of an object from one place to another in the memory of the PostScript interpreter. "Fetching" and "storing" objects are copying operations. It is possible to derive a new object by copying an existing one, perhaps with modifications.

When a simple object is copied, all of its parts (type, attributes, and value) are copied together. When a composite object is copied, the value is not copied; instead, the original and copy objects *share* the same value. Consequently, any changes made to the substructure of one object's value also appear as part of the other object's value.

The sharing of composite objects' values in the PostScript language corresponds to the use of pointers in system programming languages such as C and Pascal. Indeed, the PostScript interpreter uses pointers to implement shared values: a composite object contains a pointer to its value. However, the PostScript language does not have any explicit notion of a pointer. It is better to think in terms of the copying and sharing notions presented here.

The values of simple objects are contained in the objects themselves. The values of composite objects reside in a special region of memory called *virtual memory* or *VM*. Section 3.7, "Memory Management," describes the behavior of VM.

3.3.2 Attributes of Objects

In addition to type and value, each object has one or more *attributes*. These attributes affect the behavior of the object when it is executed or when certain operations are performed on it. They do not affect its behavior when it is treated strictly as data; so, for example, two integers with the same value are considered "equal" even if their attributes differ.

Literal and Executable

Every object is either *literal* or *executable*. This distinction comes into play when the interpreter attempts to execute the object.

- If the object is *literal*, the interpreter treats it strictly as data and pushes it on the operand stack for use as an operand of some subsequent operator.
- If the object is *executable*, the interpreter executes it.

What it means to execute an object depends on the object's type; this is described in Section 3.5, "Execution." For some object types, such as integers, execution consists of pushing the object on the operand stack; the distinction between literal and executable integers is meaningless. But for other types, such as names, operators, and arrays, execution consists of performing a different action.

- Executing an *executable name* causes it to be looked up in the current dictionary context and the associated value to be executed.
- Executing an *executable operator* causes some built-in action to be performed.
- Executing an *executable array* (otherwise known as a procedure) causes the elements of the array to be executed in turn.

As described in Section 3.2, "Syntax," some tokens produce literal objects and some produce executable ones.

- Integer, real, and string constants are always literal objects.
- Names are literal if they are preceded by **/** and executable if they are not.
- The **[** and **]** operators, when executed, produce a literal array object with the enclosed objects as elements. Likewise, **<<** and **>>** *(LanguageLevel 2)* produce a literal dictionary object.
- { and } enclose an executable array or procedure.

Note: *As mentioned above, it does not matter whether an object is literal or executable when it is accessed as data, only when it is executed. However, referring to an executable object by name often causes that object to be executed automatically; see Section 3.5.5, "Execution of Specific Types." To avoid unintended behavior, it is best to use the executable attribute only for objects that are meant to be executed, such as procedures.*

Access

The other attribute of an object is its *access*. Only composite objects have access attributes, which restrict the set of operations that can be performed on the object's value.

There are four types of access. In increasing order of restriction, they are:

1. *Unlimited*. Normally, objects have unlimited access: all operations defined for that object are allowed. However, packed array objects always have read-only (or even more restricted) access.
2. *Read-only*. An object with read-only access may not have its value written, but may still be read or executed.
3. *Execute-only*. An object with execute-only access may not have its value either read or written, but may still be executed by the PostScript interpreter.
4. *None*. An object with no access may not be operated on in any way by a PostScript program. Such objects are not of any direct use to PostScript programs, but serve internal purposes that are not documented in this book.

The literal/executable distinction and the access attribute are entirely independent, although there are combinations that are not of any practical use (for example, a literal array that is execute-only).

With one exception, attributes are properties of an *object* itself and not of its *value*. Two composite objects can share the same value but have different literal/executable or access attributes. The exception is the dictionary type: a dictionary's access attribute is a property of the value, so multiple dictionary objects sharing the same value have the same access attribute.

3.3.3 Integer and Real Objects

The PostScript language provides two types of numeric object: *integer* and *real*. Integer objects represent mathematical integers within a certain interval centered at 0. Real objects approximate mathematical real numbers within a much larger interval, but with limited precision; they are implemented as floating-point numbers.

Most PostScript arithmetic and mathematical operators can be applied to numbers of both types. The interpreter performs automatic type conversion when necessary. Some operators expect only integers or a subrange of the integers as operands. There are operators to convert from one data type to another explicitly. Throughout this book, *number* means an object whose type is either integer or real.

The range and precision of numbers is limited by the internal representations used in the machine on which the PostScript interpreter is running. Appendix B gives these limits for typical implementations of the PostScript interpreter.

Note: *The machine representation of integers is accessible to a PostScript program through the bitwise operators. However, the representation of integers may depend on the CPU architecture of the implementation. The machine representation of real numbers is not accessible to PostScript programs.*

3.3.4 Boolean Objects

The PostScript language provides boolean objects with values *true* and *false* for use in conditional and logical expressions. The names **true** and **false** are associated with values of this type. Boolean objects are the results of the relational (comparison) and logical operators. Various other operators return them as status information. Boolean objects are mainly used as operands for the control operators **if** and **ifelse**.

3.3.5 Array Objects

An *array* is a one-dimensional collection of objects accessed by a numeric index. Unlike arrays in many other computer languages, PostScript arrays may be heterogeneous; that is, an array's elements may be any combination of numbers, strings, dictionaries, other arrays, or any other objects. A *procedure* is an array that can be executed by the PostScript interpreter.

All arrays are indexed from 0, so an array of n elements has indices from 0 through $n - 1$. All accesses to arrays are bounds-checked, and a reference with an out-of-bounds index results in a **rangecheck** error. The length of an array is subject to an implementation limit; see Appendix B.

The PostScript language directly supports only one-dimensional arrays. Arrays of higher dimension can be constructed by using arrays as elements of arrays, nested to any depth.

As discussed earlier, an array is a composite object. When an array object is copied, the value is *not* copied. Instead, the old and new objects share the same value. Additionally, there is an operator (**getinterval**) that creates a new array object whose value is a subinterval of an existing array; the old and new objects share the array elements in that subinterval.

3.3.6 Packed Array Objects

A *packed array* is a more compact representation of an ordinary array, intended primarily for use as a procedure. A packed array object is distinct from an ordinary array object (it has type *packedarray* instead of *array*), but in most respects it behaves the same as an ordinary array. Its principal distinguishing feature is that it usually occupies much less space in memory (see Section B.2, "Virtual Memory Use").

Throughout this book, any mention of a procedure may refer to either an executable array or an executable packed array. The two types of array are not distinguishable when they are executed, only when they are treated as data. See the introduction to the array operators in Section 3.6, "Overview of Basic Operators."

3.3.7 String Objects

A *string* is similar to an array, but its elements must be integers in the range 0 to 255. The string elements are not integer objects, but are stored in a more compact format. However, the operators that access string elements accept or return ordinary integer objects with values in the range 0 to 255. The length of a string is subject to an implementation limit; see Appendix B.

String objects are conventionally used to hold text, one character per string element. However, the PostScript language does not have a distinct "character" syntax or data type and does not require that the integer elements of a string encode any particular character set. String objects may also be used to hold arbitrary binary data.

To enhance program portability, strings appearing literally as part of a PostScript program should be limited to characters from the printable ASCII character set, with other characters inserted by means of the *\ddd* escape convention (see Section 3.2.2, "ASCII Encoding"). ASCII text strings are fully portable; ASCII base-85 text strings are fully portable among LanguageLevel 2 and LanguageLevel 3 PostScript interpreters.

Like an array, a string is a composite object. Copying a string object or creating a subinterval (substring) results in sharing the string's value.

3.3.8 Name Objects

A *name* is an atomic symbol uniquely defined by a sequence of characters. Names serve the same purpose as "identifiers" in other programming languages: as tags for variables, procedures, and so on. However, PostScript names are not just language artifacts, but are first-class data objects, similar to "atoms" in LISP.

A name object is ordinarily created when the scanner encounters a PostScript token consisting entirely of regular characters, perhaps preceded by /, as described in Section 3.2, "Syntax." However, a name may also be created by explicit conversion from a string, so there is no restriction on the set of characters that can be included in names. The length of a name, however, is subject to an implementation limit; see Appendix B.

Unlike a string, a name is a *simple* object not made up of other objects. Although a name is defined by a sequence of characters, those characters are not "elements" of the name. A name object, although logically simple, does have an invisible "value" that occupies space in VM.

A name is *unique*. Any two name objects defined by the same sequence of characters are identical copies of each other. Name equality is based on an exact match between the corresponding characters defining each name. The case of letters must match, so the names `A` and `a` are different. Literal and executable objects can be equal, however.

The interpreter can efficiently determine whether two existing name objects are equal without comparing the characters that define the names. This makes names useful as keys in dictionaries.

Names do not *have* values, unlike variable or procedure names in other programming languages. However, names can be *associated* with values in dictionaries.

3.3.9 Dictionary Objects

A *dictionary* is an associative table whose entries are pairs of PostScript objects. The first element of an entry is the *key* and the second element is the *value*. The PostScript language includes operators that insert an entry into a dictionary, look up a key and fetch the associated value, and perform various other operations.

Keys are normally name objects. The PostScript syntax and the interpreter are optimized for this most common case. However, a key may be any PostScript object except **null** (defined later). If you attempt to use a string as a key, the PostScript interpreter will first convert the string to a name object; thus, *strings and names are interchangeable when used as keys in dictionaries*. Consequently, a string used as a dictionary key is subject to the implementation limit on the length of a name.

A dictionary has the capacity to hold a certain maximum number of entries; the capacity is specified when the dictionary is created. PostScript interpreters of different LanguageLevels differ in their behavior when a program attempts to insert an entry into a dictionary that is full: in LanguageLevel 1, a **dictfull** error occurs; in LanguageLevels 2 and 3, the interpreter enlarges the dictionary automatically. The length of a dictionary is also subject to an implementation limit; see Appendix B.

Dictionaries ordinarily associate the names and values of a program's components, such as variables and procedures. This association corresponds to the conventional use of identifiers in other programming languages. But there are many other uses for dictionaries. For example, a PostScript font program contains a dictionary that associates the names of characters with the procedures for drawing those characters' shapes (see Chapter 5).

There are three primary methods for accessing dictionaries:

- Operators exist to access a specific dictionary supplied as an operand.
- There is a *current dictionary* and a set of operators to access it implicitly.
- The interpreter automatically looks up executable names it encounters in the program being executed.

The interpreter maintains a *dictionary stack* defining the current dynamic name space. Dictionaries may be pushed on and popped off the dictionary stack at will. The topmost dictionary on the stack is the *current dictionary*.

When the interpreter looks up a key implicitly—for example, when it executes a name object—it searches for the key in the dictionaries on the dictionary stack. It searches first in the topmost dictionary, then in successively lower dictionaries on the dictionary stack, until it either finds the key or exhausts the dictionary stack.

In LanguageLevel 1, there are two built-in dictionaries permanently on the dictionary stack; they are called **systemdict** and **userdict**. In LanguageLevels 2 and 3, there are three dictionaries: **systemdict**, **globaldict**, and **userdict**.

- **systemdict** is a read-only dictionary that associates the names of all the PostScript operators (those defined in this book) with their values (the built-in actions that implement them). It also contains other definitions, including the standard local and global dictionaries listed in Section 3.7.5, "Standard and User-Defined Dictionaries," as well as various named constants such as *true* and *false*.
- **globaldict** *(LanguageLevel 2)* is a writeable dictionary in *global VM*. This is explained in Section 3.7.2, "Local and Global VM."
- **userdict** is a writeable dictionary in *local VM*. It is the default modifiable naming environment normally used by PostScript programs.

userdict is the topmost of the permanent dictionaries on the dictionary stack. The **def** operator puts definitions there unless the program has pushed some other dictionary on the dictionary stack. Applications can and should create their own dictionaries rather than put things in **userdict**.

A dictionary is a composite object. Copying a dictionary object does not copy the dictionary's contents. Instead, the contents are shared.

3.3.10 Operator Objects

An *operator* object represents one of the PostScript language's built-in actions. When the object is executed, its built-in action is invoked. Much of this book is devoted to describing the semantics of the various operators.

Operators have names. Most operators are associated with names in **systemdict**: the names are the keys and the operators are the associated values. When the interpreter executes one of these names, it looks up the name in the context of the dictionary stack. Unless the name has been defined in some dictionary higher on the dictionary stack, the interpreter finds its definition in **systemdict**, fetches the associated value (the operator object itself), and executes it.

All standard operators are defined in **systemdict**. However, an application that tests whether an operator is defined should not use the **known** operator to determine whether the operator is in **systemdict**; it should instead use the **where** operator to check all dictionaries on the dictionary stack. Using **where** enables proper handling of operator emulations (see Appendix D).

Note: *There are some special internal PostScript operators whose names begin with an at sign (@). These operators are not officially part of the PostScript language and are not defined in* ***systemdict****. They may appear as an "offending command" in error messages.*

There is nothing special about an operator name, such as **add**, that distinguishes it as an operator. Rather, the name **add** is associated in **systemdict** with the operator for performing addition, and execution of the operator causes the addition to occur. Thus the name **add** is not a "reserved word," as it might be in other programming languages. Its meaning can be changed by a PostScript program.

Throughout this book, the notation **add** means "the operator object associated with the name **add** in **systemdict**" or, occasionally, in some other dictionary.

3.3.11 File Objects

A *file* is a readable or writeable stream of characters transferred between the PostScript interpreter and its environment. The characters in a file may be stored permanently—in a disk file, for instance—or may be generated dynamically and transferred via a communication channel.

A *file object* represents a file. There are operators to open a file and create a file object for it. Other operators access an open file to read, write, and process characters in various ways—as strings, as PostScript tokens, as binary data represented in hexadecimal, and so on.

Standard input and output files are always available to a PostScript program. The standard input file is the usual source of programs to be interpreted; the standard output file is the usual destination of such things as error and status messages.

Although a file object does not have components visible at the PostScript language level, it is composite in the sense that all copies of a file object share the same underlying file as their value. If a file operator has a side effect on the underlying file, such as closing it or changing the current position in the stream, all file objects sharing the file are affected.

The properties of files and the operations on them are described in more detail in Section 3.8, "File Input and Output."

3.3.12 Mark Objects

A *mark* is a special object used to denote a position on the operand stack. This use is described in the presentation of stack and array operators in Section 3.6, "Overview of Basic Operators." There is only one value of type mark, created by invoking the operator **mark**, **[**, or **<<**. Mark objects are not legal operands for most operators. They *are* legal operands for **]**, **>>**, **counttomark**, **cleartomark**, and a few generic operators such as **pop** and **type**.

3.3.13 Null Objects

The PostScript interpreter uses *null* objects to fill empty or uninitialized positions in composite objects when they are created. There is only one value of type null; the name **null** is associated with a null object in **systemdict**. Null objects are not legal operands for most operators.

3.3.14 Save Objects

Save objects represent snapshots of the state of the PostScript interpreter's memory. They are created and manipulated by the **save** and **restore** operators, introduced in Section 3.7.3, "Save and Restore."

3.3.15 Other Object Types

FontID objects are special objects used in the construction of fonts; see Section 5.2, "Font Dictionaries."

A *gstate* object *(LanguageLevel 2)* represents an entire *graphics state*; see Section 4.2, "Graphics State."

3.4 Stacks

The PostScript interpreter manages five stacks representing the execution state of a PostScript program. Three of them—the operand, dictionary, and execution stacks—are described here; the other two—the graphics state stack and clipping path stack—are presented in Chapter 4. Stacks are "last in, first out" (LIFO) data structures. In this book, "the stack" with no qualifier always means the operand stack.

- The *operand stack* holds arbitrary PostScript objects that are the operands and results of PostScript operators being executed. The interpreter pushes objects on the operand stack when it encounters them as literal data in a program being executed. When an operator requires one or more operands, it obtains them by popping them off the top of the operand stack. When an operator returns one or more results, it does so by pushing them on the operand stack.
- The *dictionary stack* holds only dictionary objects. The current set of dictionaries on the dictionary stack defines the environment for all implicit name searches, such as those that occur when the interpreter encounters an executable name. The role of the dictionary stack is introduced in Section 3.3, "Data Types and Objects," and is further explained in Section 3.5, "Execution."
- The *execution stack* holds executable objects (mainly procedures and files) that are in intermediate stages of execution. At any point in the execution of a PostScript program, this stack represents the program's call stack. Whenever the interpreter suspends execution of an object to execute some other object, it pushes the *new* object on the execution stack. When the interpreter finishes executing an object, it pops that object off the execution stack and resumes executing the suspended object beneath it.

The three stacks are independent and there are different ways to access each of them:

- The operand stack is directly under the control of the PostScript program being executed. Objects may be pushed and popped arbitrarily by various operators.
- The dictionary stack is also under PostScript program control, but it can hold only dictionaries. The bottom three dictionaries on the stack—**systemdict**, **globaldict**, and **userdict**—(or the bottom two, in LanguageLevel 1) cannot be popped off. The only operators that can alter the dictionary stack are **begin**, **end**, and **cleardictstack**.
- The execution stack is under the control of the PostScript interpreter. It can be read but not directly modified by a PostScript program.

When an object is pushed on a stack, the *object* is copied onto the stack from wherever it was obtained; however, in the case of a composite object (such as an array, a string, or a dictionary), the object's *value* is not copied onto the stack, but rather is shared with the original object. Similarly, when a composite object is popped off a stack and put somewhere, only the object itself is moved, not its value. See Section 3.3, "Data Types and Objects," for more details.

The maximum capacity of stacks may be limited; see Appendices B and C.

3.5 Execution

Execution semantics are different for each of the various object types. Also, execution can be either *immediate*, occurring as soon as the object is created by the scanner, or *deferred* to some later time.

3.5.1 Immediate Execution

Some example PostScript program fragments will help clarify the concept of execution. Example 3.1 illustrates the immediate execution of a few operators and operands to perform some simple arithmetic.

Example 3.1

```
40 60 add 2 div
```

The interpreter first encounters the literal integer object 40 and pushes it on the operand stack. Then it pushes the integer object 60 on the operand stack.

Next, it encounters the executable name object **add**, which it looks up in the environment of the current dictionary stack. Unless **add** has been redefined elsewhere, the interpreter finds it associated with an operator object, which it executes. This invokes a built-in function that pops the two integer objects off the operand stack, adds them together, and pushes the result (a new integer object whose value is 100) back on the operand stack.

The rest of the program fragment is executed similarly. The interpreter pushes the integer 2 on the operand stack and then executes the name **div**. The **div** operator pops two operands off the stack (the integers whose values are 2 and 100), divides the second-to-top one by the top one (100 divided by 2, in this case), and pushes the real result 50.0 on the stack.

The source of the objects being executed by the PostScript interpreter does not matter. They may have been contained within an array or scanned in from a character stream. Executing a sequence of objects produces the same result regardless of where the objects come from.

3.5.2 Operand Order

In Example 3.1, 40 is the first and 60 is the second operand of the **add** operator. That is, objects are referred to according to *the order in which they are pushed on the operand stack.* This is the reverse of the order in which they are popped off by the **add** operator. Similarly, the result pushed by the **add** operator is the first operand of the **div** operator, and 2 is its second operand.

The same terminology applies to the results of an operator. If an operator pushes more than one object on the operand stack, the first object pushed is the first result. This order corresponds to the usual left-to-right order of appearance of operands in a PostScript program.

3.5.3 Deferred Execution

The first line of Example 3.2 defines a procedure named average that computes the average of two numbers. The second line applies that procedure to the integers 40 and 60, producing the same result as Example 3.1.

Example 3.2

```
/average {add 2 div} def
40 60 average
```

The interpreter first encounters the literal name average. Recall from Section 3.2, "Syntax," that / introduces a literal name. The interpreter pushes this object on the operand stack, as it would any object having the literal attribute.

Next, the interpreter encounters the executable array {add 2 div}. Recall that { and } enclose a *procedure* (an executable array or executable packed array object) that is produced by the scanner. This procedure contains three elements: the executable name **add**, the literal integer 2, and the executable name **div**. The interpreter has not encountered these elements yet.

Here is what the interpreter does:

1. Upon encountering this procedure object, the interpreter pushes it on the operand stack, even though the object has the executable attribute. This is explained shortly.
2. The interpreter then encounters the executable name **def**. Looking up this name in the current dictionary stack, it finds **def** to be associated in **systemdict** with an operator object, which it invokes.
3. The **def** operator pops two objects off the operand stack (the procedure {add 2 div} and the name average). It enters this pair into the current dictionary (most likely **userdict**), creating a new association having the name average as its key and the procedure {add 2 div} as its value.
4. The interpreter pushes the integer objects 40 and 60 on the operand stack, then encounters the executable name average.
5. It looks up average in the current dictionary stack, finds it to be associated with the procedure {add 2 div}, and *executes* that procedure. In this case, execution of the array object consists of executing the elements of the array—the objects **add**, 2, and **div**—in sequence. This has the same effect as executing those objects directly. It produces the same result: the real object 50.0.

Why did the interpreter treat the procedure as data in the first line of the example but execute it in the second, despite the procedure having the executable attribute in both cases? There is a special rule that determines this behavior: An executable array or packed array encountered *directly* by the interpreter is treated as data

(pushed on the operand stack), but an executable array or packed array encountered *indirectly*—as a result of executing some other object, such as a name or an operator—is invoked as a procedure.

This rule reflects how procedures are ordinarily used. Procedures appearing directly (either as part of a program being read from a file or as part of some larger procedure in memory) are usually part of a definition or of a construct, such as a conditional, that operates on the procedure explicitly. But procedures obtained indirectly—for example, as a result of looking up a name—are usually intended to be executed. A PostScript program can override these semantics when necessary.

3.5.4 Control Constructs

In the PostScript language, control constructs such as conditionals and iterations are specified by means of operators that take procedures as operands. Example 3.3 computes the maximum of the values associated with the names a and b, as in the steps that follow.

Example 3.3

```
a b gt {a} {b} ifelse
```

1. The interpreter encounters the executable names a and b in turn and looks them up. Assume both names are associated with numbers. Executing the numbers causes them to be pushed on the operand stack.
2. The **gt** (greater than) operator removes two operands from the stack and compares them. If the first operand is greater than the second, it pushes the boolean value *true*. Otherwise, it pushes *false*.
3. The interpreter now encounters the procedure objects {a} and {b}, which it pushes on the operand stack.
4. The **ifelse** operator takes three operands: a boolean object and two procedures. If the boolean object's value is *true*, **ifelse** causes the first procedure to be executed; otherwise, it causes the second procedure to be executed. All three operands are removed from the operand stack before the selected procedure is executed.

In this example, each procedure consists of a single element that is an executable name (either a or b). The interpreter looks up this name and, since it is associated

with a number, pushes that number on the operand stack. So the result of executing the entire program fragment is to push on the operand stack the greater of the values associated with a and b.

3.5.5 Execution of Specific Types

An object with the literal attribute is *always* treated as data—pushed on the operand stack by the interpreter—regardless of its type. Even operator objects are treated this way if they have the literal attribute.

For many objects, executing them has the same effect as treating them as data. This is true of integer, real, boolean, dictionary, mark, save, gstate, and fontID objects. So the distinction between literal and executable objects of these types is meaningless. The following descriptions apply *only* to objects having the executable attribute.

- An *executable array* or *executable packed array* (procedure) object is pushed on the operand stack if it is encountered directly by the interpreter. If it is invoked *indirectly* as a result of executing some other object (a name or an operator), it is *called* instead. The interpreter calls a procedure by pushing it on the execution stack and then executing the array elements in turn. When the interpreter reaches the end of the procedure, it pops the procedure object off the execution stack. (Actually, it pops the procedure object when there is one element remaining and then pushes that element; this permits unlimited depth of "tail recursion" without overflowing the execution stack.)
- An *executable string* object is pushed on the execution stack. The interpreter then uses the string as a source of characters to be converted to tokens and interpreted according to the PostScript syntax rules. This continues until the interpreter reaches the end of the string. Then it pops the string object from the execution stack.
- An *executable file* object is treated much the same as a string: The interpreter pushes it on the execution stack. It reads the characters of the file and interprets them as PostScript tokens until it encounters end-of-file. Then it closes the file and pops the file object from the execution stack. See Section 3.8, "File Input and Output."
- An *executable name* object is looked up in the environment of the current dictionary stack and its associated value is executed. The interpreter looks first in the top dictionary on the dictionary stack and then in other dictionaries suc-

cessively lower on the stack. If it finds the name as a key in some dictionary, it executes the associated value. To do that, it examines the value's type and executable attribute and performs the appropriate action described in this section. Note that if the value is a procedure, the interpreter executes it. If the interpreter fails to find the name in any dictionary on the dictionary stack, an **undefined** error occurs.

- An *executable operator* object causes the interpreter to perform one of the built-in operations described in this book.
- An *executable null* object causes the interpreter to perform no action. In particular, it does *not* push the object on the operand stack.

3.6 Overview of Basic Operators

This is an overview of the general-purpose PostScript operators, excluding all operators that deal with graphics and fonts, which are described in later chapters. The information here is insufficient for actual programming; it is intended only to acquaint you with the available facilities. For complete information about any particular operator, you should refer to the operator's detailed description in Chapter 8.

3.6.1 Stack Operators

The operand stack is the PostScript interpreter's mechanism for passing arguments to operators and for gathering results from operators. It is introduced in Section 3.4, "Stacks."

There are various operators that rearrange or manipulate the objects on the operand stack. Such rearrangement is often required when the results of some operators are to be used as arguments to other operators that require their operands in a different order. These operators manipulate only the objects themselves; they do not copy the values of composite objects.

- **dup** duplicates an object.
- **exch** exchanges the top two elements of the stack.
- **pop** removes the top element from the stack.
- **copy** duplicates portions of the operand stack.

- **roll** treats a portion of the stack as a circular queue.
- **index** accesses the stack as if it were an indexable array.
- **mark** marks a position on the stack.
- **clear** clears the stack.
- **count** counts the number of elements on the stack.
- **counttomark** counts the elements above the highest mark. This is used primarily for array construction (described later), but has other applications as well.
- **cleartomark** removes all elements above the highest mark and then removes the mark itself.

3.6.2 Arithmetic and Mathematical Operators

The PostScript language includes a conventional complement of arithmetic and mathematical operators. In general, these operators accept either integer or real number objects as operands. They produce either integers or real numbers as results, depending on the types of the operands and the magnitude of the results. If the result of an operation is mathematically meaningless or cannot be represented as a real number, an **undefinedresult** error occurs.

- **add**, **sub**, **mul**, **div**, **idiv**, and **mod** are arithmetic operators that take two arguments.
- **abs**, **neg**, **ceiling**, **floor**, **round**, and **truncate** are arithmetic operators that take one argument.
- **sqrt**, **exp**, **ln**, **log**, **sin**, **cos**, and **atan** are mathematical and trigonometric functions.
- **rand**, **srand**, and **rrand** access a pseudo-random number generator.

3.6.3 Array, Packed Array, Dictionary, and String Operators

A number of operators are *polymorphic*: they may be applied to operands of several different types and their precise functions depend on the types of the operands. Except where indicated otherwise, the operators listed below apply to any of the following types of composite objects: arrays, packed arrays, dictionaries, and strings.

- **get** takes a composite object and an index (or a key, in the case of a dictionary) and returns a single element of the object.
- **put** stores a single element in an array, dictionary, or string. This operator does not apply to packed array objects, because they always have read-only (or even more restrictive) access.
- **copy** copies the *value* of a composite object to another composite object of the same type, replacing the second object's former value. This is different from merely copying the object. See Section 3.3.1, "Simple and Composite Objects" for a discussion of copying objects.
- **length** returns the number of elements in a composite object.
- **forall** accesses all of the elements of a composite object in sequence, calling a procedure for each one.
- **getinterval** creates a new object that shares a subinterval of an array, a packed array, or a string. This operator does not apply to dictionary objects.
- **putinterval** overwrites a subinterval of one array or string with the contents of another. This operator does not apply to dictionary or packed array objects, although it can overwrite a subinterval of an array with the contents of a packed array.

In addition to the polymorphic operators, there are operators that apply to only one of the array, packed array, dictionary, and string types. For each type, there is an operator (**array**, **packedarray**, **dict**, **string**) that creates a new object of that type and a specified length. These four operators explicitly create new composite object values, consuming virtual memory (VM) resources (see Section 3.7.1, "Virtual Memory"). Most other operators read and write the values of composite objects but do not create new ones. Operators that return composite results usually require an operand that is the composite object into which the result values are to be stored. The operators are organized this way to give programmers maximum control over consumption of VM.

Array, packed array, and string objects have a fixed length that is specified when the object is created. In LanguageLevel 1, dictionary objects also have this property. In LanguageLevels 2 and 3, a dictionary's capacity can grow beyond its initial allocation.

The following operators apply only to arrays and (sometimes) packed arrays:

- **aload** and **astore** transfer all the elements of an array to or from the operand stack in a single operation. **aload** may also be applied to a packed array.
- The array construction operators **[** and **]** combine to produce a new array object whose elements are the objects appearing between the brackets. The **[** operator, which is a synonym for **mark**, pushes a mark object on the operand stack. Execution of the program fragment between the **[** and the **]** causes zero or more objects to be pushed on the operand stack. Finally, the **]** operator counts the number of objects above the mark on the stack, creates an array of that length, stores the elements from the stack in the array, removes the mark from the stack, and pushes the array on the stack.
- **setpacking** and **currentpacking** *(both LanguageLevel 2)* control a mode setting that determines the type of procedure objects the scanner generates when it encounters a sequence of tokens enclosed in { and }. If the array packing mode is *true*, the scanner produces packed arrays; if the mode is *false*, it produces ordinary arrays. The default value is *false*.
- Packed array objects always have read-only (or even more restricted) access, so the **put**, **putinterval**, and **astore** operations are not allowed on them. Accessing arbitrary elements of a packed array object can be quite slow; however, accessing the elements sequentially, as the PostScript interpreter and the **forall** operator do, is efficient.

The following operators apply only to dictionaries:

- **begin** and **end** push new dictionaries on the dictionary stack and pop them off.
- **def** and **store** associate keys with values in dictionaries on the dictionary stack; **load** and **where** search for keys there.
- **countdictstack**, **cleardictstack**, and **dictstack** operate on the dictionary stack.
- **known** queries whether a key is present in a specific dictionary.
- **maxlength** obtains a dictionary's maximum capacity.
- **undef** *(LanguageLevel 2)* removes an individual key from a dictionary.
- **<<** and **>>** *(LanguageLevel 2)* construct a dictionary consisting of the bracketed objects interpreted as key-value pairs.

The following operators apply only to strings:

- **search** and **anchorsearch** perform textual string searching and matching.
- **token** scans the characters of a string according to the PostScript language syntax rules, without executing the resulting objects.

There are many additional operators that use array, dictionary, or string operands for special purposes—for instance, as transformation matrices, font dictionaries, or text.

3.6.4 Relational, Boolean, and Bitwise Operators

The relational operators compare two operands and produce a boolean result indicating whether the relation holds. Any two objects may be compared for equality (**eq** and **ne**—equal and not equal); numbers and strings may be compared by the inequality operators (**gt**, **ge**, **lt**, and **le**—greater than, greater than or equal to, less than, and less than or equal to).

The boolean and bitwise operators (**and**, **or**, **xor**, **true**, **false**, and **not**) compute logical combinations of boolean operands or bitwise combinations of integer operands. The bitwise shift operator **bitshift** applies only to integers.

3.6.5 Control Operators

The control operators modify the interpreter's usual sequential execution of objects. Most of them take a procedure operand that they execute conditionally or repeatedly.

- **if** and **ifelse** execute a procedure conditionally depending on the value of a boolean operand. (**ifelse** is introduced in Section 3.5, "Execution.")
- **exec** executes an arbitrary object unconditionally.
- **for**, **repeat**, **loop**, and **forall** execute a procedure repeatedly. Several specialized graphics and font operators, such as **pathforall** and **kshow**, behave similarly.
- **exit** transfers control out of the scope of any of these looping operators.
- **countexecstack** and **execstack** are used to read the execution stack.

A PostScript program may terminate prematurely by executing the **stop** operator. This occurs most commonly as a result of an error; the default error handlers (in **errordict**) all execute **stop**.

The **stopped** operator establishes an execution environment that encapsulates the effect of a **stop**. That is, **stopped** executes a procedure given as an operand, just the same as **exec**. If the interpreter executes **stop** during that procedure, it terminates the procedure and resumes execution at the object immediately after the **stopped** operator.

3.6.6 Type, Attribute, and Conversion Operators

These operators deal with the details of PostScript types, attributes, and values, introduced in Section 3.3, "Data Types and Objects."

- **type** returns the type of any operand as a name object (integertype, realtype, and so on).
- **xcheck**, **rcheck**, and **wcheck** query the literal/executable and access attributes of an object.
- **cvlit** and **cvx** change the literal/executable attribute of an object.
- **readonly**, **executeonly**, and **noaccess** reduce an object's access attribute. Access can only be reduced, never increased.
- **cvi** and **cvr** convert between integer and real types, and interpret a numeric string as an integer or real number.
- **cvn** converts a string to a name object defined by the characters of the string.
- **cvs** and **cvrs** convert objects of several types to a printable string representation.

3.7 Memory Management

A PostScript program executes in an environment with these major components: stacks, virtual memory, standard input and output files, and the graphics state.

- The *operand stack* is working storage for objects that are the operands and results of operators. The *dictionary stack* contains dictionary objects that define

the current name space. The *execution stack* contains objects that are in partial stages of execution by the PostScript interpreter. See Section 3.4, "Stacks."

- *Virtual memory* (VM) is a storage pool for the values of all composite objects. The adjective "virtual" emphasizes the behavior of this memory visible at the PostScript language level, not its implementation in computer storage.
- The *standard input file* is the normal source of program text to be executed by the PostScript interpreter. The *standard output file* is the normal destination of output from the **print** operator and of error messages. Other files can exist as well. See Section 3.8, "File Input and Output."
- The *graphics state* is a collection of parameters that control the production of text and graphics on a raster output device. See Section 4.2, "Graphics State."

This section describes the behavior of VM and its interactions with other components of the PostScript execution environment. It describes facilities for controlling the environment as a whole. The PostScript interpreter can execute a sequence of self-contained PostScript programs as independent "jobs"; similarly, each job can have internal structure whose components are independent of each other.

Some PostScript interpreters can support *multiple execution contexts*—the execution of multiple independent PostScript programs at the same time. Each context has an environment consisting of stacks, VM, graphics state, and certain other data. Under suitable conditions, objects in VM can be shared among contexts; there are means to regulate concurrent access to the shared objects.

This edition of this book does not document the multiple contexts extension, although it does indicate which components of a PostScript program's environment are maintained on a per-context basis. Further information about multiple contexts can be found in the second edition of this book and in the *Display PostScript System* manuals.

3.7.1 Virtual Memory

As described in Section 3.3, "Data Types and Objects," objects may be either simple or composite. A simple object's value is contained in the object itself. A composite object's value is stored separately; the object contains a reference to it. Virtual memory (VM) is the storage in which the values of composite objects reside.

For example, the program fragment

```
234 (string1)
```

pushes two objects, an integer and a string, on the operand stack. The integer, which is a simple object, contains the value 234 as part of the object itself. The string, which is a composite object, contains a reference to the value string1, which is a text string that resides in VM. The elements of the text string are characters (actually, integers in the range 0 to 255) that can be individually selected or replaced.

Here is another example:

```
{234 (string1)}
```

This pushes a single object, a two-element executable array, on the operand stack. The array is a composite object whose value resides in VM. The value in turn consists of two objects, an integer and a string. Those objects are elements of the array; they can be individually selected or replaced.

Several composite objects can share the same value. For example, in

```
{234 (string1)} dup
```

the **dup** operator pushes a second copy of the array object on the operand stack. The two objects share the same value—that is, the same storage in VM. So replacing an element of one array will affect the other. Other types of composite objects, including strings and dictionaries, behave similarly.

Creating a new composite object consumes VM storage for its value. This occurs in two principal ways:

- The scanner allocates storage for each composite literal object that it encounters. Composite literals are delimited by (…), < … >, <~ … ~>, and { … }. The first three produce strings; the fourth produces an executable array or packed array. There also are binary encodings for composite objects.
- Some operators explicitly create new composite objects and allocate storage for them. The **array**, **packedarray**, **dict**, **string**, and **gstate** operators create new array, packed array, dictionary, string, and gstate objects, respectively. Also, the bracketing constructs […] and << … >> create new array and dictionary ob-

jects, respectively. The brackets are just special names for operators; the closing bracket operators allocate the storage.

For the most part, consumption and management of VM storage is under the control of the PostScript program. Aside from the operators mentioned above and a few others that are clearly documented, most operators do not create new composite objects or allocate storage in VM. Some operators place their results in existing objects supplied by the caller. For example, the **cvs** (convert to string) operator overwrites the value of a supplied string operand and returns a string object that shares a substring of the supplied string's storage.

3.7.2 Local and Global VM

There are two divisions of VM containing the values of composite objects: *local* and *global*. Only composite objects occupy VM. An "object in VM" means a "composite object whose *value* occupies VM"; the location of the object (for example, on a stack or stored as an element of some other object) is immaterial.

Global VM exists only in LanguageLevel 2 and LanguageLevel 3 interpreters. In LanguageLevel 1 interpreters, all of VM is local.

Local VM is a storage pool that obeys a stacklike discipline. Allocations in local VM and modifications to existing objects in local VM are subject to the **save** and **restore** operators. These operators bracket a section of a PostScript program whose local VM activity is to be encapsulated. **restore** deallocates new objects and undoes modifications to existing objects that were made since the matching **save** operation. **save** and **restore** are described in Section 3.7.3, "Save and Restore."

Global VM is a storage pool for objects that do not obey a fixed discipline. Objects in global VM can come into existence and disappear in an arbitrary order during execution of a program. Modifications to existing objects in global VM are not affected by occurrences of **save** and **restore** within the program. However, an entire job's VM activity can be encapsulated, enabling separate jobs to be executed independently. This is described in Section 3.7.7, "Job Execution Environment."

In a hierarchically structured program such as a page description, local VM is used to hold information whose lifetime conforms to the structure; that is, it persists to the end of a structural division, such as a single page. Global VM may be

used to hold information whose lifetime is independent of the structure, such as definitions of fonts and other resources that are loaded dynamically during the execution of a program.

Control over allocation of objects in local versus global VM is provided by the **setglobal** operator *(LanguageLevel 2)*. This operator establishes a *VM allocation mode*, a boolean value that determines where subsequent allocations are to occur (*false* means local, *true* means global). It affects objects created implicitly by the scanner and objects created explicitly by operators. The default VM allocation mode is local; a program can switch to global allocation mode when it needs to.

The following example illustrates the creation of objects in local and global VM:

```
/lstr (string1) def
/ldict 10 dict def
true setglobal
/gstr (string2) def
/gdict 5 dict def
false setglobal
```

In the first line, when the scanner encounters (string1), it allocates the string object in local VM. In the second line, the **dict** operator allocates a new dictionary in local VM. The third line switches to global VM allocation mode. The fourth and fifth lines allocate a string object and a dictionary object in global VM. The sixth line switches back to local VM allocation mode. The program associates the four newly created objects with the names lstr, ldict, gstr, and gdict in the current dictionary (presumably **userdict**).

An object in global VM is not allowed to contain a reference to an object in local VM. An attempt to store a local object as an element of a global object will result in an **invalidaccess** error. The reason for this restriction is that subsequent execution of the **restore** operator might deallocate the local object, leaving the global object with a "dangling" reference to a nonexistent object.

This restriction applies only to storing a *composite* object in local VM as an element of a *composite* object in global VM. All other combinations are allowed. The following example illustrates this, using the objects that were created in the preceding example.

```
ldict /a lstr put       % Allowed—a local object into a local dict
gdict /b gstr put       % Allowed—a global object into a global dict
ldict /c gstr put       % Allowed—a global object into a local dict
gdict /d lstr put       % Not allowed (invalidaccess error)—a local object into a global dict
gdict /e 7 put          % Allowed—a simple object into any dict
```

There are no restrictions on storing simple objects, such as integers and names, as elements of either local or global composite objects. The **gcheck** operator inquires whether an object can be stored as an element of a global composite object. It returns *true* for a simple object or for a composite object in global VM, or *false* for a composite object in local VM.

3.7.3 Save and Restore

The **save** operator takes a snapshot of the state of local VM and returns a *save object* that represents the snapshot. The **restore** operator causes local VM to revert to a snapshot generated by a preceding **save** operation. Specifically, **restore** does the following:

- Discards all objects in local VM that were created since the corresponding **save**, and reclaims the memory they occupied
- Resets the values of all composite objects in local VM, except strings, to their state at the time of the **save**
- Performs an implicit **grestoreall** operation, which resets the graphics state to its value at the time of the **save** (see Section 4.2, "Graphics State")
- Closes files that were opened since the corresponding **save**, so long as those files were opened while local VM allocation mode was in effect (see Section 3.8, "File Input and Output")

The effects of **restore** are limited to the ones described above. In particular, **restore** does not:

- Affect the contents of the operand, dictionary, and execution stacks. If a stack contains a reference to a composite object in local VM that would be discarded by the **restore** operation, the **restore** is not allowed; an **invalidrestore** error occurs.
- Affect any objects that reside in global VM, except as described in Section 3.7.7, "Job Execution Environment."

- Undo side effects outside VM, such as writing data to files or rendering graphics on the raster output device. (However, the implicit **grestoreall** may deactivate the current device, thereby erasing the current page; see Section 6.2.6, "Device Initialization and Page Setup," for details.)

The **save** and **restore** operators can be nested to a limited depth (see Appendix B for implementation limits). A PostScript program can use **save** and **restore** to encapsulate the execution of an embedded program that also uses **save** and **restore**.

save and **restore** are intended for use in structured programs such as page descriptions. The conventions for structuring programs are introduced in Section 2.4.2, "Program Structure," and described in detail in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*. In such programs, **save** and **restore** serve the following functions:

- A document consists of a prolog and a script. The prolog contains definitions that are used throughout the document. The script consists of a sequence of independent pages. Each page has a **save** at the beginning and a **restore** at the end, immediately before the **showpage** operator. Each page begins execution with the initial conditions established in local VM by the prolog. There are no unwanted legacies from previous pages.
- A page sometimes contains additional substructure, such as embedded illustrations, whose execution needs to be encapsulated. The encapsulated program can make wholesale changes to the contents of local VM to serve its own purposes. By bracketing the program with **save** and **restore**, the enclosing program can isolate the effects of the embedded program.
- As a PostScript program executes, new composite objects accumulate in local VM. These include objects created by the scanner, such as literal string tokens, and objects allocated explicitly by operators. The **restore** operator reclaims all local VM storage allocated since the corresponding **save**; executing **save** and **restore** periodically ensures that unreclaimed objects will not exhaust available VM resources. In LanguageLevel 1, **save** and **restore** are the *only* way to reclaim VM storage. Even in higher LanguageLevels, explicit reclamation by **save** and **restore** is much more efficient than automatic reclamation (described in Section 3.7.4, "Garbage Collection").
- The PostScript interpreter uses **save** and **restore** to encapsulate the execution of individual jobs, as described in Section 3.7.7, "Job Execution Environment."

3.7.4 Garbage Collection

In addition to the **save** and **restore** operators for explicit VM reclamation, LanguageLevels 2 and 3 include a facility for automatic reclamation, popularly known as a *garbage collector*. The garbage collector reclaims the memory occupied by composite objects that are no longer accessible to the PostScript program.

For example, after the program

```
/a (string1) def
/a (string2) def
(string3) show
```

is executed, the string object string1 is no longer accessible, since the dictionary entry that referred to it has been replaced by a different object, string2. Similarly, the string object string3 is no longer accessible, since the **show** operator consumes its operand but does not store it anywhere. These inaccessible strings are candidates for garbage collection.

Garbage collection normally takes place without explicit action by the PostScript program. It has no effects that are visible to the program. However, the presence of a garbage collector strongly influences the style of programming that is permissible. If no garbage collector is present, a program that consumes VM endlessly and never executes **save** and **restore** will eventually exhaust available memory and cause a **VMerror**.

There is a cost associated with creating and destroying composite objects in VM. The most common case is that literal objects—particularly strings, user paths, and binary object sequences—are immediately consumed by operators such as **show** and **ufill**, and never used again. The garbage collector is engineered to deal with this case inexpensively, so application programs should not hesitate to take advantage of it. However, the cost of garbage collection is greater for objects that have longer lifetimes or are allocated explicitly. Programs that frequently require temporary objects are encouraged to create them once and reuse them instead of creating new ones—for example, allocate a string object *before* an image data acquisition procedure, rather than within it (see Section 4.10.7, "Using Images").

Even with garbage collection, the **save** and **restore** operators still have their standard behavior. That is, **restore** resets all accessible objects in local VM to their state at the time of the matching **save**. It reclaims all composite objects created in

local VM since the **save** operation, and does so very cheaply. On the other hand, garbage collection is the only way to reclaim storage in global VM, since **save** and **restore** normally do not affect global VM.

With garbage collection comes the ability to explicitly discard composite objects that are no longer needed. This can be done in an order unrelated to the time of creation of those objects, as opposed to the stacklike order imposed by **save** and **restore**. This technique is particularly desirable for very large objects, such as font definitions.

If the only reference to a particular composite object is an element of some array or dictionary, replacing that element with something else (using **put**, for instance) renders the object inaccessible. Alternatively, the **undef** operator removes a dictionary entry entirely; that is, it removes both the key and the value of a key-value pair, as opposed to replacing the value with some other value. In either case, the removed object becomes a candidate for garbage collection.

Regardless of the means used to remove a reference to a composite object, if the object containing the reference is in local VM, the action can be undone by a subsequent **restore**. This is true even for **undef**. Consider the following example:

```
/a (string1) def
save
    currentdict /a undef
restore
```

Execution of **undef** removes the key a and its value from the current dictionary, seemingly causing the object string1 to become inaccessible. However, assuming that the current dictionary is **userdict** (or some other dictionary in local VM), **restore** reinstates the deleted entry, since it existed at the time of the corresponding **save**. The value is still accessible and cannot be garbage-collected.

As a practical matter, this means that the technique of discarding objects explicitly (in expectation of their being garbage-collected) is useful mainly for objects in global VM, where **save** and **restore** have no effect, and for objects in local VM that were created at the current level of **save** nesting.

3.7.5 Standard and User-Defined Dictionaries

A job begins execution with three standard dictionaries on the dictionary stack (in order from bottom to top):

- **systemdict**, a global dictionary that is permanently read-only and contains mainly operators
- **globaldict** *(LanguageLevel 2)*, a global dictionary that is writeable
- **userdict**, a local dictionary that is writeable

There are other standard dictionaries that are the values of permanent named entries in **systemdict**. Some of these are in local VM and some in global VM, as shown in Tables 3.3 and 3.4.

A PostScript program can also create new dictionaries in either local or global VM, then push them on the dictionary stack or store them as entries in **userdict** or **globaldict**.

TABLE 3.3 Standard local dictionaries

DICTIONARY	DESCRIPTION
userdict	Standard writeable local dictionary. Initially, it is the top dictionary on the dictionary stack, making it the current dictionary.
errordict	Error dictionary. See Section 3.11, "Errors."
$error	Dictionary accessed by the built-in error-handling procedures to store stack snapshots and other information. See Section 3.11, "Errors."
statusdict	Dictionary for product-specific operators and other definitions. See Chapter 8.
FontDirectory	Dictionary for font definitions. It is normally read-only, but is updated by **definefont** and consulted by **findfont**. See Sections 3.9, "Named Resources," and 5.2, "Font Dictionaries."

TABLE 3.4 Standard global dictionaries

DICTIONARY	DESCRIPTION
systemdict	Read-only system dictionary containing all operators and other definitions that are standard parts of the PostScript language. It is the bottom dictionary on the dictionary stack.
globaldict	*(LanguageLevel 2)* Standard writeable global dictionary. It is on the dictionary stack between **systemdict** and **userdict**.
GlobalFontDirectory	*(LanguageLevel 2)* Dictionary for font definitions in global VM. It is normally read-only, but is updated by **definefont** and consulted by **findfont**. See Sections 3.9, "Named Resources," and 5.2, "Font Dictionaries."

The dictionaries **userdict** and **globaldict** are intended to be the principal repositories for application-defined dictionaries and other objects. When a PostScript program creates a dictionary in local VM, it then typically associates that dictionary with a name in **userdict**. Similarly, when the program creates a dictionary in global VM, it typically associates the dictionary with a name in **globaldict**. Note that the latter step requires explicit action on the part of the program. Entering global VM allocation does *not* alter the dictionary stack (say, to put **globaldict** on top).

***Note: systemdict**, a global dictionary, contains several entries whose values are local dictionaries, such as **userdict** and **$error**. This is an exception to the normal rule, described in Section 3.7.2, "Local and Global VM," that prohibits objects in global VM from referring to objects in local VM.*

The principal intended use of global VM is to hold font definitions and other resources that are loaded dynamically during execution of a PostScript program. The **findresource** operator loads resources into global VM automatically when appropriate. However, any program can take advantage of global VM when its properties are useful. The following guidelines are suggested:

- Objects that are created during the prolog can be in either local or global VM; in either case, they will exist throughout the job, since they are defined outside the **save** and **restore** that enclose individual pages of the script. A dictionary in local VM reverts to the initial state defined by the prolog at the end of each page. This is usually the desirable behavior. A dictionary in global VM accumu-

lates changes indefinitely and never reverts to an earlier state; this is useful when there is a need to communicate information from one page to another (strongly discouraged in a page description).

- When using a writeable dictionary in global VM, you must be careful about what objects you store in it. Attempting to store a local composite object in a global dictionary will cause an **invalidaccess** error. For this reason, it is advisable to segregate local and global data and to use global VM only for those objects that must persist through executions of **save** and **restore**.
- In general, the prologs for most existing PostScript programs do not work correctly if they are simply loaded into global VM. The same is true of some fonts, particularly Type 3 fonts. These programs must be altered to define global and local information separately. Typically, global VM should be used to hold procedure definitions and constant data; local VM should be used to hold temporary data needed during execution of the procedures.
- Creating gstate (graphics state) objects in global VM is particularly risky. This is because the graphics state almost always contains one or more local objects, which cannot be stored in a global gstate object (see the **currentgstate** operator in Chapter 8).

3.7.6 User Objects

Some applications require a convenient and efficient way to refer to PostScript objects previously constructed in VM. The conventional way to accomplish this is to store such objects as named entries in dictionaries and later refer to them by name. In a PostScript program written by a programmer, this approach is natural and straightforward. When the program is generated mechanically by another program, however, it is more convenient to number the objects with small integers and later refer to them by number. This technique simplifies the bookkeeping the application program must do.

LanguageLevel 2 provides built-in support for a single space of numbered objects, called *user objects*. There are three operators, **defineuserobject**, **undefineuserobject**, and **execuserobject**, that manipulate an array named **UserObjects**. These operators do not introduce any fundamental capability, but merely provide convenient and efficient notation for accessing the elements of a special array.

Example 3.4 illustrates the intended use of user objects.

Example 3.4

```
17 {ucache 132 402 316 554 setbbox … } cvlit defineuserobject
17 execuserobject ufill
```

The first line of the example constructs an interesting object that is to be used repeatedly (in this case, a user path; see Section 4.6, "User Paths") and associates the index 17 with this object.

The second line pushes the user object on the operand stack, from which **ufill** takes it. **execuserobject** *executes* the user object associated with index 17. However, because the object in this example is not executable, the result of the execution is to push the object on the operand stack.

defineuserobject manages the **UserObjects** array automatically; there is no reason for a PostScript program to refer to **UserObjects** explicitly. The array is allocated in local VM and defined in **userdict**. This means that the effect of **defineuserobject** is subject to **save** and **restore**. The values of user objects given to **defineuserobject** can be in either local or global VM.

3.7.7 Job Execution Environment

As indicated in Section 2.4, "Using the PostScript Language," the conventional model of a PostScript interpreter is a "print server"—a single-threaded process that consumes and executes a sequence of "print jobs," each of which is a complete, independent PostScript program. This model is also appropriate for certain other environments, such as a document previewer running on a host computer.

The notion of a print job is not formally a part of the PostScript language, because it involves not only the PostScript interpreter but also some description of the environment in which the interpreter operates. Still, it is useful to describe a general *job* (and *job server*) model that is accurate for most PostScript printers, though perhaps lacking in some details. Information about communication protocols, job control, system management, and so on, does not appear here, but rather in documentation for specific products.

A job begins execution in an initial environment that consists of the following:

- An empty operand stack
- A dictionary stack containing the standard dictionaries—**systemdict**, **globaldict** *(LanguageLevel 2)*, and **userdict**
- Execution and graphics state stacks reset to their standard initial state, with no vestiges of previous jobs
- The contents of VM (local and global)
- Miscellaneous interpreter parameters

During execution, the job may alter its environment. Ordinarily, when a job finishes, the environment reverts to its initial state to prepare for the next job. That is, the job is *encapsulated*. The server accomplishes this encapsulation by executing **save** and **restore** and by explicitly resetting stacks and parameters between jobs.

With suitable authorization, a job can make persistent alterations to objects in VM. That is, the job is not encapsulated. Instead, its alterations appear as part of the initial state of the next and all subsequent jobs. This is accomplished by means of the **startjob** and **exitserver** facilities, described below.

Server Operation

A job server is presented with a sequence of files via one or more communication channels. For each file, the server performs the following sequence of steps:

1. Establish standard input and output file objects for the channel from which the file is to be obtained. The means by which this is done is implementation-dependent.
2. Execute **save**. This is the outermost **save**, which unlike a normal **save** obtains a snapshot of the initial state of objects in both local and global VM.
3. Establish the default initial state for the interpreter: empty operand stack, local VM allocation mode, default user space for the raster output device, and so on.
4. Execute the standard input file until it reaches end-of-file or an error occurs. If an error occurs, report it and flush input to end-of-file.

5. Clear the operand stack and reset the dictionary stack to its initial state.
6. Execute **restore**, causing objects in VM (both local and global) to revert to the state saved in step 2.
7. Close the standard input and output files, transmitting an end-of-file indication over the communication channel.

Ordinarily, the server executes all of the above steps once for each file that it receives. Each file is treated as a separate job, and each job is encapsulated.

Altering Initial VM

A program can circumvent job encapsulation and alter the initial VM for subsequent jobs. To do so, it can use either **startjob** *(LanguageLevel 2)* or **exitserver** (available in all implementations that include a job server). This capability is controlled by a password. The system administrator can choose not to make the capability available to ordinary users. Applications and drivers must be prepared to deal with the possibility that altering the initial VM is not allowed.

***Note: startjob** and **exitserver** should be invoked only by a print manager, spooler, or system administration program. They should never be used by an application program composing a page description. Appendix G gives more guidelines for using **startjob** and **exitserver**.*

startjob is invoked as follows:

```
true password startjob
```

where *password* is a string or an integer (see Section C.3.1, "Passwords"). If the password is correct, **startjob** causes the server to execute steps 5, 6, 3, and 4 in the sequence above. In other words, it logically ends the current job, undoing all modifications it has made so far, and starts a new job. However, it does not precede the new job with a **save** operation, so its execution is not encapsulated. Furthermore, it does not disturb the standard input and output files; the interpreter resumes consuming the remainder of the same input file.

Having started an unencapsulated job, the PostScript program can alter VM in arbitrary ways. Such alterations are persistent. If the job simply runs to completion, ending step 5 in the sequence above, the server skips step 6 (since there is no

saved VM snapshot to restore), continues with step 7, and processes the next job normally starting at step 1.

Alternatively, a program can explicitly terminate its alterations to initial VM:

```
false password startjob
```

This operation has the effect of executing steps 2, 3, and 4, logically starting yet another job that is encapsulated in the normal way, but still continuing to read from the same file.

If **startjob** executes successfully, it always starts a new job in the sense described above. It resets the stacks to their initial state and then pushes the result *true* on the operand stack. But if **startjob** is unsuccessful, it has no effect other than to push *false* on the operand stack; the effect is as if the program text before and after the occurrence of **startjob** were a single combined job.

The example sequence

```
true password startjob pop
    … Application prolog here …
false password startjob pop
    … Application script here …
```

installs the application prolog in initial VM if it is allowed to do so. However, the script executes successfully regardless of whether the attempt to alter initial VM was successful. The program can determine the outcome by testing the result returned by **startjob**.

The above sequence is an example; there is no restriction on the sequence of encapsulated and unencapsulated jobs. If the password is correct and the boolean operand to **startjob** is *true*, the job that follows it is unencapsulated; if *false*, the job is encapsulated. But if the password is incorrect, **startjob** does not start a new job; the current job simply continues.

startjob also fails to start a new job if, at the time it is executed, the current **save** nesting is more than one level deep. In other words, **startjob** works only when the current **save** level is equal to the level at which the current job started. This permits a file that executes **startjob** to be encapsulated as part of another job simply by bracketing it with **save** and **restore**.

Note: *If an unencapsulated job uses* ***save*** *and* ***restore****, the* ***save*** *and* ***restore*** *operations affect global as well as local VM, since they are at the outermost* ***save*** *level. Also, if the job ends with one or more* ***save*** *operations pending, a* ***restore*** *to the outermost saved VM is performed automatically.*

exitserver

exitserver is an unofficial LanguageLevel 1 feature that is retained in higher LanguageLevels for compatibility. Although **exitserver** has never been a formal part of the PostScript language, it exists in nearly every Adobe PostScript product, and some applications have come to depend on it. The **startjob** feature, described above, is more flexible and is preferred for new applications in LanguageLevels 2 and 3.

The canonical method of invoking **exitserver** is

```
serverdict begin password exitserver
```

This has the same effect as

```
true password startjob not
        {/exitserver errordict /invalidaccess get exec}
    if
```

In other words, if successful, **exitserver** initiates an unencapsulated job that can alter initial VM; if unsuccessful, it generates an **invalidaccess** error. Like **startjob**, a successful **exitserver** operation resets the stacks to their initial state: it removes **serverdict** from the dictionary stack. The program that follows (terminated by end-of-file) is executed as an unencapsulated job.

In many implementations, successful execution of **exitserver** sends the message

```
%%[exitserver: permanent state may be changed]%%
```

to the standard output file. This message is not generated by **startjob**. It is suppressed if **binary** is *true* in the **$error** dictionary; see Section 3.11.2, "Error Handling."

Note: *Aside from* ***exitserver****, the other contents of* ***serverdict*** *are not specified as part of the language. In LanguageLevels 2 and 3, the effect of executing* ***exitserver*** *more than once in the same file is the same as that of executing the equivalent* ***startjob*** *se-*

quence multiple times. In LanguageLevel 1, the effect of executing the ***exitserver*** *operator multiple times is undefined and unpredictable.*

3.8 File Input and Output

A *file* is a finite sequence of characters bounded by an end-of-file indication. These characters may be stored permanently in some place (for instance, a disk file) or they may be generated on the fly and transmitted over some communication channel. Files are the means by which the PostScript interpreter receives executable programs and exchanges data with the external environment.

There are two kinds of file: input and output. An *input file* is a source from which a PostScript program can read a sequence of characters; an *output file* is a destination to which a PostScript program can write characters. Some files can be both read and written.

The contents of a file are treated as a sequence of 8-bit bytes. In some cases, those bytes can be interpreted as text characters, such as the ASCII text representing a PostScript program. In other cases, they can be interpreted as arbitrary binary data. In the descriptions of files and file operators, the terms *character* and *byte* are synonymous.

3.8.1 Basic File Operators

A PostScript *file object* represents a file. The file operators take a file object as an operand to read or write characters. Ignoring for the moment *how* a file object comes into existence, the file operators include the following:

- **read** reads the next character from an input file.
- **write** appends a character to an output file.
- **readstring**, **readline**, and **writestring** transfer the contents of strings to and from files.
- **readhexstring** and **writehexstring** read and write binary data represented in the file by hexadecimal notation.
- **token** scans characters from an input file according to the PostScript language syntax rules.

- **exec**, applied to an input file, causes the PostScript interpreter to execute a PostScript program from that file.

The operators that write to a file do not necessarily deliver the characters to their destination immediately. They may leave some characters in buffers for reasons of implementation or efficiency. The **flush** and **flushfile** operators deliver these buffered characters immediately. These operators are useful in certain situations, such as during two-way interactions with another computer or with a human user, when such data must be transmitted immediately.

Standard Input and Output Files

All PostScript interpreters provide a *standard input file* and a *standard output file*, which usually represent a real-time communication channel to and from another computer. The standard input and output files are always present; it is not necessary for a program to create or close them.

The PostScript interpreter reads and interprets the standard input file as PostScript program text. It sends error and status messages to the standard output file. Also, a PostScript program may execute the **print** operator to send arbitrary data to the standard output file. Note that **print** is a *file* operator; it has nothing to do with placing text on a page or causing pages to emerge from a printer.

It is seldom necessary for a PostScript program to deal explicitly with file objects for the standard files, because the PostScript interpreter reads the standard input file by default and the **print** operator references the standard output file implicitly. Additionally, the file currently being read by the PostScript interpreter is available via the **currentfile** operator; this file need not be the standard input file. However, when necessary, a program may apply the **file** operator to the identifying strings %stdin or %stdout to obtain file objects for the standard input and output files; see Section 3.8.3, "Special Files."

End-of-Line Conventions

The PostScript language scanner and the **readline** operator recognize all three external forms of end-of-line (EOL)—CR alone, LF alone, and the CR-LF pair—and treat them uniformly, translating them as described below. The PostScript interpreter does not perform any such translation when reading data by other means or when writing data by any means.

End-of-line sequences are recognized and treated specially in the following situations:

- Any of the three forms of EOL appearing in a literal string is converted to a single LF character in the resulting string object. These three examples produce identical string objects, each with an LF character following the second word in the string:

  (any text⟨CR⟩some more text)
  (any text⟨LF⟩some more text)
  (any text⟨CR⟩⟨LF⟩some more text)

- Any of the three forms of EOL appearing immediately after \ in a string is treated as a line continuation; both the \ and the EOL are discarded. These four examples produce identical string objects:

  (any text \⟨CR⟩some more text)
  (any text \⟨LF⟩some more text)
  (any text \⟨CR⟩⟨LF⟩some more text)
  (any text some more text)

- Any of the three forms of EOL appearing outside a string is treated as a single white-space character. Since the language treats multiple white-space characters as a single white-space character, the treatment of EOL is interesting only when a PostScript token is followed by data to be read explicitly by one of the file operators. The following three examples produce identical results: the operator reads the character x from the current input file and leaves its character code (the integer 120) on the stack.

  currentfile read⟨CR⟩x
  currentfile read⟨LF⟩x
  currentfile read⟨CR⟩⟨LF⟩x

- The **readline** operator treats any of the three forms of EOL as the termination condition.

- Data read by **read** and **readstring** does not undergo EOL translation: the PostScript interpreter reads whatever characters were received from the channel. The same is true of data written by **write** and **writestring**: whatever characters the interpreter provides are sent to the channel. However, in either case the channel itself may perform some EOL translation, as discussed below.

Communication Channel Behavior

Communications functions often usurp control characters. Control codes are device-dependent and not part of the PostScript language. For example, the serial communication protocol supported by many products uses the Control-D character as an end-of-file indication. In this case, Control-D is a communications function and not logically part of a PostScript program. This applies specifically to the serial channel; other channels, such as LocalTalk™ and Ethernet, have different conventions for end-of-file and other control functions. *In all cases, communication channel behavior is independent of the actions of the PostScript interpreter.*

There are two levels of PostScript EOL translation: one in the PostScript interpreter and one in the serial communication channel. The previous description applies *only* to the EOL conventions at the level of the PostScript interpreter. The purpose of the seemingly redundant communication-level EOL translation is to maintain compatibility with diverse host operating systems and communications environments.

As discussed in Section 3.2, "Syntax," the ASCII encoding of the language is designed for maximum portability. It avoids using control characters that might be preempted by operating systems or communication channels. However, there are situations in which transmission of arbitrary binary data is desirable. For example, sampled images are represented by large quantities of binary data. The PostScript language has an alternative binary encoding that is advantageous in certain situations. There are two main ways to deal with PostScript programs that contain binary information:

- Communicate with the interpreter via binary channels exclusively. Some channels, such as LocalTalk and Ethernet, are binary by nature. They do not preempt any character codes, but instead communicate control information separately from the data. Other channels, such as serial channels, may support a binary communication protocol that allows control characters to be quoted. This approach presupposes a well-controlled environment. PostScript programs produced in that environment may not be portable to other environments.
- Take advantage of filters for encoding binary data as ASCII text. Filters are a LanguageLevel 2 feature, described in Section 3.8.4, "Filters." Programs represented in this way do not include any control codes and are therefore portable to any LanguageLevel 2 or 3 interpreter in any environment.

3.8.2 Named Files

The PostScript language provides access to named files in secondary storage. The file access capabilities are part of the integration of the language with an underlying operating system; there are variations from one such integration to another. Not all the file system capabilities of the underlying operating system are necessarily made available at the PostScript language level.

The PostScript language provides a standard set of operators for accessing named files. These operators are supported in LanguageLevels 2 and 3, as well as in certain LanguageLevel 1 implementations that have access to file systems. The operators are **file**, **deletefile**, **renamefile**, **status**, **filenameforall**, **setfileposition**, and **fileposition**. Even in LanguageLevel 1 implementations that do not support named files, the **file** operator is supported, because the special file names %stdin, %stdout, and %stderr are always allowed (see Section 3.8.3, "Special Files"). Although the language defines a standard framework for dealing with files, the detailed semantics of the file system operators, particularly file naming conventions, are operating system–dependent.

Files are stored in one or more "secondary storage devices," hereafter referred to simply as *devices*. (These are not to be confused with the "current device," which is a raster output device identified in the graphics state.) The PostScript language defines a uniform convention for naming devices, but it says nothing about how files in a given device are named. Different devices have different properties, and not all devices support all operations.

A complete file name has the form %*device*%*file*, where *device* identifies the secondary storage device and *file* is the name of the file within the device. When a complete file name is presented to a file system operator, the *device* portion selects the device; the *file* portion is in turn presented to the implementation of that device, which is operating system–dependent and environment-dependent.

Note: *Typically, file names cannot contain null characters (ASCII code 0); if a file name is specified by a string object containing a null character, the null character will effectively terminate the file name.*

When a file name is presented without a %*device*% prefix, a search rule determines which device is selected. The available storage devices are consulted in order; the requested operation is attempted on each device until the operation succeeds. The number of available devices, their names, and the order in which

they are searched is environment-dependent. Not all devices necessarily participate in such searches; some devices can be accessed only by explicitly naming them.

In an interpreter that runs on top of an operating system, there may be a device that represents the complete file system provided by the operating system. If so, by convention that device's name is os; thus, complete file names are in the form %os%*file*, where *file* conforms to underlying file system conventions. This device always participates in searches, as described above; a program can access ordinary files without specifying the %os% prefix. There may be more than one device that behaves in this way; the names of such devices are product-dependent.

Note: *The os device may impose some restrictions on the set of files that can be accessed. Restrictions are necessary when the PostScript interpreter executes with a user identity different from that of the user running the application program.*

In an interpreter that controls a dedicated product, such as a typical printer product, there can be one or more devices that represent file systems on disks and cartridges. Files on these devices have names such as %disk0%*file*, %disk1%*file*, and %cartridge0%*file*. Again, these devices participate in searches when the device name is not specified.

Each of the operators **file**, **deletefile**, **renamefile**, **status**, and **filenameforall** takes a *filename* operand—a string object that identifies a file. The name of the file can be in one of three forms:

- %*device*%*file* identifies a named file on a specific device, as described above.
- *file* (first character not %) identifies a named file on an unspecified device, which is selected by an environment-specific search rule, as described above.
- %*device* or %*device*% identifies an unnamed file on the device. Certain devices, such as cartridges, support a single unnamed file as opposed to a collection of named files. Other devices represent communication channels rather than permanent storage media. There are also special files named %stdin, %stdout, %stderr, %statementedit, and %lineedit, described in Section 3.8.3, "Special Files." The **deletefile**, **renamefile**, and **filenameforall** operators do not apply to file names of this form.

"Wildcard" file names are recognized by the **filenameforall** operator; see **filenameforall** in Chapter 8 for more information.

Creating and Closing a File Object

File objects are created by the **file** operator. This operator takes two strings: the first identifies the file and the second specifies access. **file** returns a new file object associated with that file.

An *access string* is a string object that specifies how a file is to be accessed. File access conventions are similar to the ones defined by the ANSI C standard, although some file systems may not support all access methods. The access string always begins with r, w, or a, possibly followed by +; any additional characters supply operating system–specific information. Table 3.5 lists access strings and their meanings.

TABLE 3.5 Access strings

ACCESS STRING	MEANING
r	Open for reading only. Generate an error if the file does not already exist.
w	Open for writing only. Create the file if it does not already exist. Truncate and overwrite it if it does exist.
a	Open for writing only. Create the file if it does not already exist. Append to it if it does exist.
r+	Open for reading and writing. Generate an error if the file does not already exist.
w+	Open for reading and writing. Create the file if it does not already exist. Truncate and overwrite it if it does exist.
a+	Open for reading and writing. Create the file if it does not already exist. Append to it if it does exist.

Note: *The special files %stdin, %lineedit, and %statementedit allow only r access; %stdout and %stderr allow only w access (see Section 3.8.3, "Special Files").*

Like other composite objects, such as strings and arrays, file objects have access attributes. The access attribute of a file object is based on the access string used to create it. Attempting to access a file object in a way that would violate its access attribute causes an **invalidaccess** error.

Certain files—in particular, named files on disk—are *positionable*, meaning that the data in the file can be accessed in an arbitrary order rather than only sequentially from the beginning. The **setfileposition** operator adjusts a file object so that it refers to a specified position in the underlying file; subsequent reads or writes access the file at that new position. Specifying a plus sign (+) in the access string opens a positionable file for reading and writing, as shown in Table 3.5. To ensure predictable results, it is necessary to execute **setfileposition** when switching between reading and writing.

At the end of reading or writing a file, a program should *close* the file to break the association between the PostScript file object and the actual file. The file operators close a file automatically if end-of-file is encountered during reading (see below). The **closefile** operator closes a file explicitly. **restore** closes a file if the file object was created since the corresponding **save** operation while in local VM allocation mode. Garbage collection closes a file if the file object is no longer accessible.

All operators that access files treat end-of-file and exception conditions the same. During reading, if an end-of-file indication is encountered before the requested item can be read, the file is closed and the operation returns an explicit end-of-file result. This also occurs if the file has already been closed when the operator is executed. All other exceptions during reading and any exceptions during writing result in execution of the error **ioerror**, **invalidfileaccess**, or **invalidaccess**.

3.8.3 Special Files

The **file** operator can also return special files that are identified as follows:

- %stdin, the standard input file.
- %stdout, the standard output file.
- %stderr, the standard error file. This file is for reporting low-level errors. In many configurations, it is the same as the standard output file.
- %statementedit, the statement editor filter file, described below.
- %lineedit, the line editor filter file, described below.

For example, the statements

```
(%stdin) (r) file
(%stdout) (w) file
```

push copies of the standard input and output file objects on the operand stack. These are duplicates of existing file objects, not new objects. Each execution of the **file** operator for %stdin, %stdout, or %stderr within a given job returns the same file object. A PostScript program should not close these files. In an interpreter that supports multiple execution contexts, the standard input and output files are private to each context; the standard error file is shared among all contexts.

Some PostScript interpreters support an *interactive executive,* invoked by the **executive** operator; this is described in Section 2.4.4, "Using the Interpreter Interactively." **executive** obtains commands from the user by means of a special file named %statementedit. Applying the **file** operator to the file name string %statementedit causes the following to happen:

- The **file** operator begins reading characters from the standard input file and storing them in a temporary buffer. While doing so, it echoes the characters to the standard output file. It also interprets certain control characters as editing functions for making corrections, as described in Section 2.4.4.
- When a complete statement has been entered, the **file** operator returns. A statement consists of one or more lines terminated by a newline that together form one or more complete PostScript tokens, with no opening brackets ({, (, <, or <~) left unmatched. A statement is also considered complete if it contains a syntax error.
- The returned file object represents a temporary file containing the statement that was entered, including the terminating end-of-line character. Reading from this file obtains the characters of the statement in turn; end-of-file is reported when the end of the statement is reached. Normally, this file is used as an operand to the **exec** operator, causing the statement to be executed as a PostScript program.

The %lineedit special file is similar to %statementedit, except that when reading from %lineedit, the **file** operator returns after a single line has been entered, whether or not it constitutes a complete statement. For both the special files %statementedit and %lineedit, if the standard input file reaches end-of-file before

any characters have been entered, the **file** operator issues an **undefinedfilename** error.

It is important to understand that the file object returned by **file** for the %statementedit and %lineedit special files is not the same as the standard input file. It represents a temporary file containing a single buffered statement. When the end of that statement is reached, the file is closed and the file object is no longer of any use. Successive executions of **file** for %statementedit and %lineedit return different file objects.

The %statementedit and %lineedit special files are not available in PostScript interpreters that do not support an interactive executive. PostScript programs that are page descriptions should never refer to these files.

3.8.4 Filters

A *filter (LanguageLevel 2)* is a special kind of file object that can be layered on top of some other file to transform data being read from or written to that file. When a PostScript program reads characters from an input filter, the filter reads characters from its underlying file and transforms the data in some way, depending on the filter. Similarly, when a program writes characters to an output filter, the filter transforms the data and writes the results to its underlying file.

An *encoding filter* is an output file that takes the data written to it, converts it to some encoded representation depending on the filter, and writes the encoded data to the underlying file. For example, the **ASCIIHexEncode** filter transforms binary data to an ASCII hexadecimal-encoded representation, which it writes to its underlying file. All encoding filters have **Encode** as part of their names.

A *decoding filter* is an input file that reads encoded data from its underlying file and decodes it. The program reading from the filter receives the decoded data. For example, the **ASCIIHexDecode** filter reads ASCII hexadecimal-encoded data from its underlying file and transforms it to binary. All decoding filters have **Decode** as part of their names.

Decoding filters are most likely to be used in page descriptions. An application program generating a page description can encode certain information (for example, data for sampled images) to compress it or to convert it to a portable ASCII representation. Then, within the page description itself, it invokes the corresponding decoding filter to convert the information back to its original form.

Encoding filters are unlikely to be used in most page descriptions. However, a PostScript program can use them to encode data to be sent back to the application or written to a disk file. In the interest of symmetry, the PostScript language defines both encoding and decoding filters for all of its standard data transformation algorithms. However, encoding filters are optional; not all PostScript interpreters support them.

Creating Filters

Filter files are created by the **filter** operator *(LanguageLevel 2)*. The **filter** operator expects the following operands in the order given:

1. A *data source* or *data target*. This is ordinarily a file object that represents the underlying file the filter is to read or write. However, it can also be a string or a procedure. Details are provided in Section 3.13.1, "Data Sources and Targets."
2. *Filter parameters*. All filters may take additional parameters, and some require additional parameters, to control how they operate. These parameters may be specified in a dictionary given as an operand following the data source or target; in some cases, required parameters must be given as operands following the data source or target or following the dictionary operand, if any. The dictionary operand may be omitted whenever all the dictionary-supplied parameters have the corresponding default values for that filter. Exactly which parameters and operands are required for the various filters is described in Section 3.13, "Filtered Files Details."
3. *Filter name*. This is a name object, such as **ASCIIHexDecode**, that specifies the data transformation the filter is to perform. It also determines how many parameters there are and how they are to be interpreted.

The **filter** operator returns a new file object that represents the filtered file. For an encoding filter, this is an output file, and for a decoding filter, an input file. The direction of the underlying file—that is, its read/write attribute—must match that of the filter. Filtered files can be used just the same as other files; they are valid as operands to file operators such as **read**, **write**, **readstring**, and **writestring**. Input filters are also valid as data sources for operators such as **exec** or **image**.

Since a filter is itself a file, it can be used as the underlying file for yet another filter. Filters can be cascaded to form a *pipeline* that passes the data stream through two or more encoding or decoding transformations in sequence. Example 3.5 illustrates the construction of an input pipeline for decoding sampled image data

that is embedded in the program. The application has encoded the image data twice: once using the **RunLengthEncode** method to compress the data, and then using the **ASCII85Encode** method to represent the binary compressed data as ASCII text.

Example 3.5

```
256 256 8 [256 0 0 –256 0 256]      % Other operands of the image operator
currentfile
/ASCII85Decode filter
/RunLengthDecode filter
image
… Encoded image data …
~>                                  % ASCII85 end-of-data marker
```

The **currentfile** operator returns the file object from which the PostScript interpreter is currently executing. The first execution of **filter** creates an **ASCII85Decode** filter whose underlying file is the one returned by **currentfile**. It pushes the filter file object on the stack. The second execution of **filter** creates a **RunLengthDecode** filter whose underlying file is the first filter file; it pushes the new filter file object on the stack. Finally, the **image** operator uses the second filter file as its data source. As **image** reads from its data source, the data is drawn from the underlying file and transformed by the two filters in sequence.

Standard Filters

The PostScript language supports a standard set of filters that fall into three main categories:

- *ASCII encoding and decoding filters* enable arbitrary 8-bit binary data to be represented in the printable subset of the ASCII character set. This improves the portability of the resulting data, since it avoids the problem of interference by operating systems or communication channels that preempt the use of control characters, represent text as 7-bit bytes, or impose line-length restrictions.
- *Compression and decompression filters* enable data to be represented in a compressed form. Compression is particularly valuable for large sampled images, since it reduces storage requirements and transmission time. There are several compression filters, each of which is best suited for particular kinds of data. Note that the compressed data is in 8-bit binary format, even if the original data happens to be ASCII text. For maximum portability of the encoded data,

these filters should be used with ASCII encoding filters, as illustrated above in Example 3.5.

- *Subfile filters* pass data through without modification. These filters permit the creation of file objects that access arbitrary user-defined data sources or data targets. Input filters also can read data from an underlying file up to a specified end-of-data marker.

Table 3.6 summarizes the available filters. A program can determine the complete set of filters that the PostScript interpreter supports by applying the **resourceforall** operator to the **Filter** resource category; see Section 3.9, "Named Resources."

TABLE 3.6 Standard filters

FILTER NAME	REQUIRED PARAMETERS	DESCRIPTION
ASCIIHexEncode	(none)	Encodes binary data in an ASCII hexadecimal representation. Each binary data byte is converted to two hexadecimal digits, resulting in an expansion factor of 1:2 in the size of the encoded data.
ASCIIHexDecode	(none)	Decodes ASCII hexadecimal-encoded data, producing the original binary data.
ASCII85Encode	(none)	Encodes binary data in an ASCII base-85 representation. This encoding uses nearly all of the printable ASCII character set. The resulting expansion factor is 4:5, making this encoding much more efficient than hexadecimal.
ASCII85Decode	(none)	Decodes ASCII base-85 data, producing the original binary data.
LZWEncode	(none)	Compresses data using the LZW (Lempel-Ziv-Welch) adaptive compression method, optionally after pretransformation by a predictor function. This is a good general-purpose encoding that is especially well suited for natural-language and PostScript-language text, but it is also useful for image data.
LZWDecode	(none)	Decompresses LZW-encoded data, producing the original data.
FlateEncode	(none)	*(LanguageLevel 3)* Compresses data using the public-domain zlib/deflate compression method, optionally after pretransformation by a predictor function. This is a variable-length Lempel-Ziv adaptive compression method cascaded with adaptive Huffman coding. It is a good general-purpose encoding that is especially well suited for natural-language and PostScript-language text, but it is also useful for image data.

FlateDecode	(none)	*(LanguageLevel 3)* Decompresses data encoded in zlib/deflate compressed format, producing the original data.
RunLengthEncode	record size	Compresses data using a simple byte-oriented run-length encoding algorithm. This encoding is best suited to monochrome image data, or any data that contains frequent long runs of a single byte value.
RunLengthDecode	(none)	Decompresses data encoded in the run-length encoding format, producing the original data.
CCITTFaxEncode	(none)	Compresses data using a bit-oriented encoding algorithm (the CCITT facsimile standard). This encoding is specialized to monochrome image data at 1 bit per pixel.
CCITTFaxDecode	(none)	Decompresses facsimile-encoded data, producing the original data.
DCTEncode	dictionary	Compresses continuous-tone (grayscale or color) sampled image data using a DCT (discrete cosine transform) technique based on the JPEG standard. This encoding is specialized to image data. It is "lossy," meaning that the encoding algorithm can lose some information.
DCTDecode	(none)	Decompresses DCT-encoded data, producing image sample data that approximate the original data.
ReusableStreamDecode	(none)	*(LanguageLevel 3)* From any data source, creates an input stream that can be treated as a random-access, repositionable file.
NullEncode	(none)	Passes all data through, without any modification. This permits an arbitrary data target (procedure or string) to be treated as an output file.
SubFileDecode	count, string	Passes all data through, without any modification. This permits an arbitrary data source (procedure or string) to be treated as an input file. Optionally, this filter detects an end-of-data marker in the source data stream, treating the preceding data as a subfile.

Note: *In LanguageLevel 3, all encoding filters, with the exception of the* ***NullEncode*** *filter, are optional—that is, they may or may not be present in a PostScript interpreter product. Additional nonstandard filters may be available in some products. To ensure portability, PostScript programs that are page descriptions should not depend on optional or nonstandard filters.*

Section 3.13, "Filtered Files Details," provides complete information about individual filters, including specifications of the encoding algorithms for some of

them. The section also describes the semantics of data sources and data targets in more detail.

3.8.5 Additional File Operators

There are other miscellaneous file operators:

- **status** and **bytesavailable** return status information about a file.
- **currentfile** returns the file object from which the interpreter is currently reading.
- **run** is a convenience operator that combines the functions of **file** and **exec**.

Several built-in procedures print the values of objects on the operand stack, sending a readable representation of those values to the standard output file:

- = pops one object from the operand stack and writes a text representation of its value to the standard output file, followed by a newline.
- == is similar to =, but produces results closer to full PostScript language syntax and expands the values of arrays.
- **stack** prints the entire contents of the operand stack with =, but leaves the stack unchanged.
- **pstack** performs a similar operation to **stack**, but uses ==.

Input/output and storage devices can be manipulated individually by LanguageLevel 2 operators. In particular:

- **setdevparams** and **currentdevparams** access device-dependent parameters (see Appendix C).
- **resourceforall**, applied to the **IODevice** resource category, enumerates all available device parameter sets (see the next section).

3.9 Named Resources

Some features of the PostScript language involve the use of open-ended collections of objects to control their operation. For example, the font machinery uses font dictionaries that describe the appearance of characters. The number of possible font dictionaries is unlimited. In LanguageLevels 2 and 3, this same idea

applies to forms, patterns, color rendering dictionaries, and many other categories of objects.

It is often convenient to associate these objects with names in some central registry. This is particularly true for fonts, which are assigned standard names (such as Times-Roman or Palatino-BoldItalic) when they are created. Other categories of objects also can benefit from a central naming convention.

If all available objects in a particular category (for example, all possible fonts) were *permanently* resident in VM, they could simply be stored in some dictionary. Accessing a named object would be a matter of performing **get** from the dictionary; checking whether a named object is available would be accomplished by performing a **known** operation on the dictionary.

There are many more fonts and objects of other categories than can possibly reside in VM at any given time. These objects originate from a source external to the PostScript interpreter. They are introduced into VM in two ways:

- The application or print spooler embeds the objects' definitions directly in the job stream.
- During execution, the PostScript program requests the objects by name. The interpreter loads them into VM automatically from an external source, such as a disk file, a ROM cartridge, or a network file server.

The notion of *named resources (LanguageLevel 2)* supports the second method. A *resource* is a collection of named objects that either reside in VM or can be located and brought into VM on demand. There are separate categories of resources with independent name spaces; for example, fonts and forms are distinct resource categories. Within each category, there is a collection of named resource instances. Each category can have its own policy for locating instances that are not in VM and for managing the instances that are in VM.

3.9.1 Resource Operators

There are five LanguageLevel 2 operators that apply to resources: **findresource**, **resourcestatus**, **resourceforall**, **defineresource**, and **undefineresource**. A more limited pair of operators applicable only to fonts, **findfont** and **definefont**, are available in LanguageLevel 1.

The **findresource** operator is the key feature of the resource facility. Given a resource category name and an instance name, **findresource** returns an object. If the requested resource instance does not already exist as an object in VM, **findresource** gets it from an external source and loads it into VM. A PostScript program can access named resources without knowing whether they are already in VM or how they are obtained from external storage.

Other important features include **resourcestatus**, which returns information about a resource instance, and **resourceforall**, which enumerates all available resource instances in a particular category. These operators apply to all resource instances, whether or not they reside in VM; the operators do not cause the resource instances to be brought into VM. **resourceforall** should be used with care and only when absolutely necessary, since the set of available resource instances is potentially extremely large.

A program can explicitly define a named resource instance in VM. That is, it can create an object in VM, then execute **defineresource** to associate the object with a name in a particular resource category. This resource instance will be visible in subsequent executions of **findresource**, **resourcestatus**, and **resourceforall**. A program can also execute **undefineresource** to reverse the effect of a prior **defineresource**. The **findresource** operator automatically executes **defineresource** and **undefineresource** to manage VM for resource instances that it obtains from external storage.

Resource instances can be defined in either local or global VM. The lifetime of the definition depends on the VM allocation mode in effect at the time the definition is made (see Section 3.7.2, "Local and Global VM"). Normally, both local and global resource instances are visible and available to a program. However, when the current VM allocation mode is global, only global instances are visible; this ensures correct behavior of resource instances that are defined in terms of other resource instances.

When a program executes **defineresource** to define a resource instance explicitly, the program has complete control over whether to use local or global VM. However, when execution of **findresource** causes a resource instance to be brought into VM automatically, the decision whether to use local or global VM is independent of the VM allocation mode at the time **findresource** is executed. Usually, resource instances are loaded into global VM; this enables them to be managed independently of the **save** and **restore** activity of the executing program. However, certain resource instances do not function correctly when they reside in glo-

bal VM; they are loaded into local VM instead. In general, PostScript programs using resources should not depend on knowing anything about the policies used by the resource machinery, since those policies can vary among different resource implementations.

The language does not specify a standard method for installing resources in external storage. Installation typically consists of writing to a named file in a file system. However, details of how resource names are mapped to file names and how the files are managed are environment-dependent. In some environments, resources may be installed using facilities entirely separate from the PostScript interpreter.

Resource instances are identified by keys that ordinarily are name or string objects; the resource operators treat names and strings equivalently. Use of other types of keys is permitted but not recommended. The **defineresource** operator can define a resource instance with a key that is not a name or a string, and the other resource operators can access the instance using that key. However, such a key can never match any resource instance in external storage.

3.9.2 Resource Categories

Resource categories are identified by name. Tables 3.7, 3.8, and 3.9 list the standard resource categories. Within a given category, every resource instance that resides in VM is of a particular type and has a particular intended interpretation or use.

Regular resources are those whose instances are ordinary useful objects, such as font or halftone dictionaries. For example, a program typically uses the result returned by **findresource** as an operand of some other operator, such as **scalefont** or **sethalftone**.

*Implicit resource*s represent some built-in capability of the PostScript interpreter. For example, the instances of the **Filter** category are filter names, such as **ASCII85Decode** and **CCITTFaxDecode**, that are passed directly to the **filter** operator. For such resources, the **findresource** operator returns only its name operand. However, **resourceforall** and **resourcestatus** are useful for inquiring about the availability of capabilities such as specific filter algorithms.

TABLE 3.7 Regular resources

CATEGORY NAME	OBJECT TYPE	DESCRIPTION
Font	dictionary	Font definition
CIDFont	dictionary	CIDFont definition (*LanguageLevel 3*)
CMap	dictionary	Character code mapping (*LanguageLevel 3*)
FontSet	dictionary	Bundle of font definitions (*LanguageLevel 3*)
Encoding	array	Encoding vector
Form	dictionary	Form definition
Pattern	dictionary	Pattern definition (prototype)
ProcSet	dictionary	Procedure set
ColorSpace	array	Parameterized color space
Halftone	dictionary	Halftone dictionary
ColorRendering	dictionary	Color rendering dictionary
IdiomSet	dictionary	Procedure substitution dictionary *(LanguageLevel 3)*
InkParams	dictionary	Colorant details dictionary *(LanguageLevel 3)*
TrapParams	dictionary	Trapping parameter set *(LanguageLevel 3)*
OutputDevice	dictionary	Page device capabilities *(LanguageLevel 3)*
ControlLanguage	dictionary	Control language support *(LanguageLevel 3)*
Localization	dictionary	Natural language support *(LanguageLevel 3)*
PDL	dictionary	PDL interpreter support *(LanguageLevel 3)*
HWOptions	dictionary	Hardware options *(LanguageLevel 3)*

TABLE 3.8 Resources whose instances are implicit

CATEGORY NAME	OBJECT TYPE	DESCRIPTION
Filter	name	Filter algorithm
ColorSpaceFamily	name	Color space family
Emulator	name	Language interpreter
IODevice	string	Device parameter set

ColorRenderingType	integer	Color rendering dictionary type
FMapType	integer	Composite font mapping algorithm
FontType	integer	Font dictionary type
FormType	integer	Form dictionary type
HalftoneType	integer	Halftone dictionary type
ImageType	integer	Image dictionary type
PatternType	integer	Pattern dictionary type
FunctionType	integer	Function dictionary type *(LanguageLevel 3)*
ShadingType	integer	Shading dictionary type *(LanguageLevel 3)*
TrappingType	integer	Trapping method *(LanguageLevel 3)*

TABLE 3.9 Resources used in defining new resource categories

CATEGORY NAME	OBJECT TYPE	DESCRIPTION
Category	dictionary	Resource category (recursive)
Generic	any	Prototype for new categories

The **Category** and **Generic** resources are used in defining new categories of resources. This capability is described in Section 3.9.3, "Creating Resource Categories."

The resource operators—**findresource**, **resourcestatus**, **resourceforall**, **defineresource**, and **undefineresource**—have standard behavior that is uniform across all resource categories. This behavior is specified in the operator descriptions in Chapter 8. For some categories, the operators have additional semantics that are category-specific. The following sections describe the semantics of each resource category.

Note: *Except as indicated below, the PostScript language does not prescribe that a resource category must contain any standard instances. Some categories may be populated with predefined instances, but the set of instances is product-dependent.*

Font

Instance names of the **Font** resource category are font names, such as Times-Roman. The instances are font dictionaries that are suitable for use as operands to **scalefont** or **makefont**, which produce a transformed font dictionary that can be used to paint characters on the page.

The following special-purpose operators apply only to fonts but are otherwise equivalent to the resource operators:

- **findfont**, equivalent to /Font findresource
- **definefont**, equivalent to /Font defineresource
- **undefinefont**, equivalent to /Font undefineresource

The **definefont** and **undefinefont** operators have additional font-specific semantics, which are described under those operators in Chapter 8. Those semantics also apply to **defineresource** and **undefineresource** when applied to the **Font** category. **findfont** and **definefont** are available in LanguageLevel 1, even though the general facility for named resources is a LanguageLevel 2 feature.

The font operators also maintain dictionaries of font names and **Font** resource instances that are defined in VM. Those dictionaries are **FontDirectory** (all **Font** resources in VM) and **GlobalFontDirectory** (only **Font** resources in global VM). They are obsolete, but are provided for compatibility with existing applications. The preferred method of enumerating all available **Font** resources is

```
(*) proc scratch /Font resourceforall
```

where *proc* is a procedure and *scratch* is a string used repeatedly to hold font names. This method works for all available **Font** resources, whether or not they are in VM. Normally, it is preferable to use **resourcestatus** to determine the availability of specific resources rather than enumerate all resources and check whether those of interest are in the list.

When **findresource** or **findfont** loads a font from an external source into VM, it may choose to use global VM rather than the current VM allocation mode. This choice depends on memory management algorithms used by the interpreter. It also depends on the font type, since certain Type 3 fonts do not work correctly when loaded into global VM. The details of this policy are implementation-dependent; a PostScript program should not depend on knowing what they are.

CIDFont

Instances of the **CIDFont** resource category *(LanguageLevel 3)* are dictionaries that are suitable for use with the **composefont** operator to construct CID-keyed fonts, as described in Section 5.11, "CID-Keyed Fonts." The **defineresource** operator has certain category-specific semantics when applied to the **CIDFont** category; furthermore, the **definefont** and **undefinefont** operators can be applied to CIDFonts as well as fonts. For more information on the behavior of these operators, see Section 5.11.3, "CIDFont Dictionaries."

CMap

Instances of the **CMap** resource category *(LanguageLevel 3)* are character code mapping dictionaries that are suitable for use with the **composefont** operator to construct CID-keyed fonts, as described in Section 5.11, "CID-Keyed Fonts."

FontSet

Instances of the **FontSet** resource category *(LanguageLevel 3)* are bundles of font definitions that are represented in the Compact Font Format (CFF) or other multiple-font representations, as described in Section 5.8.1, "Type 2 and Type 14 Fonts (CFF and Chameleon)." Each **FontSet** instance contains the material from which one or more **Font** instances can be constructed.

Encoding

Instances of the **Encoding** resource category are array objects, suitable for use as the **Encoding** entry of font dictionaries (see Section 5.3, "Character Encoding"). An encoding array usually contains 256 names, permitting it to be indexed by any 8-bit character code. An encoding array for use with composite fonts (described in Section 5.10, "Composite Fonts") contains integers instead of names, and can be of any length.

There are two standard encodings that are permanently defined in VM and available by name in **systemdict**:

- **StandardEncoding**, whose value is the same as the array returned by
  /StandardEncoding /Encoding findresource

- **ISOLatin1Encoding**, whose value is the same as the array returned by
  /ISOLatin1Encoding /Encoding findresource

If any other encodings exist, they are available only through **findresource**. The convenience operator **findencoding** is equivalent to /Encoding findresource.

Form

Instances of the **Form** resource category are form dictionaries, described in Section 4.7, "Forms." A form dictionary is suitable as the operand to the **execform** operator to render the form on the page.

Pattern

Instances of the **Pattern** resource category are prototype pattern dictionaries, described in Section 4.9, "Patterns." A prototype pattern dictionary is suitable as the operand to the **makepattern** operator, which produces a transformed pattern dictionary; a PostScript program can then use the resulting dictionary in painting operations by establishing a **Pattern** color space or by invoking the **setpattern** operator.

ProcSet

Instances of the **ProcSet** resource category are *procedure sets*. A procedure set is a dictionary containing named procedures or operators. Application prologs can be organized as one or more procedure sets that are available from a library instead of being included in-line in every document that uses them. The **ProcSet** resource category provides a way to organize such a library.

In LanguageLevel 3, there are several standard instances of the **ProcSet** category that are associated with specific features of the PostScript language. These procedure sets, listed in Table 3.10, contain procedures, operators, and other objects that a PostScript program can access as part of using those features.

TABLE 3.10 Standard procedure sets in LanguageLevel 3

PROCEDURE SET	ASSOCIATED LANGUAGE FEATURE
BitmapFontInit	Incremental downloading and management of glyph bitmaps in a Type 4 CIDFont (see "Type 4 CIDFonts" on page 379)
CIDInit	Building a Type 0 CIDFont ("Type 0 CIDFonts" on page 371) or a CMap dictionary (Section 5.11.4, "CMap Dictionaries")
ColorRendering	Selecting a color rendering dictionary (Section 7.1.3, "Rendering Intents")
FontSetInit	Building a **FontSet** resource ("FontSet Resources" on page 344)
Trapping	In-RIP trapping (Section 6.3, "In-RIP Trapping")

ColorSpace

Instances of the **ColorSpace** resource category are array objects that represent fully parameterized color spaces. The first element of a color space array is a color space family name; the remaining elements are parameters to the color space (see Section 4.8, "Color Spaces").

Note: *The* ***ColorSpace*** *resource category is distinct from the* ***ColorSpaceFamily*** *category, described below.*

Halftone

Instances of the **Halftone** resource category are halftone dictionaries, suitable as operands to the **sethalftone** operator (see Section 7.4, "Halftones").

ColorRendering

Instances of the **ColorRendering** resource category are color rendering dictionaries, suitable as operands to the **setcolorrendering** operator (see Section 7.1, "CIE-Based Color to Device Color").

IdiomSet

Instances of the **IdiomSet** resource category *(LanguageLevel 3)* are procedure substitution dictionaries, for use with the **bind** operator (see Section 3.12.1, "bind Operator").

InkParams and TrapParams

The LanguageLevel 3 resource categories **InkParams** and **TrapParams** are present only in products that support in-RIP trapping (see Section 6.3, "In-RIP Trapping"). Instances of **InkParams** are dictionaries that define trapping-related properties of device colorants; instances of **TrapParams** are dictionaries that define sets of trapping parameters suitable as operands to the **settrapparams** operator.

OutputDevice

Instances of the **OutputDevice** resource category *(LanguageLevel 3)* are dictionaries that describe certain capabilities of a particular page device, such as the possible page sizes or resolutions (see Section 6.4, "Output Device Dictionary").

ControlLanguage, PDL, Localization, and HWOptions

Instances of the LanguageLevel 3 resource categories **ControlLanguage**, **PDL**, **Localization**, and **HWOptions** provide information that is product-dependent, as summarized below. For further details, see the *PostScript Language Reference Supplement*.

- Instances of **ControlLanguage** are dictionaries that describe the control languages available in a product. A control language is a means for controlling product features, such as default configuration and status reporting.
- Instances of **PDL** are dictionaries that describe the page description language interpreters available in a product. This category supersedes the **Emulator** implicit resource category, because its instances provide a more complete description of each interpreter (or emulator).
- Instances of **Localization** are dictionaries that describe the natural languages (for example, English, Japanese, or German) supported by a product.

- Instances of **HWOptions** are strings that indicate the special hardware options that are present in this product.

Implicit Resources

For all implicit resources, the **findresource** operator returns the instance's key if the instance is defined. The **resourcestatus** and **resourceforall** operators have their normal behavior, although the *status* and *size* values returned by **resourcestatus** are meaningless. The **defineresource** and **undefineresource** operators are ordinarily not allowed, but the ability to define new instances of implicit resources may exist in some implementations. The mechanisms are implementation-dependent.

The instances of the **Filter** category are filter names, such as **ASCII85Decode** and **RunLengthEncode**, which are used as an operand of the **filter** operator to determine its behavior. Filters are described in Section 3.8.4, "Filters."

The instances of the **ColorSpaceFamily** category are color space family names, which appear as the first element of a color space array object. Some color spaces, such as **DeviceRGB**, are completely determined by their family name; others, such as **CIEBasedABC**, require additional parameters to describe them. Color spaces are described in Section 4.8, "Color Spaces."

The instances of the **Emulator** category are names of emulators for languages other than PostScript that may be built into a particular implementation. Those emulators are not a standard part of the PostScript language, but one or more of them may be present in some products. This category has been superseded by the **PDL** resource category in LanguageLevel 3.

The instances of the **IODevice** category are names of device parameter sets. Some parameter sets are associated with input/output devices, from which the category name **IODevice** originates. However, there are also some parameter sets that do not correspond to physical devices. The keys for all instances of this category are expressed as strings of the form %*device*%. See Section C.4, "Device Parameters."

The instances of the **ColorRenderingType**, **FMapType**, **FontType**, **FormType**, **HalftoneType**, **ImageType**, **PatternType**, **FunctionType**, **ShadingType**, and **TrappingType** categories are integers that are the acceptable values for the correspondingly named entries in various classes of special dictionaries. For example, in LanguageLevel 3 the **FMapType** category includes the integers 1 through 9 as

keys; if an interpreter supports additional **FMapType** values, the **FMapType** category will also include those values as instances.

3.9.3 Creating Resource Categories

The language support for named resources is quite general. Most of it is independent of the semantics of specific resource categories. It is occasionally useful to create new resource categories, each containing an independent collection of named instances. This is accomplished through a level of recursion in the resource machinery itself.

The resource category named **Category** contains all of the resource categories as instances. The instance names are resource category names, such as **Font**, **Form**, and **Halftone**. The instance values are dictionary objects containing information about how the corresponding resource category is implemented.

A new resource category is created by defining a new instance of the **Category** category. Example 3.6 creates a category named Widget.

Example 3.6

```
true setglobal
    /Widget catdict /Category defineresource pop
false setglobal
```

In this example, *catdict* is a dictionary describing the implementation of the Widget category. Once it is defined, instances of the Widget category can be manipulated like other categories:

```
/Frob1 w /Widget defineresource          % Returns w
/Frob1 /Widget findresource              % Returns w
/Frob1 /Widget resourcestatus            % Returns status size true
(*) proc scratch /Widget resourceforall  % Pushes (Frob1) on the stack, then calls proc
```

Here *w* is an instance of the Widget category whose type is whatever is appropriate for widgets, and /Frob1 is the name of that instance.

It is possible to redefine existing resource categories in this way. Programs that do this must ensure that the new definition correctly implements any special semantics of the category.

Category Implementation Dictionary

The behavior of all the resource operators, such as **defineresource**, is determined by entries in the resource category's implementation dictionary. This dictionary was supplied as an operand to **defineresource** when the category was created. In the example

```
/Frob1 w /Widget defineresource
```

the **defineresource** operator does the following:

1. Obtains *catdict*, the implementation dictionary for the Widget category.
2. Executes **begin** on the implementation dictionary.
3. Executes the dictionary's **DefineResource** entry, which is ordinarily a procedure but might be an operator. When the procedure corresponding to the **DefineResource** entry is called, the operand stack contains the operands that were passed to **defineresource**, except that the category name (Widget in this example) has been removed. **DefineResource** is expected to consume the remaining operands, perform whatever action is appropriate for this resource category, and return the appropriate result.
4. Executes the **end** operator. If an error occurred during step 3, it also restores the operand and dictionary stacks to their initial state.

The other resource operators—**undefineresource**, **findresource**, **resourcestatus**, and **resourceforall**—behave the same way, with the exception that **resourceforall** does not restore the stacks upon error. Aside from the steps described above, all of the behavior of the resource operators is implemented by the corresponding procedures in the dictionary.

A category implementation dictionary contains the entries listed in Table 3.11. The dictionary may also contain other information useful to the procedures in the dictionary. Since the dictionary is on the dictionary stack at the time those procedures are called, the procedures can access the information conveniently.

TABLE 3.11 Entries in a category implementation dictionary

KEY	TYPE	VALUE
DefineResource	procedure	*(Required)* A procedure that implements **defineresource** behavior.
UndefineResource	procedure	*(Required)* A procedure that implements **undefineresource** behavior.
FindResource	procedure	*(Required)* A procedure that implements **findresource** behavior. This procedure determines the policy for using global versus current VM when loading a resource from an external source.
ResourceStatus	procedure	*(Required)* A procedure that implements **resourcestatus** behavior.
ResourceForAll	procedure	*(Required)* A procedure that implements **resourceforall** behavior. This procedure should remove the category implementation dictionary from the dictionary stack before executing the procedure operand of **resourceforall**, and should put that dictionary back on the dictionary stack before returning. This ensures that the procedure operand is executed in the dictionary context in effect at the time **resourceforall** was invoked.
Category	name	*(Required)* The category name. This entry is inserted by **defineresource** when the category is defined.
InstanceType	name	*(Optional)* The expected type of instances of this category. If this entry is present, **defineresource** checks that the instance's type, as returned by the **type** operator, matches it.
ResourceFileName	procedure	*(Optional)* A procedure that translates a resource instance name to a file name (see Section 3.9.4, "Resources as Files").

A single dictionary provides the implementation for both local and global instances of a category. The implementation must maintain the local and global instances separately and must respect the VM allocation mode in effect at the time each resource operator is executed. The category implementation dictionary must be in global VM; the **defineresource** operator that installs it in the **Category** category must be executed while in global VM allocation mode.

The interpreter assumes that the category implementation procedures will be reasonably well behaved and will generate errors only due to circumstances not under their control. In this respect, they are similar to the **BuildChar** procedure in a Type 3 font or to the **PaintProc** procedure in a form or pattern, but are unlike the arbitrary procedures invoked by operators such as **forall** or **resourceforall**.

If an error occurs in a category implementation procedure, the resource operator makes a token attempt to restore the stacks and to provide the illusion that the error arose from the operator itself. The intent is that the resource operators should have the usual error behavior as viewed by a program executing them. The purpose is not to compensate for bugs in the resource implementation procedures.

Generic Category

The preceding section describes a way to define a new resource category, but it does not provide guidance about how the individual procedures in the category's dictionary should be implemented. In principle, every resource category has complete freedom over how to organize and manage resource instances, both in VM and in external storage.

Since different implementations have different conventions for organizing resource instances, especially in external storage, a program that seeks to create a new resource category might need implementation-dependent information. To overcome this problem, it is useful to have a generic resource implementation that can be copied and used to define new resource categories. The **Category** category contains an instance named **Generic**, whose value is a dictionary containing a generic resource implementation.

Example 3.7 defines the Widget resource category and is similar to Example 3.6 on page 99; however, it generates the category implementation dictionary by copying the one belonging to the Generic category. This avoids the need to know anything about how resource categories actually work.

Example 3.7

```
currentglobal                          % Save the current VM status on the stack.
true setglobal
    /Generic /Category findresource
    dup length 1 add dict copy
    dup /InstanceType /dicttype put
    /Widget exch /Category defineresource pop
setglobal                              % Restore the saved VM status.
```

The **Generic** resource category's implementation dictionary does not have an **InstanceType** entry; instances need not be of any particular type. The example above makes a copy of the dictionary with space for one additional entry and in-

serts an **InstanceType** entry with the value dicttype. As a result, **defineresource** requires that instances of the Widget category be dictionaries.

3.9.4 Resources as Files

The PostScript language does not specify how external resources are installed, how they are loaded, or what correspondence, if any, exists between resource names and file names. In general, all knowledge of such things is in the category implementation dictionary and in environment-dependent installation software.

Typically, resource instances are installed as named files, which can also be accessed by ordinary PostScript file operators such as **file** and **run**. There is a straightforward mapping from resource names to file names, though the details of this mapping vary because of restrictions on file name syntax imposed by the underlying file system.

In some implementations, including many dedicated printers, the only access to the file system is through the PostScript interpreter. In such environments, it is important for PostScript programs to be able to access the underlying resource files directly in order to install or remove them. Only resource installation or other system management software should do this. Page descriptions should never attempt to access resources as files; they should use only resource operators, such as **findresource**.

The implementation dictionary for a category can contain an optional entry, **ResourceFileName**, which is a procedure that translates from a resource name to a file name. If the procedure exists, a program can call it as follows:

1. Push the category implementation dictionary on the dictionary stack. The **ResourceFileName** procedure requires this step in order to obtain category-specific information, such as **Category**.
2. Push the instance name and a scratch string on the operand stack. The scratch string must be long enough to accept the complete file name for the resource.
3. Execute **ResourceFileName**.
4. Pop the dictionary stack.

ResourceFileName builds a complete file name in the scratch string and returns on the operand stack the substring that was used. This string can then be used as

the *filename* operand of file operators such as **file**, **deletefile**, **status**, and so on. For example, the following program fragment obtains the file name for the Times-Roman font:

```
/Font /Category findresource
begin
    /Times-Roman scratch ResourceFileName
end
```

If a **ResourceFileName** procedure for a particular category and instance name exists and executes without a PostScript error, it will leave a string on the stack. If that category maintains all of its instances as named files, this string is the name of the file for that instance. This file name may or may not contain the %*device*% prefix. Use of this file name with file operators may not succeed for a variety of reasons, including:

- The category does not maintain all of its instances as named files.
- The operator tried to delete a file from a read-only file system.
- The operator tried to write to a file system with insufficient space.

There may be a limit on the length of a resource file name, which in turn imposes a length limit on the instance name. The inherent limit on resource instance names is the same as that on name objects in general (see Appendix B). By convention, font names are restricted to fewer than 40 characters. This convention is recommended for other resource categories as well. Note that the resource file name may be longer or shorter than the resource instance name, depending on details of the name-mapping algorithm. When calling **ResourceFileName**, it is prudent to provide a scratch string at least 100 characters long.

Some implementations provide additional control over the behavior of **ResourceFileName**; see Section C.3.6, "Resource File Location."

A resource file contains a PostScript program that can be executed to load the resource instance into VM. The last action the program should take is to execute **defineresource** or an equivalent operator, such as **definefont**, to associate the resource instance with a category and a name. In other words, each resource file must be self-identifying and self-defining. The resource file must be well behaved: it must leave the stacks in their original state and it must not execute any operators (graphics operators, for instance) that are not directly related to creating the resource instance.

For most resource categories, including **Generic**, the category's **FindResource** procedure executes true setglobal before executing the resource file and restores the previous VM allocation mode afterward. As a result, the resource instance is loaded into global VM and **defineresource** defines the resource instance globally, regardless of the VM allocation mode at the time **findresource** is invoked. Unfortunately, certain resource instances behave incorrectly if they reside in global VM. Some means are required to defeat the automatic loading into global VM. Two methods are currently used:

- Some implementations of the **Font** category's **FindResource** procedure omit executing true setglobal before executing the font file. This causes fonts to be defined in the VM allocation mode in effect when **findresource** is invoked, rather than always in global VM. Details of this policy are implementation-dependent.
- If a particular resource instance is known not to work in global VM, the resource file should begin with an explicit false setglobal.

A resource file can contain header comments, as specified in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*. If there is a header comment of the form

```
%%VMusage: int int
```

then the **resourcestatus** operator returns the larger of the two integers as its *size* result. If the %%VMusage: comment is not present, **resourcestatus** may not be able to determine the VM consumption for the resource instance, in which case it will return a size of –1.

The definition of an entire resource category—that is, an instance of the **Category** category—can come from a resource file in the normal way. If any resource operator is presented with an unknown category name, it automatically executes

```
category /Category findresource
```

in an attempt to cause the resource category to become defined. Only if that fails will the resource operator generate an **undefined** error to report that the resource category is unknown.

3.10 Functions

The PostScript language includes operators and procedures that take arguments off the operand stack and put their results back on the stack. The **add** operator, for example, pops two arguments, which must be numbers, and pushes the sum of those numbers back on the stack. **add** could be viewed as a function with two input values and one output value:

$$f(x_0, x_1) = x_0 + x_1$$

Similarly, the following procedure computes the average and the square root of the product of two numbers:

```
{   2 copy add
    2 div
    3 1 roll mul
    sqrt
}
```

This could be viewed as a function of two input values and two output values:

$$f(x_0, x_1) = \frac{x_0 + x_1}{2}, \sqrt{x_0 \times x_1}$$

In general, a function can take any number (m) of input values and produce any number (n) of output values:

$$f(x_0, \ldots, x_{m-1}) = y_0, \ldots, y_{n-1}$$

LanguageLevel 3 supports an explicit, static representation for functions, known as *function dictionaries*. Functions are less general than PostScript procedures: all the input values and all the output values are numbers, and functions have no side effects. On the other hand, functions can be considerably more efficient than procedures, since they entail no PostScript operator execution.

At present, there is only one use for functions in the PostScript language: they are used to define the color values in a shading pattern (see Section 4.9.3, "Shading Patterns," and the **shfill** operator in Chapter 8). There is no operator like **exec** that explicitly calls a function. Functions are also used extensively in PDF, where there are no procedures; for more information, see the *Portable Document Format Reference Manual.*

Each function definition includes a *domain*, the set of legal values for the input. Some types of function also define a *range*, the set of legal values for the output. Values passed to the function are clipped to the domain, and values produced by the function are clipped to the range. For example, suppose the function $f(x) = x + 2$ is defined with a domain of [−1 1]. If the function is called with the value 6, that value is replaced with the nearest value in the defined domain, 1, before the function is evaluated, and the result is therefore 3. Similarly, if the function $f(x_0, x_1) = 3 \times x_0 + x_1$ is defined with a range of [0 100], and if the values −6 and 4 are passed to the function (and are within its domain), then the value produced by the function, −14, is replaced with 0, the nearest value in the defined range.

3.10.1 Function Dictionaries

A function dictionary specifies a function's representation, the set of attributes that parameterize that representation, and the additional data needed by that representation. Three types of function are available, as indicated by the dictionary's **FunctionType** entry:

- *A sampled function* (type 0) uses a table of *sample values* to represent the function. Various techniques are used to interpolate values between the sample values.
- An *exponential interpolation function* (type 2) defines a set of coefficients for an exponential function.
- A *stitching function* (type 3) is a combination of other functions, partitioned across a domain.

All function dictionaries share the entries listed in Table 3.12. In addition, each type of function dictionary must include attributes appropriate to the particular function type. The number of output values can usually be inferred from other attributes of the function; if not (as is always the case for type 0 functions), the **Range** attribute is required. The dimensionality of the function implied by the **Domain** and **Range** attributes must be consistent with the dimensionality implied by other attributes of the function; otherwise, a **rangecheck** error will occur.

TABLE 3.12 Entries common to all function dictionaries

KEY	TYPE	VALUE
FunctionType	integer	*(Required)* The function type: 0 Sampled function 2 Exponential interpolation function 3 Stitching function
Domain	array	*(Required)* An array of $2 \times m$ numbers, where m is the number of input values. For each i from 0 to $m-1$, $\textbf{Domain}_{2i}$ must be less than or equal to $\textbf{Domain}_{2i+1}$, and the ith input value, x_i, must lie in the interval $\textbf{Domain}_{2i} \le x_i \le \textbf{Domain}_{2i+1}$. Input values outside the declared domain are clipped to the nearest boundary value.
Range	array	*(Required for type 0 functions, optional otherwise; see below)* An array of $2 \times n$ numbers, where n is the number of output values. For each j from 0 to $n-1$, $\textbf{Range}_{2j}$ must be less than or equal to $\textbf{Range}_{2j+1}$, and the jth output value, y_j, must lie in the interval $\textbf{Range}_{2j} \le y_j \le \textbf{Range}_{2j+1}$. Output values outside the declared range are clipped to the nearest boundary value. If the **Range** entry is absent, no clipping is done.

Type 0 Function Dictionaries (Sampled Functions)

Type 0 function dictionaries use a sequence of sample values to provide an approximation for functions whose domains and ranges are bounded. The samples are organized as an m-dimensional table in which each entry has n components.

Sampled functions are highly general and offer reasonably accurate representations of arbitrary analytic functions at low expense. For example, a 1-input sinusoidal function can be represented over the range [0 180] with an average error of only 1 percent, using just ten samples and linear interpolation. Two-input functions require significantly more samples, but usually not a prohibitive number, so long as the function does not have high frequency variations.

The dimensionality of a sampled function is restricted only by implementation limits. However, the number of samples required to represent high-dimensionality functions multiplies rapidly unless the sampling resolution is very low. Also, the process of multilinear interpolation becomes computationally intensive if m is greater than 2. The multidimensional spline interpolation is even more computationally intensive.

In addition to the entries in Table 3.12, a type 0 function dictionary includes the entries listed in Table 3.13.

TABLE 3.13 Additional entries specific to a type 0 function dictionary

KEY	TYPE	VALUE
Order	integer	*(Optional)* The order of interpolation between samples. Allowed values are 1 and 3, specifying linear and cubic spline interpolation, respectively. Default value: 1.
DataSource	string or file	*(Required)* A string or positionable file providing the sequence of sample values that specifies the function. (A file object derived from a **ReusableStreamDecode** filter may be used here.)
BitsPerSample	integer	*(Required)* The number of bits used to represent each component of each sample value. The number must be 1, 2, 4, 8, 12, 16, 24, or 32.
Encode	array	*(Optional)* An array of $2 \times m$ numbers specifying the linear mapping of input values into the domain of the function's sample table. Default value: [0 (**Size**$_0$ − 1) 0 (**Size**$_1$ − 1) …].
Decode	array	*(Optional)* An array of $2 \times n$ numbers specifying the linear mapping of sample values into the range of values appropriate for the function's output values. Default value: Same as the value of **Range**.
Size	array	*(Required)* An array of m positive integers specifying the number of samples in each input dimension of the sample table.

The **Domain**, **Encode**, and **Size** attributes determine how the function's input variable values are mapped into the sample table. For example, if **Size** is [21 31], the default **Encode** array is [0 20 0 30], which maps the entire domain into the full set of sample table entries. Other values of **Encode** may be used.

To explain the relationship between **Domain**, **Encode**, **Size**, **Decode**, and **Range**, we use the following notation:

$$y = \text{Interpolate}(x, x_{\min}, x_{\max}, y_{\min}, y_{\max}) = (x - x_{\min}) \times \frac{(y_{\max} - y_{\min})}{(x_{\max} - x_{\min})} + y_{\min}$$

For a given value of x, Interpolate calculates the y value on the line defined by the two points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$.

When a sampled function is called, each input value x_i, for $0 \le i < m$, is clipped to the domain:

$$x_i' = \min(\max(x_i, \textbf{Domain}_{2i}), \textbf{Domain}_{2i+1})$$

That value is encoded:

$$e_i = \text{Interpolate}(x_i', \textbf{Domain}_{2i}, \textbf{Domain}_{2i+1}, \textbf{Encode}_{2i}, \textbf{Encode}_{2i+1})$$

That value is clipped to the size of the sample table in that dimension:

$$e_i' = \min(\max(e_i, 0), \textbf{Size}_i - 1)$$

The encoded input values are real numbers, not restricted to integers. Interpolation is then used to determine output values from the nearest surrounding values in the sample table. Each output value r_j, for $0 \le j < n$, is then decoded:

$$r_j' = \text{Interpolate}(r_j, 0, 2^{\textbf{BitsPerSample}} - 1, \textbf{Decode}_{2j}, \textbf{Decode}_{2j+1})$$

Finally, each decoded value is clipped to the range:

$$y_j = \min(\max(r_j', \textbf{Range}_{2j}), \textbf{Range}_{2j+1})$$

Sample data is represented as a stream of unsigned 8-bit bytes (integers in the range 0 to 255). The bytes constitute a continuous bit stream, with the high-order bit of each byte first. Each sample value is represented as a sequence of **BitsPerSample** bits. Successive values are adjacent in the bit stream; there is no padding at byte boundaries.

For a function with multidimensional input (more than one input variable), the sample values in the first dimension vary fastest, and the values in the last dimension vary slowest. For example, for a function $f(a, b, c)$, where a, b, and c vary from 0 to 9 in steps of 1, the sample values would appear in this order: $f(0, 0, 0)$, $f(1, 0, 0)$, …, $f(9, 0, 0)$, $f(0, 1, 0)$, $f(1, 1, 0)$, …, $f(9, 1, 0)$, $f(0, 2, 0)$, $f(1, 2, 0)$, …, $f(9, 9, 0)$, $f(0, 0, 1)$, $f(1, 0, 1)$, and so on.

For a function with multidimensional output (more than one output value), the values are stored in the same order as **Range**.

The **DataSource** string or file must be long enough to contain the entire sample array, as indicated by **Size**, **Range**, and **BitsPerSample**; otherwise, a **rangecheck**

error will occur. If **DataSource** is a file, the sample data begins at file position 0. The operators that use the function will reposition this file at unpredictable times; a PostScript program should not attempt to access the same file. A **ReusableStreamDecode** filter is required if in-line data or a subfile is to be used as data for a sampled function.

Example 3.8 illustrates a sampled function with 4-bit samples in an array containing 21 columns and 31 rows. The function takes two arguments, *x* and *y*, in the domain [–1 1], and returns one value, *z*, in that same range.

Example 3.8

```
<< /FunctionType 0
   /Domain [–1  1  –1  1]
   /Size [21  31]
   /Encode [0  20  0  30]
   /BitsPerSample 4
   /Range [–1  1]
   /Decode [–1  1]
   /DataSource < … >
>>
```

The *x* argument is linearly transformed by the encoding to the domain [0 20] and the *y* argument to the domain [0 30]. Using bilinear interpolation between sample points, the function computes a value for *z*, which (because **BitsPerSample** is 4) will be in the range [0 15], and the decoding transforms *z* to a number in the range [–1 1] for the result. The sample array is stored in a string of 326 bytes, calculated as follows (rounded up):

$$326 \text{ bytes} = 31 \text{ rows} \times 21 \text{ samples/row} \times 4 \text{ bits/sample} \div 8 \text{ bits/byte}$$

The first byte contains the sample for the point (–1, –1) in the high-order 4 bits and the sample for the point (–0.9, –1) in the low-order 4 bits.

The **Decode** entry can be used creatively to increase the accuracy of encoded samples corresponding to certain values in the range. For example, if the desired range of the function is [–1 1] and **BitsPerSample** is 4, the usual value of **Decode** would be [–1 1] and the sample values would be integers in the interval [0 15] (as shown in Figure 3.1). But if these values were used, the midpoint of the range (0) would not be represented exactly by any sample value, since it would fall halfway between 7 and 8. On the other hand, if the **Decode** array were [–1 +1.1428571] (or more precisely, [–1 16 14 div]) and the sample values supplied were in the in-

terval [0 14], then the desired effective range of [−1 1] would be achieved, and the range value 0 would be represented by the sample value 7.

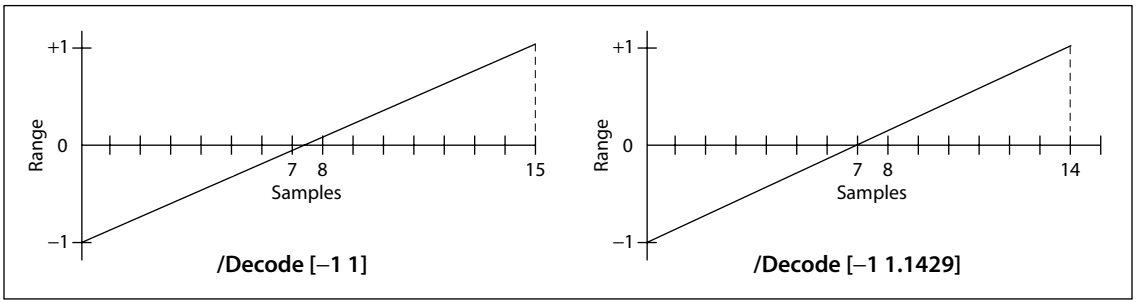


FIGURE 3.1 *Mapping with the **Decode** array*

The **Size** value for an input dimension can be 1, in which case all input values in that dimension will be mapped to the single allowed value. If **Size** is less than 4, cubic spline interpolation is not possible and **Order** 3 will be ignored if specified.

Type 2 Function Dictionary (Exponential Interpolation Functions)

Type 2 function dictionaries include a set of parameters that define an exponential interpolation of one input value and *n* output values:

$$f(x) = y_0, \ldots, y_{n-1}$$

In addition to the entries in Table 3.12 on page 108, a type 2 function dictionary includes the entries listed in Table 3.14.

Values of **Domain** must constrain x in such a way that if **N** is not an integer, all values of x must be greater than or equal to 0, and if **N** is negative, no value of x may be 0.

For typical use as an interpolation function, **Domain** will be declared as [0 1], and **N** will be a number greater than 0. The **Range** parameter is optional and can be used to clip the output to a desired range.

TABLE 3.14 Additional entries specific to a type 2 function dictionary

KEY	TYPE	VALUE
C0	array	*(Optional)* An array of n numbers defining the function result when $x = 0$ (hence the "0" in the name). Default value: [0].
C1	array	*(Optional)* An array of n numbers defining the function result when $x = 1$ (hence the "1" in the name). Default value: [1].
N	number	*(Required)* The interpolation exponent. Each input value x will return n values, given by $y_j = \mathbf{C0}_j + x^{\mathbf{N}} \times (\mathbf{C1}_j - \mathbf{C0}_j)$, for $0 \le j < n$.

Type 3 Function Dictionaries (Stitching Functions)

Type 3 function dictionaries define a "stitching" of the subdomains of several 1-input functions to produce a single new 1-input function. Since the resulting *stitching function* is a 1-input function, the domain is given by a two-element array, [$\mathbf{Domain}_0$ $\mathbf{Domain}_1$]. This domain is partitioned into k subdomains, as indicated by the dictionary's **Bounds** entry, which is an array of $k - 1$ numbers that obey the following inequality:

$$\mathbf{Domain}_0 < \mathbf{Bounds}_0 < \mathbf{Bounds}_1 < \ldots < \mathbf{Bounds}_{k-2} < \mathbf{Domain}_1$$

The value of the **Functions** entry is an array of k functions. The first function applies to x values in the first subdomain, $\mathbf{Domain}_0 \le x < \mathbf{Bounds}_0$; the second function applies to x values in the second subdomain, $\mathbf{Bounds}_0 \le x < \mathbf{Bounds}_1$; and so on. The last function applies to x values in the last subdomain, which includes the upper bound: $\mathbf{Bounds}_{k-2} \le x \le \mathbf{Domain}_1$.

The **Encode** array contains $2 \times k$ numbers. A value x from the ith subdomain is encoded as follows:

$$x' = \text{Interpolate}(x, \mathbf{Bounds}_{i-1}, \mathbf{Bounds}_i, \mathbf{Encode}_{2i}, \mathbf{Encode}_{2i+1})$$

for $0 \le i < k$. In this equation, $\mathbf{Bounds}_{-1}$ means $\mathbf{Domain}_0$, and $\mathbf{Bounds}_{k-1}$ means $\mathbf{Domain}_1$.

The value of k may be 1, in which case the **Bounds** array is empty and the single item in the **Functions** array applies to all x values, $\mathbf{Domain}_0 \le x \le \mathbf{Domain}_1$.

In addition to the entries in Table 3.12 on page 108, a type 3 function dictionary includes the entries listed in Table 3.15.

TABLE 3.15 Additional entries specific to a type 3 function dictionary

KEY	TYPE	VALUE
Functions	array	*(Required)* An array of k 1-input functions making up the stitching function. The output dimensionality of all functions must be the same, and compatible with the value of **Range** if **Range** is present.
Bounds	array	*(Required)* An array of $k - 1$ numbers that, in combination with **Domain**, define the intervals to which each function from the **Functions** array applies. **Bounds** elements must be in order of increasing value, and each value must be within the limits specified by **Domain**.
Encode	array	*(Required)* An array of $2 \times k$ numbers that, taken in pairs, map each subset of the domain defined by **Domain** and the **Bounds** array to the domain of the corresponding function.

Domain must be of size 2 (that is, $m = 1$). Note that $\textbf{Domain}_0$ must be strictly less than $\textbf{Domain}_1$ unless $k = 1$.

The stitching function is designed to make it easy to combine several functions to be used within one shading pattern, over different parts of the shading's domain. The same effect could be achieved by creating separate shading dictionaries for each of the functions, with adjacent domains. However, since each shading would have similar parameters, and because the overall effect is one shading, it is more convenient to have a single shading with multiple function definitions.

Also, function type 3 provides a general mechanism for inverting the domains of 1-input functions. For example, consider a function f with a **Domain** of [0 1], and a stitching function g with a **Domain** of [0 1], a **Functions** array containing f, and an **Encode** array of [1 0]. In effect, $g(x) = f(1 - x)$.

3.11 Errors

Various sorts of errors can occur during execution of a PostScript program. Some errors are detected by the PostScript interpreter, such as overflow of one of the interpreter's stacks. Others are detected during execution of the built-in operators, such as occurrence of the wrong type of operand.

Errors are handled in a uniform fashion that is under the control of the PostScript program. Each error is associated with a name, such as **stackoverflow** or **typecheck**. Each error name appears as a key in a special dictionary called **errordict** and is associated with a value that is the handler for that error. The complete set of error names appears in Section 8.1, "Operator Summary."

3.11.1 Error Initiation

When an error occurs, the interpreter does the following:

1. Restores the operand stack to the state it was in when it began executing the current object.
2. Pushes that object on the operand stack.
3. Looks up the error's name in **errordict** and executes the associated value, which is the *error handler* for that error.

This is everything the interpreter itself does in response to an error. The error handler in **errordict** is responsible for all other actions. A PostScript program can modify error behavior by defining its own error-handling procedures and associating them with the names in **errordict**.

The **interrupt** and **timeout** errors, which are initiated by events external to the PostScript interpreter, are treated specially. The interpreter merely executes **interrupt** or **timeout** from **errordict**, sandwiched between execution of two objects being interpreted in normal sequence. It does not push the object being executed, nor does it alter the operand stack in any other way. In other words, it omits steps 1 and 2 above.

3.11.2 Error Handling

The **errordict** dictionary present in the initial state of VM provides standard handlers for all errors. However, **errordict** is a writeable dictionary; a program can replace individual error handlers selectively. **errordict** is in local VM, so changes are subject to **save** and **restore**; see Section 3.7, "Memory Management."

The default error-handling procedures all operate in a standard way. They record information about the error in a special dictionary named **$error**, set the VM al-

location mode to local, and invoke the **stop** operator. They do not print anything or generate any text messages to %stdout or %stderr.

Execution of **stop** exits the innermost enclosing context established by the **stopped** operator. Assuming the user program has not invoked **stopped**, interpretation continues in the job server, which invoked the user program with **stopped**.

As part of error recovery, the job server executes the name **handleerror** from **errordict**. The default **handleerror** procedure accesses the error information in the **$error** dictionary and reports the error in an installation-dependent fashion. In some environments, **handleerror** simply writes a text message to the standard output file. In other environments, it invokes more elaborate error reporting mechanisms.

After an error occurs and one of the default error-handling procedures is executed, **$error** contains the entries shown in Table 3.16.

TABLE 3.16 Entries in the $error dictionary

KEY	TYPE	VALUE
newerror	boolean	A flag that is set to *true* to indicate that an error has occurred. **handleerror** sets it to *false*.
errorname	name	The name of the error that occurred.
command	any	The operator or other object being executed by the interpreter at the time the error occurred.
errorinfo	array or null	*(LanguageLevel 2)* If the error arose from an operator that takes a parameter dictionary as an operand (such as **setpagedevice** or **setdevparams**), this array contains the key and value of the incorrect parameter. (If a required entry was missing, this array contains the expected key with a null value.) **handleerror** sets **errorinfo** to *null*.
ostack	array	A snapshot of the entire operand stack immediately before the error, stored as if by the **astore** operator.
estack	array	A snapshot of the execution stack, stored as if by the **execstack** operator.
dstack	array	A snapshot of the dictionary stack, stored as if by the **dictstack** operator.

recordstacks	boolean	*(LanguageLevel 2)* A flag that controls whether the standard error handlers record the **ostack**, **estack**, and **dstack** snapshots. Default value: *true*.
binary	boolean	*(LanguageLevel 2)* A flag that controls the format of error reports produced by the standard **handleerror** procedure. *false* produces a text message; *true* produces a binary object sequence (see Section 3.14.6, "Structured Output"). Default value: *false*.

A program that wishes to modify the behavior of error handling can do so in one of two ways:

- It can change the way errors are *reported* simply by redefining **handleerror** in **errordict**. For example, a revised error handler might report more information about the context of the error, or it might produce a printed page containing the error information instead of reporting it to the standard output file.
- It can change the way errors are *invoked* by redefining the individual error names in **errordict**. There is no restriction on what an error-handling procedure can do. For example, in an interactive environment, an error handler might invoke a debugging facility that would enable the user to examine or alter the execution environment and perhaps resume execution.

3.12 Early Name Binding

Normally, when the PostScript language scanner encounters an executable name in the program being scanned, it simply produces an executable name object; it does not look up the value of the name. It looks up the name only when the name object is *executed* by the interpreter. The lookup occurs in the dictionaries that are on the dictionary stack at the time of execution.

A name object contained in a procedure is looked up each time the procedure is executed. For example, given the definition

```
/average {add 2 div} def
```

the names **add** and **div** are looked up, yielding operators to be executed, every time the **average** procedure is invoked.

This so-called *late binding* of names is an important feature of the PostScript language. However, there are situations in which *early binding* is advantageous.

There are two facilities for looking up the values of names before execution: the **bind** operator and the *immediately evaluated name*.

3.12.1 bind Operator

The **bind** operator takes a procedure operand and returns a possibly modified procedure. There are two kinds of modification: operator substitution and idiom recognition.

Operator Substitution

The **bind** operator first systematically replaces names with operators in a procedure. For each executable name whose value is an *operator* (not an array, procedure, or other type), it replaces the name with the operator object. This lookup occurs in the dictionaries that are on the dictionary stack at the time **bind** is executed. The effect of **bind** applies not only to the procedure being bound but to all subsidiary procedures (executable arrays or executable packed arrays) contained within it, nested to arbitrary depth.

When the interpreter subsequently executes this procedure, it encounters the *operator objects*, not the *names* of operators. For example, if the **average** procedure has been defined as

```
/average {add 2 div} bind def
```

then during the execution of **average**, the interpreter executes the **add** and **div** operators directly, without looking up the names **add** and **div**.

There are two main benefits to using **bind**:

- A procedure that has been bound will execute the sequence of operators that were intended when the procedure was defined, even if one or more of the operator names have been redefined in the meantime. This benefit is mainly of interest in procedures that are part of the PostScript implementation, such as **findfont** and =. Those procedures are expected to behave correctly and uniformly, regardless of how a user program may have altered its name environment.
- A bound procedure executes somewhat faster than one that has not been bound, since the interpreter need not look up the operator names each time,

but can execute the operators directly. This benefit is of interest in most PostScript programs, particularly in the prologs of page descriptions. It is worthwhile to apply **bind** to any procedure that will be executed more than a few times.

It is important to understand that **bind** replaces only those names whose values are *operators* at the time **bind** is executed. Names whose values are of other types, particularly procedures, are not disturbed. If an operator name has been redefined in some dictionary above **systemdict** on the dictionary stack *before* the execution of **bind**, occurrences of that name in the procedure will not be replaced.

Note: *Certain standard language features, such as* ***findfont****, are implemented as built-in procedures rather than as operators. Also, certain names, such as* ***true****,* ***false****, and* ***null****, are associated directly with literal values in* ***systemdict****. Occurrences of such names in a procedure are not altered by* ***bind****.*

Idiom Recognition

In LanguageLevel 3, the **bind** operator performs an additional task, known as *idiom recognition*, following the replacement of names in the bound procedure with operators. The goal of idiom recognition is to replace certain procedures ("idioms") with other procedures, typically ones that have equivalent behavior but produce better-quality results or execute more efficiently. Performing such substitution on procedures in an application's prolog can take advantage of new language features without changing the application.

The idioms and their replacements are stored as instances of the **IdiomSet** resource category. An **IdiomSet** instance is a *procedure substitution dictionary*, which typically contains idioms for a particular application's prolog. The keys in this dictionary are arbitrary. Each value in this dictionary is an array containing two procedures, a *template procedure* and a *substitute procedure*.

The **bind** operator first tests the value of the user parameter **IdiomRecognition** to see whether idiom recognition is enabled. If so, the bound procedure is compared to every template procedure in every **IdiomSet** instance. If a match is found, **bind** returns the associated substitute procedure; otherwise, it returns the bound procedure.

Two arrays or procedures are considered to match if corresponding elements either are equal (in the sense of the **eq** operator) or are both arrays whose corre-

sponding elements match in turn. The objects' attributes are disregarded during this comparison, just as they are by **eq**. Nested arrays or procedures are compared to a maximum depth of ten levels.

If substitutions may have an undesirable effect, idiom recognition can be disabled by setting the value of the user parameter **IdiomRecognition** to *false* before invoking the **bind** operator. For example, **IdiomRecognition** should be set to *false* during the construction of instances of the **IdiomSet** resource category, so that the template and substitute procedures are not themselves recognized as idioms.

Example 3.9 demonstrates how to construct an instance of the **IdiomSet** resource category.

Example 3.9

```
% Temporarily turn off idiom recognition so that bind does not change our template.
currentuserparams /IdiomRecognition get      % Save current value on stack
<</IdiomRecognition false>> setuserparams

% Define an IdiomSet resource named AdobeWinDriver containing a single substitution.
   /AdobeWinDriver
      <<    /snap                              % Name of this particular idiom (any name)
               [  % The template procedure.
                  % This is a common method in LanguageLevel 1 for aligning points
                  % on a grid in device space.
                  {   transform
                        0.25 sub round 0.25 add exch
                        0.25 sub round 0.25 add exch
                      itransform
                  } bind

                  % The substitute procedure.
                  % This procedure does not change the coordinates.
                  % Assume that setstrokeadjust is on.
                  { } bind
               ]
      >>
   /IdiomSet defineresource pop

<</IdiomRecognition  3  -1  roll>> setuserparams        % Return idiom recognition
                                                        % to its previous state

% If the restored value was true, bind will now replace occurrences of the template
% procedure with the substitute procedure.
```

The template and substitute procedures should be bound explicitly during the definition of the **IdiomSet** instance, since no automatic binding occurs on either of these procedures during idiom recognition. The comparison during idiom recognition occurs after the candidate procedure is bound; a successful match depends on the template also being bound. Generally, the substitute procedure should be bound, unless lookup of operator names during each execution of the substitute procedure is specifically desired.

Instances of the **IdiomSet** resource category reside in VM, either local or global; if local, they are subject to the **save** and **restore** operators. The **bind** operator follows the usual rules about visibility of resources according to the current VM allocation mode. That is, if the current VM allocation mode is global, only globally defined instances of **IdiomSet** are considered, whereas if the current allocation mode is local, both locally and globally defined instances are considered. Additionally, substitution will not occur if the candidate procedure is in global VM but the proposed substitute procedure is in local VM.

Multiple instances of the **IdiomSet** resource category may contain identical template procedures, but only one will be in effect when idiom recognition is enabled. The instance that takes precedence is not predictable.

As mentioned earlier, idiom recognition is performed by matching the template procedures in the **IdiomSet** resource instances. This is unlike all other resource categories, whose instances are selected according to their keys. This matching by value occurs only for **IdiomSet** instances that are defined in VM; **bind** does not consider instances that are not in VM but only in external storage.

To ensure that the instances in VM are consistent with the external ones, the interpreter automatically invokes **findresource** to load external **IdiomSet** instances into VM at the beginning of each job and at certain other times. If a PostScript program installs an external **IdiomSet** instance, it should then execute **undefineresource** to ensure that any existing instance of **IdiomSet** in VM with the same key is removed and replaced by the external instance.

3.12.2 Immediately Evaluated Names

LanguageLevels 2 and 3, as well as most LanguageLevel 1 implementations (see Appendix A), include a syntax feature called *immediately evaluated names*. When the PostScript language scanner encounters a token of the form *//name* (a name preceded by two slashes with no intervening spaces), it immediately looks up the

name and substitutes the corresponding value. This lookup occurs in the dictionaries on the dictionary stack at the time the scanner encounters the token. If it cannot find the name, an **undefined** error occurs.

The substitution occurs *immediately*—even inside an executable array delimited by { and }, where execution is deferred. Note that this process is a *substitution* and not an *execution*; that is, the name's value is not executed, but rather is substituted for the name itself, just as if the **load** operator were applied to the name.

The most common use of immediately evaluated names is to perform early binding of objects (other than operators) in procedure definitions. The **bind** operator, described in the previous section, performs early binding of operators; binding objects of other types requires the explicit use of immediately evaluated names.

Example 3.10 illustrates the use of an immediately evaluated name to bind a reference to a dictionary.

Example 3.10

```
/mydict << … >> def
/proc
   {   //mydict begin
           …
       end
   } bind def
```

In the definition of proc, //mydict is an immediately evaluated name. At the moment the scanner encounters the name, it substitutes the name's current value, which is the dictionary defined earlier in the example. The first element of the executable array proc is a dictionary object, not a name object. When proc is executed, it will access that dictionary, even if in the meantime mydict has been redefined or the definition has been removed.

Another use of immediately evaluated names is to refer directly to permanent objects: standard dictionaries, such as **systemdict**, and constant literal objects, such as the values of **true**, **false**, and **null**. On the other hand, it does not make sense to treat the names of variables as immediately evaluated names. Doing so would cause a procedure to be irrevocably bound to particular values of those variables.

A word of caution: Indiscriminate use of immediately evaluated names may change the behavior of a program. As discussed in Section 3.5, "Execution," the

behavior of a procedure differs depending on whether the interpreter encounters it directly or as the result of executing some other object (a name or an operator). Execution of the program fragments

```
{… b …}
{… //b …}
```

will have different effects if the value of the name b is a procedure. So it is inadvisable to treat the names of operators as immediately evaluated names. A program that does so will malfunction in an environment in which some operators have been redefined as procedures. This is why **bind** applies only to names whose values are operators, not procedures or other types.

3.13 Filtered Files Details

LanguageLevels 2 and 3 define a special kind of file called a *filter*, which reads or writes an underlying file and transforms the data in some way. Filters are introduced in Section 3.8.4, "Filters." This section describes the semantics of filters in more detail. It includes information about:

- The use of files, procedures, and strings as data sources and targets
- End-of-data conventions
- Details of individual filters
- Specifications of encoding algorithms for some filters

All features described in this section are LanguageLevel 2 features except for those labeled as LanguageLevel 3.

3.13.1 Data Sources and Targets

As stated in Section 3.8.4, "Filters," there are two kinds of filters: *decoding* filters and *encoding* filters. A decoding filter is an input file that reads from an underlying *data source* and produces transformed data as it is read. An encoding filter is an output file that takes the data written to it and writes transformed data to an underlying *data target*. Data sources and data targets may be files, procedures, or strings.

Files

A file is the most common data source or target for a filter. A file used as a data source must be an input file, and one used as a data target must be an output file; otherwise, an **invalidaccess** error occurs.

If a file is a data source for a decoding filter, the filter reads from it as necessary to satisfy demands on the filter, until either the filter reaches its end-of-data (EOD) condition or the data source reaches end-of-file. If a file is a data target for an encoding filter, the filter writes to it as necessary to dispose of data that has been written to the filter and transformed.

Closing a filter file does not close the underlying file, unless explicitly directed by the **CloseSource** or **CloseTarget** filter parameter *(LanguageLevel 3)*. A program typically creates a decoding filter to process data embedded in the program file itself—the one designated by **currentfile**. When the filter reaches EOD, execution of the underlying file resumes. Similarly, a program can embed the output of an encoding filter in the middle of an arbitrary data stream being written to the underlying output file.

Once a program has begun reading from or writing to a filter, it should not attempt to access the underlying file in any way until the filter has been closed. Doing so could interfere with the operation of the filter and leave the underlying file in an unpredictable state. However, it is safe to access the underlying file after execution of **filter** but before the first read or write of the filter file, except in certain uses of the **ReusableStreamDecode** filter. The method for establishing a filter pipeline in Example 3.5 on page 84 depends on this.

Procedures

The data source or target can be a procedure. When the filter file is read or written, it calls the procedure to obtain input data to be decoded or to dispose of output data that has been encoded. This enables the data to be supplied or consumed by an arbitrary program.

If a procedure is a data source, the filter calls it whenever it needs to obtain input data. The procedure must return on the operand stack a readable string containing any number of bytes of data. The filter pops this string from the stack and uses its contents as input to the filter. This process repeats until the filter encounters end-of-data (EOD). Any leftover data in the final string is discarded. The

procedure can return a string of length 0 to indicate that no more data is available.

If a procedure is a data target, the filter calls it whenever it needs to dispose of output data. Before calling the procedure, it pushes two operands on the stack: a string and a boolean flag. It expects the procedure to consume these operands and return a string. The filter calls the procedure in the following three situations:

- On the first write to the filter after the **filter** operator creates it, the filter calls the data target procedure with an empty string and the boolean value *true*. The procedure must return a writeable string of nonzero length, into which the filter can write filtered data.
- Whenever the filter needs to dispose of accumulated output data, it calls the procedure again, passing it a string containing the data and the boolean value *true*. This string is either the same string that was returned from the previous call or a substring of that string. The procedure must now do whatever is appropriate with the data, then return either the same string or another string into which the filter can write additional filtered data.
- When the filter file is closed, it calls the procedure a final time, passing it a string or substring containing the remaining output data, if any, and the boolean value *false*. The procedure must now do whatever is appropriate with the data and perform any required end-of-data actions, then return a string. Any string (including one of length 0) is acceptable. The filter does not use this string, but merely pops it off the stack.

It is normal for the data source or target procedure to return the same string each time. The string is allocated once at the beginning and serves simply as a buffer that is used repeatedly. Each time a data source procedure is called, it fills the string with one buffer's worth of data and returns it. Similarly, each time a data target procedure is called, it first disposes of any buffered data passed to it, then returns the original string for reuse.

Between successive calls to the data source or target procedure, a program should not do anything that would alter the contents of the string returned by that procedure. The filter reads or writes the string at unpredictable times, so altering it could disrupt the operation of the filter. If the string returned by the procedure is reclaimed by a **restore** operation before the filter becomes closed, the results are unpredictable. Typically, an **ioerror** occurs.

Note: *If a filter file object is reclaimed by* **restore** *or garbage collection before being closed, it is closed automatically; however, the data target procedure is not called.*

One use of procedures as data sources or targets is to run filters "backward." Filters are organized so that decoding filters are input files and encoding filters are output files. Normally, a PostScript program obtains encoded data from some external source, decodes it, and uses the decoded data; or it generates some data, encodes it, and sends it to some external destination. The organization of filters supports this model. However, if a program must provide the input to a decoding filter or consume the output of an encoding filter, it can do so by using procedures as data sources or targets.

Strings

If a string is a data source, the filter simply uses its contents as data to be decoded. If the filter encounters EOD, it ignores the remainder of the string. Otherwise, it continues until it has exhausted the string data. Until the filter is closed, the string should be treated as read-only. Writing into such a string will have unpredictable consequences for the data read from the filter.

If a string is a data target, the filter writes encoded data into it. This continues until the filter is closed. The contents of the string are not dependable until that time. If the filter exhausts the capacity of the string, an **ioerror** occurs. There is no way to determine how much data the filter has written into the string; if a program needs to know, it should use a procedure as the data target.

3.13.2 End-of-Data and End-of-File

A filter can reach a state in which it cannot continue filtering data. This is called the *end-of-data* (EOD) condition. Most decoding (input) filters can detect an EOD marker encoded in the data they are reading. The nature of this marker depends on the filter. Most encoding (output) filters append an EOD marker to the data they are writing. This generally occurs automatically when the filter file is closed. In a few instances, the EOD condition is based on predetermined information, such as a byte count or a line count, rather than on an explicit marker in the encoded data.

A file object, including a filter, can be closed at an arbitrary time, and a readable file can run out of data. This is called the *end-of-file* (EOF) condition. When a

decoding filter detects EOD and all the decoded data has been read, the filter reaches the EOF condition. The underlying data source or target for a filter can itself reach EOF. This usually results in the filter reaching EOF, perhaps after some delay.

For efficient operation, filters must be buffered. The PostScript interpreter automatically provides buffering as part of the filter file object. Because of the effects of buffering, the filter reads from its data source or writes to its data target at irregular times, not necessarily each time the filter file itself is read or written. Also, many filtering algorithms require an unpredictable amount of state to be held within the filter object.

Decoding Filters

Before encountering EOD, a decoding filter reads an unpredictable amount of data from its data source. However, when it encounters EOD, it stops reading from its data source. If the data source is a file, encoded data that is properly terminated by EOD can be followed by additional unencoded data, which a program can then read directly from that file.

When a filter reaches EOD and all the decoded data has been read from it, the filter file reaches EOF and is closed automatically. Automatic closing of input files at EOF is a standard feature of *all* file objects, not just of filters. (The **ReusableStreamDecode** filter is an exception; see "ReusableStreamDecode Filter" on page 153.) Unlike other file objects, a filter reaches EOF and is closed immediately after the *last* data character is read from it, rather than at the following attempt to read a character. A filter also reaches EOF if its data source runs out of data by reaching EOF.

Note: *Data for a filter must be terminated by an explicit EOD, even if the program reading from the filter (executing the* ***image*** *operator, for example) reads only the exact amount of data that is present.*

Applying **flushfile** to a decoding filter causes data to be drawn from the data source until the filter reaches EOD or the source runs out of data, whichever occurs first. This operator can be used to flush the remainder of the encoded data from the underlying file when the reading of filtered data must be terminated prematurely. After the **flushfile** operation, the underlying file is positioned so that the next read from that file will begin immediately following the EOD of the encoded data. If a program closes a decoding filter prematurely before it reaches

EOD and *without* explicitly flushing it, the data source will be in an indeterminate state. Because of buffering, there is no dependable way to predict how much data will have been consumed from the data source.

Encoding Filters

As stated earlier, writing to an encoding (output) filter causes it to write encoded data to its data target. However, because of the effects of buffering, the writes to the data target occur at unpredictable times. The only way to ensure that all encoded data has been written is to close the filter.

Most encoding filters can accept an indefinite amount of data to be encoded. The amount usually is not specified in advance. Closing the filter causes an EOD marker to be written to the data target at the end of the encoded data. The nature of the EOD marker depends on the filter being used; it is sometimes under the control of parameters specified when the filter is created.

The **DCTEncode** filter requires the amount of data to be specified in advance, when the filter is created. When that amount of data has been encoded, the filter reaches the EOD condition automatically. Attempting to write additional data to the filter causes an **ioerror**, possibly after some delay.

Some data targets can become unable to accept further data. For instance, if the data target is a string, the string may become full. If the data target is a file, the file may become closed. Attempting to write to a filter whose data target cannot accept data causes an **ioerror**.

Applying **flushfile** to an encoding filter file causes the filter to flush buffered data to its data target to the extent possible. If the data target is a file, **flushfile** is also invoked for it. The effect of **flushfile** will propagate all the way down a filter pipeline. However, because of the nature of filter algorithms, it is not possible to guarantee that all data stored as part of a filter's internal state will be flushed.

On the other hand, applying **closefile** to an encoding filter flushes both the buffered data and the filter's internal state. This causes all encoded data to be written to the data target, followed by an EOD marker, if appropriate.

When a program closes a pipeline consisting of two or more encoding filters, it must close each component filter file in sequence, starting with the one that was

created last (in other words, the one farthest upstream). This ensures that all buffered data and all appropriate EOD markers are written in the proper order.

If a filter file object is reclaimed by **restore** or garbage collection before being closed, it is closed automatically (as is the case for all file objects); however, no attempt is made to close a filter pipeline in the correct order. Errors arising from closing in the wrong order are ignored, and filter target procedures are not called.

CloseSource and CloseTarget

CloseSource and **CloseTarget** *(both LanguageLevel 3)* are optional boolean parameters in the parameter dictionary for decoding and encoding filters, respectively. These parameters govern the disposition of the filter's data source or target when the **closefile** operator is applied to the filter explicitly, or implicitly in one of the following ways: by the **restore** operator, by garbage collection, or (except for the **ReusableStreamDecode** filter) by reaching EOD.

If **CloseSource** or **CloseTarget** is *false* (as they are by default), no additional action is taken on the data source or target; this is the behavior in LanguageLevel 2. However, if the parameter is *true*, then after **closefile** has been applied to the filter, it is also applied to the filter's data source or target. This process propagates through an entire pipeline, unless a filter is reached whose **CloseSource** or **CloseTarget** parameter is *false*; that filter is closed, but its source or target is not.

3.13.3 Details of Individual Filters

As stated in Section 3.8.4, "Filters," the PostScript language supports three categories of standard filters: ASCII encoding and decoding filters, compression and decompression filters, and subfile filters. The following sections document the individual filters.

Some of the encoded formats these filters support are the same as or similar to those supported by applications or utility programs on many computer systems. It should be straightforward to make those programs compatible with the filters. Also, C language implementations of some filters are available from the Adobe Developers Association.

ASCIIHexDecode Filter

source /ASCIIHexDecode filter
source dictionary /ASCIIHexDecode filter

The **ASCIIHexDecode** filter decodes data encoded as ASCII hexadecimal and produces binary data. For each pair of ASCII hexadecimal digits (0–9 and either A–F or a–f), it produces one byte of binary data. All white-space characters—space, tab, carriage return, line feed, form feed, and null—are ignored. The character > indicates EOD. Any other characters will cause an **ioerror**.

If the filter encounters EOD when it has read an odd number of hexadecimal digits, it will behave as if it had read an additional 0 digit.

The parameter dictionary can be used to specify the **CloseSource** parameter *(LanguageLevel 3)*.

ASCIIHexEncode Filter

target /ASCIIHexEncode filter
target dictionary /ASCIIHexEncode filter

The **ASCIIHexEncode** filter encodes binary data as ASCII hexadecimal. For each byte of binary data, it produces two ASCII hexadecimal digits (0–9 and either A–F or a–f). It inserts a newline in the encoded output at least once every 80 characters, thereby limiting the lengths of lines.

When the **ASCIIHexEncode** filter is closed, it writes a > character as an EOD marker.

The parameter dictionary can be used to specify the **CloseTarget** parameter *(LanguageLevel 3)*.

ASCII85Decode Filter

source /ASCII85Decode filter
source dictionary /ASCII85Decode filter

The **ASCII85Decode** filter decodes data encoded in the ASCII base-85 encoding format and produces binary data. See the description of the **ASCII85Encode** filter for a definition of the ASCII base-85 encoding.

The ASCII base-85 data format uses the characters ! through u and the character z. All white-space characters—space, tab, carriage return, line feed, form feed, and null—are ignored. If the **ASCII85Decode** filter encounters the character ~ in its input, the next character must be > and the filter will reach EOD. Any other characters will cause the filter to issue an **ioerror**. Also, any character sequences that represent impossible combinations in the ASCII base-85 encoding will cause an **ioerror**.

The parameter dictionary can be used to specify the **CloseSource** parameter *(LanguageLevel 3)*.

ASCII85Encode Filter

target /ASCII85Encode filter
target dictionary /ASCII85Encode filter

The **ASCII85Encode** filter encodes binary data in the ASCII base-85 encoding. Generally, for every 4 bytes of binary data, it produces 5 ASCII printing characters in the range ! through u. It inserts a newline in the encoded output at least once every 80 characters, thereby limiting the lengths of lines.

When the **ASCII85Encode** filter is closed, it writes the 2-character sequence ~> as an EOD marker.

Binary data bytes are encoded in 4-tuples (groups of 4). Each 4-tuple is used to produce a 5-tuple of ASCII characters. If the binary 4-tuple is $(b_1\ b_2\ b_3\ b_4)$ and the encoded 5-tuple is $(c_1\ c_2\ c_3\ c_4\ c_5)$, then the relation between them is

$$(b_1 \times 256^3) + (b_2 \times 256^2) + (b_3 \times 256^1) + b_4 =$$
$$(c_1 \times 85^4) + (c_2 \times 85^3) + (c_3 \times 85^2) + (c_4 \times 85^1) + c_5$$

In other words, 4 bytes of binary data are interpreted as a base-256 number and then converted into a base-85 number. The five "digits" of this number, $(c_1\ c_2\ c_3\ c_4\ c_5)$, are then converted into ASCII characters by adding 33, which is the ASCII code for !, to each. ASCII characters in the range ! to u are used, where ! represents the value 0 and u represents the value 84. As a special case, if all five digits are 0, they are represented by a single character z instead of by !!!!!.

If the **ASCII85Encode** filter is closed when the number of characters written to it is not a multiple of 4, it uses the characters of the last, partial 4-tuple to produce a last, partial 5-tuple of output. Given n (1, 2, or 3) bytes of binary data, it first appends $4 - n$ zero bytes to make a complete 4-tuple. Then, it encodes the 4-tuple in the usual way, but without applying the z special case. Finally, it writes the first $n + 1$ bytes of the resulting 5-tuple. Those bytes are followed immediately by the ~> EOD marker. This information is sufficient to correctly encode the number of final bytes and the values of those bytes.

The following conditions constitute encoding violations:

- The value represented by a 5-tuple is greater than $2^{32} - 1$.
- A z character occurs in the middle of a 5-tuple.
- A final partial 5-tuple contains only one character.

These conditions never occur in the output produced by the **ASCII85Encode** filter. Their occurrence in the input to the **ASCII85Decode** filter causes an **ioerror**.

The ASCII base-85 encoding is similar to one used by the public domain utilities *btoa* and *atob*, which are widely available on workstations. However, it is not exactly the same; in particular, it omits the begin-data and end-data marker lines, and it uses a different convention for marking end-of-data.

The parameter dictionary can be used to specify the **CloseTarget** parameter *(LanguageLevel 3)*.

LZWDecode Filter

source /LZWDecode filter
source dictionary /LZWDecode filter

The **LZWDecode** filter decodes data that is encoded in a Lempel-Ziv-Welch compressed format. See the description of the **LZWEncode** filter for details of the format and a description of the filter parameters.

LZWEncode Filter

target /LZWEncode filter
target dictionary /LZWEncode filter

The **LZWEncode** filter encodes ASCII or binary data according to the basic LZW (Lempel-Ziv-Welch) data compression method. LZW is a variable-length, adaptive compression method that has been adopted as one of the standard compression methods in the *tag image file format* (TIFF) standard. The output produced by the **LZWEncode** filter is always binary, even if the input is ASCII text.

LZW compression can discover and exploit many patterns in its input data. In its basic form, it is especially well suited to natural-language and PostScript-language text. The filter also supports optional pretransformation by a predictor function, as described in the section "Predictor Functions" on page 139; this improves compression of sampled image data.

Note: *The LZW compression method is the subject of United States patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Adobe Systems has licensed this patent for use in its products. Independent software vendors (ISVs) may be required to license this patent to develop software using the LZW method to compress PostScript programs or data for use with Adobe products. Unisys has agreed that ISVs may obtain such a license for a modest one-time fee. Additional information can be obtained on the World Wide Web at <http://www.unisys.com/LeadStory/lzwfaq.html>.*

An **LZWDecode** or **LZWEncode** parameter dictionary may contain any of the entries listed in Table 3.17. Unless otherwise noted, a decoding filter's parameters must match the parameters used by the encoding filter that generated its input data.

TABLE 3.17 Entries in an LZWEncode or LZWDecode parameter dictionary

KEY	TYPE	VALUE
EarlyChange	integer	*(Optional)* A code indicating when to increase the code word length. The TIFF specification can be interpreted to imply that code word length increases are postponed as long as possible. However, some existing implementations of LZW increase the code word length one code word earlier than necessary. The PostScript language supports both interpretations. If **EarlyChange** is 0, code word length increases are postponed as long as possible. If it is 1, they occur one code word early. Default value: 1.

UnitLength	integer	*(Optional; LanguageLevel 3)* The size of the units encoded, in bits. The allowed values are 3 through 8. See "UnitLength and LowBitFirst" on page 136. Default value: 8. A value other than the default is permitted only for **LZWDecode** and should not be used in combination with a predictor (specified by a **Predictor** value greater than 1; see Table 3.20).
LowBitFirst	boolean	*(Optional; LanguageLevel 3)* A flag that determines whether the code words are packed into the encoded data stream low-order bit first (*true*) or high-order bit first (*false*). See "UnitLength and LowBitFirst" on page 136. Default value: *false*. A value other than the default is permitted only for **LZWDecode**.
Predictor	integer	*(Optional)* See Table 3.20 on page 141.
Columns	integer	*(Optional)* See Table 3.20 on page 141.
Colors	integer	*(Optional)* See Table 3.20 on page 141.
BitsPerComponent	integer	*(Optional)* See Table 3.20 on page 141.
CloseSource	boolean	*(Optional; LanguageLevel 3; **LZWDecode** only)* A flag specifying whether closing the filter should also close its data source. Default value: *false*.
CloseTarget	boolean	*(Optional; LanguageLevel 3; **LZWEncode** only)* A flag specifying whether closing the filter should also close its data target. Default value: *false*.

In LanguageLevel 3, the size of the units encoded is determined by the optional **UnitLength** entry in the **LZWDecode** parameter dictionary; its default value is 8. The following general discussion of the encoding scheme refers to this LanguageLevel 3 parameter; for LanguageLevel 2, assume a unit size of 8.

The encoded data consists of a sequence of codes that can be from (**UnitLength** + 1) to a maximum of 12 bits long. Each code denotes a single character of input data (0 to $2^{\textbf{UnitLength}} - 1$), a clear-table marker ($2^{\textbf{UnitLength}}$), an EOD marker ($2^{\textbf{UnitLength}} + 1$), or a table entry representing a multicharacter sequence that has been encountered previously in the input ($2^{\textbf{UnitLength}} + 2$ and greater). In the normal case where **UnitLength** is 8, the clear-table marker is 256 and the EOD marker is 257.

Initially, the code length is (**UnitLength** + 1) bits and the table contains only entries for the ($2^{\textbf{UnitLength}} + 2$) fixed codes. As encoding proceeds, entries are appended to the table, associating new codes with longer and longer input character sequences. The encoding and decoding filters maintain identical copies of this table.

Whenever both the encoder and decoder independently (but synchronously) realize that the current code length is no longer sufficient to represent the number of entries in the table, they increase the number of bits per code by 1. For a **UnitLength** of 8, the first output code that is 10 bits long is the one following the creation of table entry 511, and so on for 11 (1023) and 12 (2047) bits. Codes are never longer than 12 bits, so entry 4095 is the last entry of the LZW table.

The encoder executes the following sequence of steps to generate each output code:

1. Accumulate a sequence of one or more input characters matching some sequence already present in the table. For maximum compression, the encoder should find the longest such sequence.
2. Emit the code corresponding to that sequence.
3. Create a new table entry for the first unused code. Its value is the sequence found in step 1 followed by the *next* input character.

For example, suppose **UnitLength** is 8 and the input consists of the following sequence of ASCII character codes:

```
45 45 45 45 45 65 45 45 45 66
```

Starting with an empty table, the encoder proceeds as shown in Table 3.18.

TABLE 3.18 Typical LZW encoding sequence

INPUT SEQUENCE	OUTPUT CODE	CODE ADDED TO TABLE	SEQUENCE REPRESENTED BY NEW CODE
–	256 (clear-table)	–	–
45	45	258	45 45
45 45	258	259	45 45 45
45 45	258	260	45 45 65
65	65	261	65 45
45 45 45	259	262	45 45 45 66
–	257 (EOD)	–	–

Codes are packed into a continuous bit stream, high-order bit first (assuming that **LowBitFirst** is *false*). This stream is then divided into 8-bit bytes, high-order bit first. Thus, codes can straddle byte boundaries arbitrarily. After the EOD marker (code value of 257), any leftover bits in the final byte are set to 0.

In the example above, all the output codes are 9 bits long; they would pack into bytes as follows (represented in hexadecimal):

```
80 0B 60 50 22 0C 0E 02
```

To adapt to changing input sequences, the encoder may at any point issue a clear-table code, which causes both the encoder and decoder to restart with initial tables and 9-bit codes. By convention, the encoder begins by issuing a clear-table code. It must issue a clear-table code when the table becomes full; it may do so sooner.

UnitLength and LowBitFirst

As indicated earlier, the default value of **UnitLength** is 8 and of **LowBitFirst** is *false*. These are the only values supported in LanguageLevel 2. Moreover, even in LanguageLevel 3, values other than the default are permitted only for **LZWDecode**, not for **LZWEncode**. This support is provided as a convenience for decoding images from other sources (principally GIF files) that use representations other than the default. The default values are recommended for general document interchange.

Data that has been LZW-encoded with a **UnitLength** less than 8 consists only of codes in the range 0 to $2^{\textbf{UnitLength}} - 1$; consequently, the **LZWDecode** filter produces only codes in that range when read. **UnitLength** also affects the encoded representation, as described above.

LZW is a bit-stream protocol, and the codes of compressed data do not necessarily fall on byte boundaries. The **LowBitFirst** parameter controls how these codes get packed into a byte stream.

- If **LowBitFirst** is *false* (the default), codes are packed into bytes high-order bit first. That is, bits of a code are stored into the available bits of a byte starting with the highest-order bit. When a code straddles a byte boundary, the high-order portion of the code appears in the low-order bits of one byte; the low-

order portion of the code appears in the high-order bits of the next byte. For example, the sequence of 9-bit output codes in Table 3.18 is encoded as

80 0B 60 50 22 0C 0E 02

- If **LowBitFirst** is *true*, codes are packed into bytes low-order bit first. That is, bits of a code are stored into the available bits of a byte starting with the lowest-order bit. When a code straddles a byte boundary, the low-order portion of the code appears in the high-order bits of one byte; the high-order portion of the code appears in the low-order bits of the next byte. For example, the sequence of 9-bit output codes in Table 3.18 would be encoded as

  00 5B 08 14 18 64 60 40

FlateDecode Filter

source /FlateDecode filter
source dictionary /FlateDecode filter

The **FlateDecode** filter *(LanguageLevel 3)* decodes data encoded in zlib/deflate compressed format. See the description of the **FlateEncode** filter for details of the format.

FlateEncode Filter

target /FlateEncode filter
target dictionary /FlateEncode filter

The **FlateEncode** filter *(LanguageLevel 3)* encodes ASCII or binary data. Encoding is based on the public-domain zlib/deflate compression method, which is a variable-length Lempel-Ziv adaptive compression method cascaded with adaptive Huffman coding. This method is referred to below as the Flate method. It is fully defined in Internet Engineering Task Force Requests for Comments (IETF RFCs) 1950 and 1951. The output produced by the **FlateEncode** filter is always binary, even if the input is ASCII text.

Flate compression can discover and exploit many patterns in its input data. In its basic form, it is especially well suited to natural-language and PostScript-language text. The filter also supports optional pretransformation by a predictor function, as described in the section "Predictor Functions" on page 139; this improves compression of sampled image data.

A **FlateDecode** or **FlateEncode** parameter dictionary may contain any of the entries listed in Table 3.19. Unless otherwise noted, a decoding filter's parameters must match the parameters used by the encoding filter that generated its input data.

TABLE 3.19 Entries in a FlateEncode or FlateDecode parameter dictionary

KEY	TYPE	VALUE
Effort	integer	*(Optional;* ***FlateEncode*** *only)* A code controlling the amount of memory used and the execution speed for Flate compression. Allowed values range from −1 to 9. A value of 0 compresses rapidly but not tightly, using little auxiliary memory. A value of 9 compresses slowly but as tightly as possible, using a large amount of auxiliary memory. A value of −1 is mapped to a value within the range 0 to 9 that is a "reasonable" default for the implementation. Default value: −1.
Predictor	integer	*(Optional)* See Table 3.20 on page 141.
Columns	integer	*(Optional)* See Table 3.20 on page 141.
Colors	integer	*(Optional)* See Table 3.20 on page 141.
BitsPerComponent	integer	*(Optional)* See Table 3.20 on page 141.
CloseSource	boolean	*(Optional;* ***FlateDecode*** *only)* A flag specifying whether closing the filter should also close its data source. Default value: *false*.
CloseTarget	boolean	*(Optional;* ***FlateEncode*** *only)* A flag specifying whether closing the filter should also close its data target. Default value: *false*.

Comparison of LZW and Flate Encoding

Flate encoding, like LZW encoding, discovers and exploits many patterns in its input data, whether text or images. Thanks to its cascaded adaptive Huffman coding, Flate-encoded output is usually substantially more compact than LZW-encoded output for the same input. Flate and LZW decoding speeds are comparable, but Flate encoding is considerably slower than LZW encoding.

Usually, the **FlateEncode** and **LZWEncode** filters compress their inputs substantially. In the worst case, however, the **FlateEncode** filter *expands* its input by no more than a factor of 1.003, plus the effects of algorithm tags added by PNG predictors (described below) and the effects of any explicit **flushfile** operations. LZW

compression has a worst-case expansion of at least a factor of 1.125, which can increase to nearly 1.5 in some implementations (plus the effects of PNG tags).

Predictor Functions

LZWEncode and **FlateEncode** filters compress more compactly if their input data is highly predictable. One way to increase the predictability of many continuous-tone sampled images is to replace each sample with the difference between that sample and some predictor function applied to earlier neighboring samples. If the predictor function works well, the postprediction data will cluster toward 0.

The parameter dictionary for the LZW and Flate filters may contain any of the four entries **Predictor**, **Columns**, **Colors**, and **BitsPerComponent** to specify a predictor function. When a predictor is selected, the encoding filter applies the predictor function to the data before compression; the decoding filter applies the complementary predictor function after decompression. Unless otherwise noted, a decoding filter's parameters must match the parameters used by the encoding filter that generated its input data.

Two groups of predictor functions are supported. The first, the TIFF group, consists of the single function that is Predictor 2 in the TIFF standard. (In the TIFF standard, Predictor 2 applies only to LZW compression, but here it applies to Flate compression as well.) TIFF Predictor 2 predicts that each color component of a sample will be the same as the corresponding color component of the sample immediately to the left.

The second supported group of predictor functions, the PNG group, consists of the "filters" of the World Wide Web Consortium's Portable Network Graphics recommendation, documented in IETF RFC 2083. The term *predictors* is used here instead of *filters* to avoid confusion. There are five basic PNG predictor algorithms:

None	No prediction
Sub	Predicts the same as the sample to the left
Up	Predicts the same as the sample above
Average	Predicts the average of the sample to the left and the sample above
Paeth	A nonlinear function of the sample above, the sample to the left, and the sample to the upper-left

The two groups of predictor functions have some common features. Both assume the following:

- Data is presented in order, from the top row to the bottom row and from left to right within a row.
- A row occupies a whole number of bytes, rounded up if necessary.
- Samples and their components are packed into bytes from high- to low-order bits.
- All color components of samples outside the image (which are necessary for predictions near the boundaries) are 0.

The two groups differ in the following ways:

- With PNG predictors, the encoded data explicitly identifies the predictor function used for each row, so different rows can be predicted with different algorithms to improve compression. The TIFF predictor has no such identifier; the same algorithm applies to all rows.
- The TIFF function group predicts each color component from the prior instance of that color component, taking into account the bits per component and the number of components per sample. In contrast, the PNG function group predicts each byte of data as a function of the corresponding byte of one or more previous image samples, regardless of whether there are multiple color components in a byte, or whether a single color component spans multiple bytes. This can yield significantly better speed but with somewhat worse compression.

Table 3.20 describes the predictor-related entries in a parameter dictionary for an LZW or Flate filter.

TABLE 3.20 Predictor-related entries in an LZW or Flate filter parameter dictionary

KEY	TYPE	VALUE
Predictor	integer	*(Optional)* A code that selects the predictor function: 1 No predictor (normal encoding or decoding) 2 TIFF Predictor 2 ≥10 (*LanguageLevel 3*) PNG predictor. For **LZWEncode** and **FlateEncode**, this selects the specific PNG predictor function(s) to be used, as indicated below. For **LZWDecode** and **FlateDecode**, any of these values merely indicates that PNG predictors are in use; the predictor function is explicitly encoded in the incoming data. The values of **Predictor** for the encoding and decoding filters need not match if they are both greater than or equal to 10. 10 PNG predictor, None function 11 PNG predictor, Sub function 12 PNG predictor, Up function 13 PNG predictor, Average function 14 PNG predictor, Paeth function 15 PNG predictor in which the encoding filter automatically chooses the optimum function separately for each row Default value: 1.
Columns	integer	*(Optional; used only if **Predictor** is greater than 1)* The number of samples in each row. Default value: 1.
Colors	integer	*(Optional; used only if **Predictor** is greater than 1)* The number of interleaved color components per sample; must be 1 or greater. Default value: 1.
BitsPerComponent	integer	*(Optional; used only if **Predictor** is greater than 1)* The number of bits used to represent each color component of a sample; must be 1, 2, 4, or 8. Default value: 8.

RunLengthDecode Filter

source /RunLengthDecode filter
source dictionary /RunLengthDecode filter

The **RunLengthDecode** filter decodes data encoded in the run-length encoding format. The encoded data consist of pairs of run-length bytes and data. See the description of the **RunLengthEncode** filter for details of the format. A run length of 128 indicates EOD.

The parameter dictionary may be used to specify the **CloseSource** parameter *(LanguageLevel 3)*.

RunLengthEncode Filter

target recordsize /RunLengthEncode filter
target dictionary recordsize /RunLengthEncode filter

The **RunLengthEncode** filter encodes data in a simple byte-oriented format based on run length. The compressed data format is a sequence of runs, where each run consists of a *length* byte followed by 1 to 128 bytes of data. If the *length* byte is in the range 0 to 127, the following *length* + 1 bytes (1 to 128 bytes) are to be copied literally upon decompression. If *length* is in the range 129 to 255, the following single byte is to be replicated 257 – *length* times (2 to 128 times) upon decompression.

When the **RunLengthEncode** filter is closed, it writes a final byte, with value 128 as an EOD marker.

recordsize is a nonnegative integer specifying the number of bytes in a "record" of source data. The **RunLengthEncode** filter will not create a run that contains data from more than one source record. If *recordsize* is 0, the filter does not treat its source data as records. The notion of a "record" is irrelevant in the context of the PostScript interpreter (in particular, the **image** operator does not require its data to be divided into records). A nonzero *recordsize* is useful only if the encoded data is to be sent to some application program that requires it.

This encoding is very similar to that used by the Apple® Macintosh® PackBits routine and by TIFF Data Compression scheme #32773. Output from PackBits is acceptable as input to the **RunLengthDecode** filter if an EOD marker (byte value 128) is appended to it. Output from the **RunLengthEncode** filter is acceptable to

UnpackBits if the *recordsize* parameter is equal to the length of one scan line for the image being encoded.

The parameter dictionary can be used to specify the **CloseTarget** parameter *(LanguageLevel 3)*. Note that there is no means for specifying *recordsize* in the parameter dictionary; it must be an explicit operand of the **RunLengthEncode** filter.

CCITTFaxDecode Filter

source /CCITTFaxDecode filter
source dictionary /CCITTFaxDecode filter

The **CCITTFaxDecode** filter decodes image data that has been encoded according to the CCITT facsimile standard. See the description of the **CCITTFaxEncode** filter for details of the filter parameters.

If the **CCITTFaxDecode** filter encounters improperly encoded source data, it will issue an **ioerror**. It will not perform any error correction or resynchronization, except as noted for **DamagedRowsBeforeError** in Table 3.21.

CCITTFaxEncode Filter

target /CCITTFaxEncode filter
target dictionary /CCITTFaxEncode filter

The **CCITTFaxEncode** filter encodes image data according to the CCITT facsimile (fax) standard. This encoding is defined by an international standards organization, the International Telecommunication Union (ITU), formerly known as the Comité Consultatif International Téléphonique et Télégraphique (International Coordinating Committee for Telephony and Telegraphy). The encoding is designed to achieve efficient compression of monochrome (1 bit per pixel) image data at relatively low resolutions. The encoding algorithm is not described in this book, but rather in the ITU standard (see the Bibliography). We refer to that standard as the CCITT standard for historical reasons.

Note: *PostScript language support for the CCITT standard is limited to encoding and decoding of image data. It does not include initial connection and handshaking protocols that would be required to communicate with a fax machine. The purpose of these filters is to enable efficient interchange of bilevel sampled images between an application program and a PostScript interpreter.*

The **CCITTFaxDecode** and **CCITTFaxEncode** filters support two encoding schemes, Group 3 and Group 4, and various optional features of the CCITT standard. Table 3.21 describes the contents of the parameter dictionary for these filters.

TABLE 3.21 Entries in a CCITTFaxEncode or CCITTFaxDecode parameter dictionary

KEY	TYPE	VALUE
Uncompressed	boolean	*(Optional)* A flag indicating whether the **CCITTFaxEncode** filter is permitted to use uncompressed encoding when advantageous. Uncompressed encoding is an optional part of the CCITT fax encoding standard. Its use can prevent significant data expansion when encoding certain image data, but many fax machine manufacturers and software vendors do not support it. The **CCITTFaxDecode** filter always accepts uncompressed encoding. Default value: *false*.
K	integer	*(Optional)* An integer that selects the encoding scheme to be used: <0 Pure two-dimensional encoding (Group 4) 0 Pure one-dimensional encoding (Group 3, 1-D) >0 Mixed one- and two-dimensional encoding (Group 3, 2-D), in which a line encoded one-dimensionally can be followed by at most **K** – 1 lines encoded two-dimensionally The **CCITTFaxEncode** filter uses the value of **K** to determine how to encode the data. The **CCITTFaxDecode** filter distinguishes among negative, zero, and positive values of **K** to determine how to interpret the encoded data; however, it does not distinguish between different positive **K** values. Default value: 0.
EndOfLine	boolean	*(Optional)* A flag indicating whether the **CCITTFaxEncode** filter prefixes an end-of-line bit pattern to each line of encoded data. The **CCITTFaxDecode** filter always accepts end-of-line bit patterns, but requires them to be present only if **EndOfLine** is *true*. Default value: *false*.
EncodedByteAlign	boolean	*(Optional)* A flag indicating whether the **CCITTFaxEncode** filter inserts extra 0 bits before each encoded line so that the line begins on a byte boundary. If *true*, the **CCITTFaxDecode** filter skips over encoded bits to begin decoding each line at a byte boundary. If *false*, the filters neither generate nor expect extra bits in the encoded representation. Default value: *false*.
Columns	integer	*(Optional)* The width of the image in pixels. If **Columns** is not a multiple of 8, the filters adjust the width of the unencoded image to the next multiple of 8. This adjustment is necessary for consistency with the

		image operator, which requires that each line of source data start on a byte boundary. Default value: 1728.
Rows	integer	*(Optional; **CCITTFaxDecode** only)* The height of the image in scan lines. If **Rows** is 0 or absent, the image's height is not predetermined; the encoded data must be terminated by an end-of-block bit pattern or by the end of the filter's data source. Default value: 0.
EndOfBlock	boolean	*(Optional)* A flag indicating whether the **CCITTFaxEncode** filter appends an end-of-block pattern to the encoded data. If *true*, the **CCITTFaxDecode** filter expects the encoded data to be terminated by end-of-block, overriding the **Rows** parameter. If *false*, the **CCITTFaxDecode** filter stops when it has decoded the number of lines indicated by **Rows** or when its data source is exhausted, whichever happens first. Default value: *true*. The end-of-block pattern is the CCITT end-of-facsimile-block (EOFB) or return-to-control (RTC) appropriate for the **K** parameter.
BlackIs1	boolean	*(Optional)* A flag indicating whether 1 bits are to be interpreted as black pixels and 0 bits as white pixels, the reverse of the normal PostScript language convention for image data. Default value: *false*.
DamagedRowsBeforeError	integer	*(Optional; **CCITTFaxDecode** only)* The number of damaged rows of data to be tolerated before an **ioerror** is generated; applies only if **EndOfLine** is *true* and **K** is nonnegative. Tolerating a damaged row means locating its end in the encoded data by searching for an **EndOfLine** pattern, then substituting decoded data from the previous row if the previous row was not damaged, or a white scan line if the previous row was also damaged. Default value: 0.
CloseSource	boolean	*(Optional; LanguageLevel 3; **CCITTFaxDecode** only)* A flag specifying whether closing the filter should also close its data source. Default value: *false*.
CloseTarget	boolean	*(Optional; LanguageLevel 3; **CCITTFaxEncode** only)* A flag specifying whether closing the filter should also close its data target. Default value: *false*.

The CCITT fax standard specifies a bilevel picture encoding in terms of black and white pixels. It does not define a representation for the unencoded image data in terms of 0 and 1 bits in memory. However, the PostScript language (specifically, the **image** operator) does impose a convention: normally, 0 means black and 1 means white. Therefore, the **CCITTFaxEncode** filter normally encodes 0 bits as black pixels and 1 bits as white pixels. Similarly, the **CCITTFaxDecode** filter

normally produces 0 bits for black pixels and 1 bits for white pixels. The **BlackIs1** parameter can be used to reverse this convention if necessary.

The fax encoding method is bit-oriented, not byte-oriented. This means that, in principle, encoded or decoded data might not end at a byte boundary. The **CCITTFaxEncode** and **CCITTFaxDecode** filters deal with this problem in the following ways:

- Unencoded data is treated as complete scan lines, with unused bits inserted at the end of each scan line to fill out the last byte. This is compatible with the convention the **image** operator uses.
- Encoded data is ordinarily treated as a continuous, unbroken bit stream. The **EncodedByteAlign** parameter can be used to cause each encoded scan line to be filled to a byte boundary; this method is not prescribed by the CCITT standard, and fax machines never do this, but some software packages find it convenient to encode data this way.
- When a filter reaches EOD, it always skips to the next byte boundary following the encoded data.

DCTDecode Filter

source /DCTDecode filter
source dictionary /DCTDecode filter

The **DCTDecode** filter decodes grayscale or color image data in JPEG baseline encoded format. The description of the **DCTEncode** filter provides details of the format and the related filter parameters. All of the **DCTEncode** parameters (except **CloseTarget**) are allowed for **DCTDecode**; however, usually no parameters are needed except **ColorTransform** (and possibly **CloseSource**, in LanguageLevel 3), because all information required for decoding an image is normally contained in the JPEG signalling parameters, which accompany the encoded data in the compressed data stream.

The decoded data is a stream of image samples, each of which consists of 1, 2, 3, or 4 color components, interleaved on a per-sample basis. Each component value occupies one 8-bit byte. The dimensions of the image and the number of components per sample depend on parameters that were specified when the image was encoded. Given suitable parameters, the **image** operator can consume data directly from a **DCTDecode** filter.

Note: *The JPEG standard also allows an image's components to be sent as separate scans instead of interleaved; however, that format is not useful with the* ***image*** *operator, because* ***image*** *requires that components from separate sources be read in parallel.*

DCTEncode Filter

target dictionary /DCTEncode filter

The **DCTEncode** filter encodes grayscale or color image data in JPEG baseline format. JPEG is the ISO Joint Photographic Experts Group, an organization responsible for developing an international standard for compression of color image data (see the Bibliography). Another informal abbreviation for this standard is JFIF, for JPEG File Interchange Format. DCT refers to the primary technique (discrete cosine transform) used in the encoding and decoding algorithms. The algorithm can achieve very impressive compression of color images. For example, at a compression ratio of 10 to 1, there is little or no perceptible degradation in quality.

Note: *The compression algorithm is "lossy," meaning that the data produced by the* ***DCTDecode*** *filter is not exactly the same as the data originally encoded by the* ***DCTEncode*** *filter. These filters are designed specifically for compression of sampled continuous-tone images, not for general data compression.*

Input to the **DCTEncode** filter is a stream of image samples, each of which consists of 1, 2, 3, or 4 color components, interleaved on a per-sample basis. Each component value occupies one 8-bit byte. The dimensions of the image and the number of components per sample must be specified in the filter's parameter dictionary. The dictionary can also contain other optional parameters that control the operation of the encoding algorithm. Table 3.22 describes the contents of this dictionary.

TABLE 3.22 Entries in a DCTEncode parameter dictionary

KEY	TYPE	VALUE
Columns	integer	*(Required)* The width of the image in samples per scan line.
Rows	integer	*(Required)* The height of the image in scan lines.
Colors	integer	*(Required)* The number of color components in the image; must be 1, 2, 3, or 4.
HSamples	array, packed array, or string	*(Optional)* A sequence of horizontal sampling factors (one per color component). If **HSamples** is an array or a packed array, the elements must be integers; if it is a string, the elements are interpreted as integers in the range 0 to 255. The *i*th element of the sequence specifies the sampling factor for the *i*th color component. Allowed sampling factors are 1, 2, 3, and 4. The default value is an array containing 1 for all components, meaning that all components are to be sampled at the same rate. When the sampling factors are not all the same, **DCTEncode** subsamples the image for those components whose sampling factors are less than the largest one. For example, if **HSamples** is `[4 3 2 1]` for a 4-color image, then for every 4 horizontal samples of the first component, **DCTEncode** sends only 3 samples of the second component, 2 of the third, and 1 of the fourth. However, **DCTDecode** inverts this sampling process so that it produces the same amount of data as was presented to **DCTEncode**. In other words, this parameter affects only the encoded, and not the unencoded or decoded, representation. The filters deal correctly with the situation in which the width or height of the image is not a multiple of the corresponding sampling factor.
VSamples	array, packed array, or string	*(Optional)* A sequence of vertical sampling factors (one per color component). Interpretation and default value are the same as for **HSamples**. The JPEG standard imposes a restriction on the values in the **HSamples** and **VSamples** sequences, taken together: For each color component, multiply its **HSamples** value by its **VSamples** value, then add all of the products together. The result must not exceed 10.
QuantTables	array or packed array	*(Optional)* An array or packed array of quantization tables (one per color component). The *i*th element of **QuantTables** is the table to be used, after scaling by **QFactor**, for quantization of the *i*th color component. As many as four unique quantization tables can be specified, but several elements of the **QuantTables** array can refer to the same table.

Each table must be an array, a packed array, or a string. If it is an array or a packed array, the elements must be numbers; if it is a string, the elements are interpreted as integers in the range 0 to 255. In either case, each table must contain 64 numbers organized according to the zigzag pattern defined by the JPEG standard. After scaling by **QFactor**, every element is rounded to the nearest integer in the range 1 to 255. Default value: implementation-dependent.

QFactor number

(Optional) A scale factor applied to the elements of **QuantTables**. This parameter enables straightforward adjustment of the tradeoff between image compression and image quality without respecifying the quantization tables. Valid values are in the range 0 to 1,000,000. A value less than 1 improves image quality but decreases compression; a value greater than 1 increases compression but degrades image quality. Default value: 1.0.

HuffTables array or packed array

(Optional) An array or packed array of at least $2 \times$ **Colors** encoding tables. The pair of tables at indices $2 \times i$ and $2 \times i + 1$ in **HuffTables** are used to construct Huffman tables for coding the *i*th color component. The first table in each pair is used for the DC coefficients, the second for the AC coefficients. At most two DC tables and two AC tables can be specified, but several elements of the **HuffTables** array can refer to the same tables. Default value: implementation-dependent.

Each table must be an array, a packed array, or a string. If it is an array or a packed array, the elements must be numbers; if it is a string, the elements are interpreted as integers in the range 0 to 255. The first 16 values specify the number of codes of each length from 1 to 16 bits. The remaining values are the symbols corresponding to each code; they are given in order of increasing code length. This information is sufficient to construct a Huffman coding table according to an algorithm given in the JPEG standard. A **QFactor** value other than 1.0 may alter this computation.

ColorTransform integer

(Optional) A code specifying a transformation to be performed on the sample values:

0 No transformation.

1 If **Colors** is 3, transform RGB values to YUV before encoding and from YUV to RGB after decoding. If **Colors** is 4, transform CMYK values to YUVK before encoding and from YUVK to CMYK after decoding. This option is ignored if **Colors** is 1 or 2.

If performed, these transformations occur entirely within the **DCTEncode** and **DCTDecode** filters. The RGB and YUV used here have nothing to do with the color spaces defined as part of the Adobe imaging model. The purpose of converting from RGB to YUV is to separate luminance and chrominance information (see below).

		The default value of **ColorTransform** is 1 if **Colors** is 3 and 0 otherwise. In other words, conversion between RGB and YUV is performed for all three-component images unless explicitly disabled by setting **ColorTransform** to 0. Additionally, the **DCTEncode** filter inserts an Adobe-defined marker code in the encoded data indicating the **ColorTransform** value used. If present, this marker code overrides the **ColorTransform** value given to **DCTDecode**. Thus it is necessary to specify **ColorTransform** only when decoding data that does not contain the Adobe-defined marker code.
CloseTarget	boolean	*(Optional; LanguageLevel 3)* A flag specifying whether closing the filter should also close its data target. Default value: *false*.

Specifying the optional parameters properly requires understanding the details of the encoding algorithm, which is described in the JPEG standard. The **DCTDecode** and **DCTEncode** filters do not support certain features of the standard that are irrelevant to images following PostScript language conventions; in particular, progressive JPEG is not supported. Additionally, Adobe has made certain choices regarding reserved marker codes and other optional features of the standard; contact the Adobe Developers Association for further information.

The default values for **QuantTables** and **HuffTables** in a **DCTEncode** parameter dictionary are chosen without reference to the image color space and without specifying any particular tradeoff between image quality and compression. Although they will work, they will not produce optimal results for most applications. For superior compression, applications should provide custom **QuantTables** and **HuffTables** arrays rather than relying on the default values.

Better compression is often possible for color spaces that treat luminance and chrominance separately than for those that do not. The RGB to YUV conversion provided by the filters is one attempt to separate luminance and chrominance; it conforms to CCIR recommendation 601-1. Other color spaces, such as the CIE 1976 L*a*b* space, may also achieve this objective. The chrominance components can then be compressed more than the luminance by using coarser sampling or quantization, with no degradation in quality.

Unlike other encoding filters, the **DCTEncode** filter requires that a specific amount of data be written to it: **Columns** × **Rows** samples of **Colors** bytes each. The filter reaches EOD at that point. It cannot accept further data, so attempting

to write to it will cause an **ioerror**. The program must now close the filter file to cause the buffered data and EOD marker to be flushed to the data target.

SubFileDecode Filter

source EODCount EODString /SubFileDecode filter
source dictionary EODCount EODString /SubFileDecode filter
source dictionary /SubFileDecode filter *(LanguageLevel 3)*

The **SubFileDecode** filter does not perform data transformation, but it can detect an EOD condition. Its output is always identical to its input, up to the point where EOD occurs. The data preceding the EOD is called a *subfile* of the underlying data source.

The **SubFileDecode** filter can be used in a variety of ways:

- A subfile can contain data that should be read or executed conditionally, depending on information that is not known until execution. If a program decides to ignore the information in a subfile, it can easily skip to the end of the subfile by invoking **flushfile** on the filter file.
- Subfiles can help recover from errors that occur in encapsulated programs. If the encapsulated program is treated as a subfile, the enclosing program can regain control if an error occurs, flush to the end of the subfile, and resume execution from the underlying data source. The application, not the PostScript interpreter, must provide such error handling; it is not the default error handling provided by the PostScript interpreter.
- The **SubFileDecode** filter enables an arbitrary data source (procedure or string) to be treated as an input file. This use of subfiles does not require detection of an EOD marker.

The **SubFileDecode** filter requires two parameters, *EODCount* and *EODString*, which specify the condition under which the filter is to recognize EOD. The filter will allow data to pass through the filter until it has encountered exactly *EODCount* instances of the *EODString*; then it will reach EOD.

In LanguageLevel 2, *EODCount* and *EODString* are specified as operands on the stack. In LanguageLevel 3, they may alternatively be specified in the **SubFileDecode** parameter dictionary (as shown in Table 3.23). They *must* be specified in the parameter dictionary if the **SubFileDecode** filter is used as one of the filters in a **ReusableStreamDecode** filter (described in the next section).

TABLE 3.23 Entries in a SubFileDecode parameter dictionary *(LanguageLevel 3)*

KEY	TYPE	VALUE
EODCount	integer	*(Required)* The number of occurrences of **EODString** that will be passed through the filter and made available for reading.
EODString	string	*(Required)* The end-of-data string.
CloseSource	boolean	*(Optional)* A flag specifying whether closing the filter should also close its data source. Default value: *false.*

EODCount must be a nonnegative integer. If it is greater than 0, all input data up to and including that many occurrences of *EODString* will be passed through the filter and made available for reading. If *EODCount* is 0, the first occurrence of *EODString* will be consumed by the filter, but it will *not* be passed through the filter.

EODString is ordinarily a string of nonzero length. It is compared with successive subsequences of the data read from the data source. This comparison is based on equality of 8-bit character codes, so matching is case-sensitive. Each occurrence of *EODString* in the data is counted once. Overlapping instances of *EODString* will not be recognized. For example, an *EODString* of eee will be recognized only once in the input XeeeeX.

EODString may also be of length 0, in which case the **SubFileDecode** filter will simply pass *EODCount* bytes of arbitrary data. This is dependable only for binary data, when suitable precautions have been taken to protect the data from any modification by communication channels or operating systems. Ordinary ASCII text is subject to modifications such as translation between different end-of-line conventions, which can change the byte count in unpredictable ways.

A recommended value for *EODString* is a document structuring comment, such as %%EndBinary. Including newline characters in *EODString* is *not* recommended; translating the data between different end-of-line conventions could subvert the string comparisons.

If *EODCount* is 0 and *EODString* is of length 0, detection of EOD markers is disabled; the filter will not reach EOD. This is useful primarily when using procedures or strings as data sources. *EODCount* is not allowed to be negative.

ReusableStreamDecode Filter

source /ReusableStreamDecode filter
source dictionary /ReusableStreamDecode filter

Certain PostScript features require that large blocks of data be available, in their entirety, for use one or more times during the invocation of those features. Examples of such data blocks include:

- Data for a sampled function (see Section 3.10, "Functions")
- Image data or encapsulated PostScript (EPS) referenced from the **PaintProc** procedure of a form dictionary (see Section 4.7, "Forms")
- Mesh data for shading dictionaries (see Section 4.9.3, "Shading Patterns")

Such data can be stored in strings, but only if the amount of data is less than the implementation limit imposed on string objects. (See Appendix B for implementation limits.) To overcome this limit, LanguageLevel 3 defines *reusable streams*.

A reusable stream is a file object produced by a **ReusableStreamDecode** filter. Conceptually, this filter consumes all of its source data at the time the **filter** operator is invoked and then makes the data available as if it were contained in a temporary file. The filter file can be positioned as if it were a random-access disk file; its capacity is limited only by the amount of storage available.

Except for **ReusableStreamDecode** filters, a decoding filter is an input file that can be read only once. When it reaches EOF, it is automatically closed, and no further data can be read from it. No data is read from the filter's source during the execution of the **filter** operator.

In contrast, a **ReusableStreamDecode** filter is an input file that can be read many times. When it reaches EOF, it does *not* automatically close, but merely stays at EOF. It can be repositioned, when it reaches EOF or at any other time, for further reading. In some cases, *all* of the data is read from the filter's source during the execution of the **filter** operator.

A reusable stream has a *length*, which is the total number of bytes in its data source. The stream can be positioned anywhere from 0, which denotes the beginning of the stream, to *length*, which denotes the EOF.

When data is read from the filter's source, it may or may not be buffered in memory or written to a temporary disk file, depending on the type of data source, the availability of storage, and details of the implementation and system memory management.

The **AsyncRead** flag in the filter's parameter dictionary specifies whether all of the data should be read from the data source during the execution of the **filter** operator (**AsyncRead** *false*, the default), or whether this may be postponed until the data is needed (**AsyncRead** *true*). Asynchronous reading may require less memory or have better performance, but caution is required: attempts to read from the same data source through a separate stream may produce incorrect results.

Regardless of the value of **AsyncRead**, a string or file that is used as the data source for a reusable stream, as for any other decoding filter, should be considered read-only until the stream is closed. Writing into such a string or file will have unpredictable consequences for the data read from the stream.

A reusable stream's parameter dictionary can also specify additional filters that are to be applied to the data source before it is read by the **ReusableStreamDecode** filter. This has an effect equivalent to supplying the same filter pipeline as the data source of the **ReusableStreamDecode** filter. However, specifying those filters in the **ReusableStreamDecode** filter dictionary can improve efficiency by allowing the implementation more flexibility in determining how to read and buffer the data.

The following operators can be applied to a reusable stream:

- **closefile** closes the file. This occurs implicitly when the file is reclaimed by the **restore** operator or garbage collection. Closing the file reclaims any temporary memory or disk space that was used to buffer the file's contents.
- **fileposition** returns the current file position. The result is always in the range 0 to *length*.
- **setfileposition** sets the file position to a value in the range 0 to *length*.
- **resetfile** sets the file position to 0.
- **flushfile** sets the file position to *length*.
- **bytesavailable** returns *length* minus the current file position.

Table 3.24 lists the entries in the **ReusableStreamDecode** parameter dictionary.

TABLE 3.24 Entries in a ReusableStreamDecode parameter dictionary

KEY	TYPE	VALUE
Filter	array or name	*(Optional)* An array of names of decoding filters that are to be applied before delivering data to the reader. The names must be specified in the order they should be applied to decode the data. For example, data encoded using LZW and then ASCII base-85 encoding filters should be decoded with the **Filter** value [/ASCII85Decode /LZWDecode]. If only one filter is required, the value of **Filter** may be the name of that filter.
DecodeParms	array or dictionary	*(Optional)* An array of parameter dictionaries used by the decoding filters that are specified by the **Filter** parameter, listed in the same order. If a filter requires no parameters, the corresponding item in the **DecodeParms** array must be **null**. If the value of **Filter** is a name, **DecodeParms** must be the parameter dictionary for that filter. If no parameters are required for any of the decoding filters, **DecodeParms** may be omitted. Note that the **SubFileDecode** filter requires a parameter dictionary with entries for both **EODCount** and **EODString**. All occurrences of **CloseSource** in the parameter dictionaries are ignored. When the reusable stream is closed, all the filters are also closed, independent of the value of **CloseSource** in the reusable stream itself. The original source of the reusable stream is closed only if the value of **CloseSource** in the reusable stream is *true*.
Intent	integer	*(Optional)* A code representing the intended use of the reusable stream, which may help optimize the data storage or caching strategy. If the value is omitted or is not one of the following values, the default value of 0 is used. 0 Image data 1 Image mask data 2 Sequentially accessed lookup table data (such as a threshold array) 3 Randomly accessed lookup table data (such as the table of values for a sampled function)
AsyncRead	boolean	*(Optional)* A flag that controls when data from the source is to be read. If *false*, all the data from the source is read when the filter is created. If *true*, data from the source may or may not be read when the filter is created; reading may be postponed until the data is needed. Any operation on the filter may cause all of the data to be read. Default value: *false*.
CloseSource	boolean	*(Optional)* A flag specifying whether closing the filter should also close its data source. Default value: *false*.

NullEncode Filter

target /NullEncode filter
target dictionary /NullEncode filter

The **NullEncode** filter is an encoding filter that performs no data transformation; its output is always identical to its input. The purpose of this filter is to allow an arbitrary data target (procedure or string) to be treated as an output file. as described in Section 3.13.1, "Data Sources and Targets." Note that there is no **NullDecode** filter as such, because the **SubFileDecode** filter can be configured to serve that function.

The parameter dictionary can be used to specify the **CloseTarget** parameter *(LanguageLevel 3)*.

3.14 Binary Encoding Details

In LanguageLevels 2 and 3, the scanner recognizes two encoded forms of the PostScript language in addition to ASCII. These are *binary token* encoding and *binary object sequence* encoding. All three encoding formats can be mixed in any program.

The *binary token* encoding represents elements of the PostScript language as individual syntactic entities. This encoding emphasizes compactness over efficiency of generation or interpretation. Still, the binary token encoding is usually more efficient than ASCII. Most elements of the language, such as integers, real numbers, and operator names, are represented by fewer characters in the binary encoding than in the ASCII encoding. Binary encoding is most suitable for environments in which communication bandwidth or storage space is the scarce resource.

The *binary object sequence* encoding represents a sequence of one or more PostScript objects as a single syntactic entity. This encoding is not compact, but it can be generated and interpreted very efficiently. In this encoding, most elements of the language are in a natural machine representation or something very close to one. Also, this encoding is oriented toward sending fully or partially precompiled sequences of objects, as opposed to sequences generated "on the fly." Binary object sequence encoding is most suitable for environments in which execution costs dominate communication costs.

Use of the binary encodings requires that the communication channel between the application and the PostScript interpreter be fully transparent. That is, the channel must be able to carry an arbitrary sequence of 8-bit character codes, with no characters reserved for communications functions, no "line" or "record" length restrictions, and so on. If the communication channel is not transparent, an application must use the ASCII encoding. Alternatively, it can make use of the filters that encode binary data as ASCII text (see Section 3.13, "Filtered Files Details").

The various language encodings apply only to characters the PostScript language scanner consumes. Applying **exec** to an executable file or string object invokes the scanner, as does the **token** operator. File operators such as **read** and **readstring**, however, read the incoming sequence of characters as data, not as encoded PostScript programs.

The first character of each token determines what encoding is to be used for that token. If the character code is in the range 128 to 159 (that is, one of the first 32 codes with the high-order bit set), one of the binary encodings is used. For binary encodings, the character code is treated as a *token type*: it determines which encoding is used and sometimes also specifies the type and representation of the token.

Note: *The codes 128 to 159 are control characters in most standard character sets, such as ISO and JIS; they do not have glyphs assigned to them and are unlikely to be used to construct names in PostScript programs. Interpretation of binary encodings can be disabled; see the* ***setobjectformat*** *operator in Chapter 8.*

Characters following the token type character are interpreted according to the same encoding until the end of the token is reached, regardless of character codes. A character code outside the range 128 to 159 can appear within a multiple-byte binary encoding. A character code in the range 128 to 159 can appear within an ASCII string literal or a comment. However, a binary token type character terminates a preceding ASCII name or number token.

In the following descriptions, the term *byte* is synonymous with *character* but emphasizes that the information represents binary data instead of ASCII text.

3.14.1 Binary Tokens

Binary tokens are variable-length binary encodings of certain types of PostScript objects. A binary token represents an object that can also be represented in the ASCII encoding, but it can usually represent the object with fewer characters. The binary token encoding is usually the most compact representation of a program.

Semantically, a binary token is equivalent to some corresponding ASCII token. When the scanner encounters the binary encoding for the integer 123, it produces the same result as when it encounters an ASCII token consisting of the characters 1, 2, and 3. That is, it produces an integer object whose value is 123; the object is the same and occupies the same amount of space if stored in VM, whether it came from a binary or an ASCII token.

Unlike the ASCII and binary object sequence encodings, the binary token encoding is incomplete; not everything in the language can be expressed as a binary token. For example, it does not make sense to have binary token encodings of { and }, because their ASCII encodings are already compact. It also does not make sense to have binary encodings for the names of operators that are rarely used, because their contribution to the overall length of a PostScript program is negligible. The incompleteness of the binary token encoding is not a problem, because ASCII and binary tokens can be mixed.

The binary token encoding is summarized in Table 3.25. A binary token begins with a token type byte. A majority of the token types (132 to 149) are used for binary tokens; the remainder are used for binary object sequences or are unassigned. The token type determines how many additional bytes constitute the token and how the token is interpreted.

TABLE 3.25 Binary token interpretation

TOKEN TYPE(S)	ADDITIONAL BYTES	INTERPRETATION
128–131	—	Binary object sequence (see Section 3.14.2, "Binary Object Sequences").
132	4	32-bit integer, high-order byte first.
133	4	32-bit integer, low-order byte first.
134	2	16-bit integer, high-order byte first.
135	2	16-bit integer, low-order byte first.

136	1	8-bit integer, treating the byte after the token type as a signed number n; $-128 \le n \le 127$.
137	3 or 5	16- or 32-bit fixed-point number. The number representation (*size*, *byte order*, and *scale*) is encoded in the byte immediately following the token type; the remaining 2 or 4 bytes constitute the number itself. The representation parameter is treated as an unsigned integer r: $0 \le r \le 31$ — 32-bit fixed-point number, high-order byte first. *scale* (the number of bits of fraction) is equal to r. $32 \le r \le 47$ — 16-bit fixed-point number, high-order byte first; *scale* equals $r - 32$. $128 \le r \le 175$ — Same as $r - 128$, except that all numbers are given low-order byte first.
138	4	32-bit IEEE standard real, high-order byte first.
139	4	32-bit IEEE standard real, low-order byte first.
140	4	32-bit native real.
141	1	Boolean. The byte following the token type gives the value 0 for *false*, 1 for *true*.
142	1 + *n*	String of length *n*. The parameter *n* is in the byte following the token type; $0 \le n \le 255$. The *n* characters of the string follow the parameter.
143	2 + *n*	Long string of length *n*. The 16-bit parameter *n* is contained in the two bytes following the token type, represented high-order byte first; $0 \le n \le 65{,}535$. The *n* bytes of the string follow the parameter.
144	2 + *n*	Same as 143 except that *n* is encoded low-order byte first.
145 or 146	1	Literal (145) or executable (146) encoded system name. The system name index (in the range 0 to 255) is contained in the byte following the token type. This is described in detail in Section 3.14.3, "Encoded System Names."
147–148	—	Reserved (Display PostScript extension).
149	3 + *data*	Homogeneous number array, which consists of a 4-byte header, including the token type, followed by a variable-length array of numbers whose size and representation are specified in the header. The header is described in detail below.
150–159	—	Unassigned. Occurrence of a token with any of these types will cause a **syntaxerror**.

The encodings for integer, real, and boolean objects are straightforward; they are explained in Section 3.14.4, "Number Representations." The other token types require additional discussion.

Fixed-Point Numbers

A *fixed-point number* is a binary number having integer and fractional parts. The position of the binary point is specified by a separate *scale* value. In a fixed-point number of *n* bits, the high-order bit is the sign, the next $n - scale - 1$ bits are the integer part, and the low-order *scale* bits are the fractional part. For example, if the number is 16 bits wide and *scale* is 5, it is interpreted as a sign, a 10-bit integer part, and a 5-bit fractional part. A negative number is represented in twos-complement form.

There are both 16- and 32-bit fixed-point numbers, enabling an application to make a tradeoff between compactness and precision. Regardless of the token's length, the object produced by the scanner for a fixed-point number is an integer if *scale* is 0; otherwise it is a real number. A 32-bit fixed-point number takes more bytes to represent than a 32-bit real number. It is useful only if the application already represents numbers that way. Using this representation makes somewhat more sense in homogeneous number arrays, described below.

String Tokens

A string token specifies the string's length as a 1- or 2-byte unsigned integer. The specified number of characters of the string follow immediately. All characters are treated literally. There is no special treatment of \ (backslash) or other characters.

Encoded System Names

An *encoded system name* token selects a name object from the system name table and uses it as either a literal or an executable name. This mechanism is described in Section 3.14.3, "Encoded System Names."

Homogeneous Number Arrays

A *homogeneous number array* is a single binary token that represents a literal array object whose elements are all numbers. Figure 3.2 illustrates the organization of the homogeneous number array.

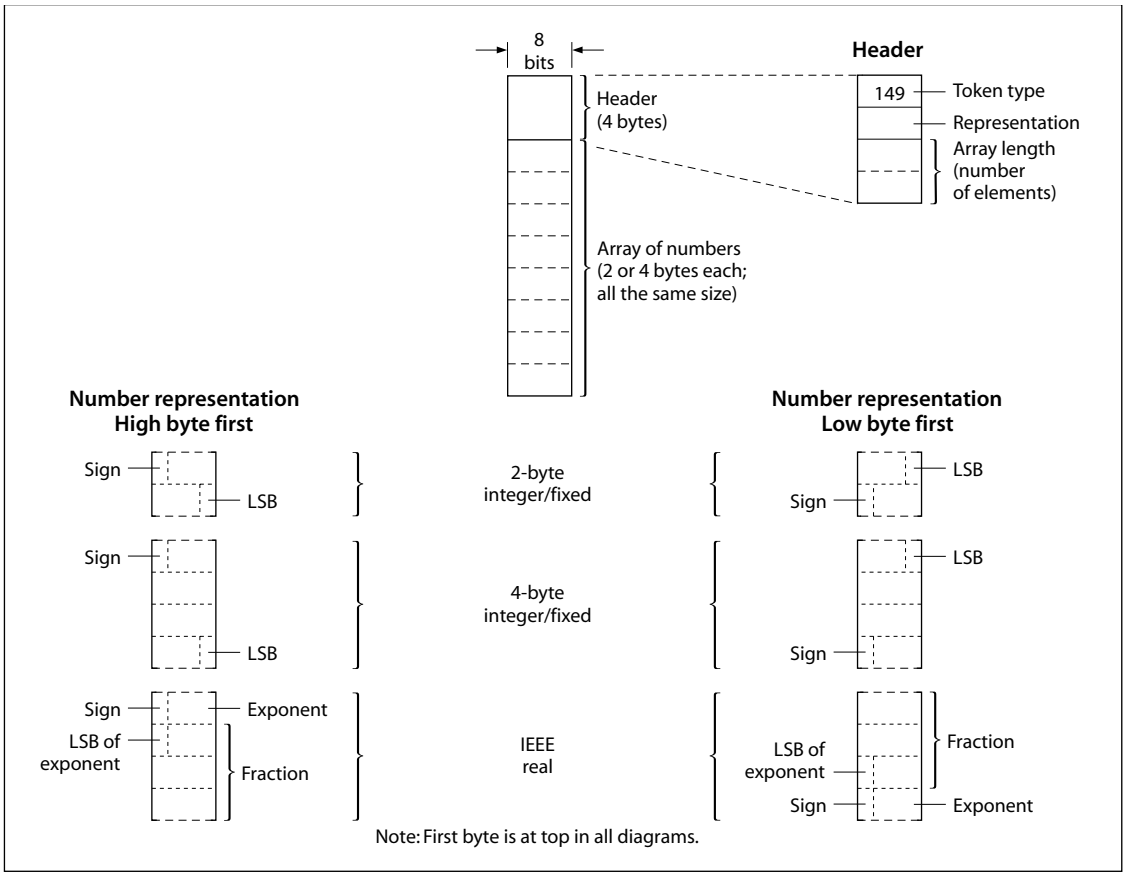


FIGURE 3.2 *Homogeneous number array*

The token consists of a 4-byte header, including the token type, followed by an arbitrarily long sequence of numbers. All of the numbers are represented in the same way, which is specified in the header. The header consists of the token type byte (149, denoting a homogeneous number array), a byte that describes the number representation, and two bytes that specify the array length (number of

elements). The number representation is treated as an unsigned integer r in the range 0 to 255 and is interpreted as shown in Table 3.26.

TABLE 3.26 Number representation in header for a homogeneous number array

REPRESENTATION	INTERPRETATION
$0 \le r \le 31$	32-bit fixed-point number, high-order byte first. *scale* (the number of bits of fraction) is equal to r.
$32 \le r \le 47$	16-bit fixed-point number, high-order byte first. *scale* equals $r - 32$.
48	32-bit IEEE standard real, high-order byte first.
49	32-bit native real.
$128 \le r \le 177$	Same as $r - 128$, except that all numbers are given low-order byte first.

This interpretation is similar to that of the representation parameter r in individual fixed-point number tokens.

The array length is given by the last two bytes of the header, treated as an unsigned 16-bit number n. The byte order in this field is specified by the number representation: $r < 128$ indicates high-order byte first; $r \ge 128$ indicates low-order byte first. Following the header are $2 \times n$ or $4 \times n$ bytes, depending on representation, that encode successive numbers of the array.

When the homogeneous number array is consumed by the PostScript language scanner, the scanner produces a *literal array object*. The elements of this array are all integers if the representation parameter r is 0, 32, 128, or 160, specifying fixed-point numbers with a *scale* of 0. Otherwise, they are all real numbers. Once scanned, such an array is indistinguishable from an array produced by other means and occupies the same amount of space.

Although the homogeneous number array representation is useful in its own right, it is particularly useful with operators that take an encoded number string as an operand. This is described in Section 3.14.5, "Encoded Number Strings."

3.14.2 Binary Object Sequences

A *binary object sequence* is a single token that describes an executable array of objects, each of which may be a simple object, a string, or another array nested to arbitrary depth. The entire sequence can be constructed, transmitted, and scanned as a single, self-contained syntactic entity.

Semantically, a binary object sequence is an ordinary executable array, as if the objects in the sequence were surrounded by { and }, but with one important difference: its execution is immediate instead of deferred. That is, when the PostScript interpreter encounters a binary object sequence in a file being executed directly, the interpreter performs an implicit **exec** operation instead of pushing the array on the operand stack, as it ordinarily would do. This special treatment does not apply when a binary object sequence appears in a context where execution is already deferred—for example, nested in ASCII-encoded { and } or consumed by the **token** operator.

Because a binary object sequence is syntactically a single token, the scanner processes it completely before the interpreter executes it. The VM allocation mode in effect at the time the binary object sequence is scanned determines whether the entire array and all of its composite objects are allocated in local or in global VM.

The encoding emphasizes ease of construction and interpretation over compactness. Each object is represented by 8 successive bytes. In the case of simple objects, these 8 bytes describe the entire object—type, attributes, and value. In the case of composite objects, the 8 bytes include a reference to some other part of the binary object sequence where the value of the object resides. The entire structure is easy to describe using the data type definition facilities of implementation languages, such as C and Pascal.

Figure 3.3 shows the organization of the binary object sequence.

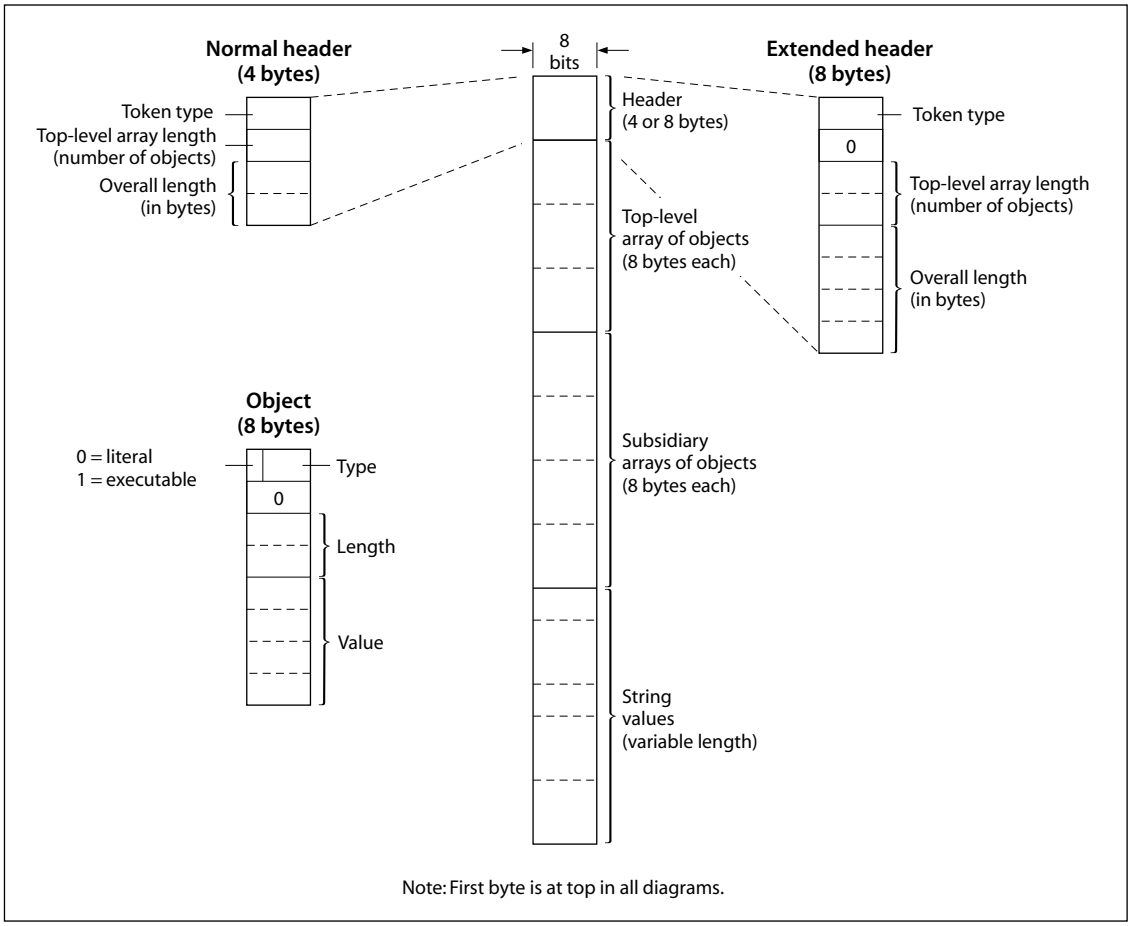


FIGURE 3.3 *Binary object sequence*

A binary object sequence consists of four parts, in the following order:

1. *Header.* 4 or 8 bytes of information about the binary object sequence as a whole
2. *Top-level array.* A sequence of objects, 8 bytes each, which constitute the value of the main array object
3. *Subsidiary arrays.* More 8-byte objects, which constitute the values of nested array objects

4. *String values.* An unstructured sequence of bytes, which constitute the values of string objects and the text of name objects

The first byte of the header is the token type, mentioned earlier. Four token types denote a binary object sequence and select a number representation for all integers and real numbers embedded within it (see Section 3.14.4, "Number Representations"):

128	High-order byte first, IEEE standard real format
129	Low-order byte first, IEEE standard real format
130	High-order byte first, native real format
131	Low-order byte first, native real format

There are two forms of header, *normal* and *extended*, as shown in Figure 3.3. The normal header can describe a binary object sequence that has no more than 255 top-level objects and 65,535 bytes overall. The extended header is required for sequences that exceed these limits.

Following the header is an uninterrupted sequence of 8-byte objects that constitute the top-level array and subsidiary arrays. The length of this sequence is not explicit. It continues until the earliest string value referenced from an object in the sequence, or until the end of the entire token.

The first byte of each object in the sequence gives the object's literal/executable attribute in the high-order bit and its type in the low-order 7 bits. The attribute values are:

0	Literal
1	Executable

The meaning of the type field is given in Table 3.27.

The second byte of an object is unused; its value must be 0. The third and fourth bytes constitute the object's length field; the fifth through eighth bytes constitute its value field. The interpretation of the length and value fields depends on the object's type and is given in Table 3.27. Again, the byte order within these fields is determined by the number representation for the binary object sequence overall.

TABLE 3.27 Object type, length, and value fields

TYPE CODE	OBJECT TYPE	LENGTH FIELD	VALUE FIELD
0	null	Unused	Unused
1	integer	Unused	32-bit signed integer
2	real	Selects representation of value	Floating- or fixed-point number
3	name	Selects interpretation of value	Offset or index
4	boolean	Unused	0 for *false*, 1 for *true*
5	string	Number of elements	Offset of first element
6	immediately evaluated name	Selects interpretation of value	Offset or index
9	array	Number of elements	Offset of first element
10	mark	Unused	Unused

For a real number, the length field selects the representation of the number in the value field: if the length *n* is 0, the value is a floating-point number; otherwise, the value is a fixed-point number, using *n* as its scale factor (see Section 3.14.1, "Binary Tokens").

For strings and arrays, the length field specifies the number of elements (characters in a string or objects in an array). It is treated as a 16-bit unsigned integer. The value field specifies the offset, in bytes, of the start of the object's value relative to the first byte of the first object in the top-level array. An array offset must refer somewhere within the top-level or subsidiary arrays; it must be a multiple of 8. A string offset must refer somewhere within the string values. The strings have no alignment requirement and need not be null-terminated or otherwise delimited. If the length of a string or array object is 0, its value is disregarded.

For name objects, the length field is treated as a 16-bit signed integer *n* that selects one of three interpretations of the value field:

$n > 0$	The value is an offset to the text of the name, just as with a string. *n* is the name's length, which must be within the implementation limit for names.
$n = 0$	Reserved (Display PostScript extension).
$n = -1$	The value is a *system name index* (see Section 3.14.3, "Encoded System Names").

An immediately evaluated name object corresponds to the *//name* syntax of the ASCII encoding (see Section 3.12.2, "Immediately Evaluated Names"). Aside from the type code, its representation is the same as a name. However, with an immediately evaluated name object, the scanner immediately looks up the name in the environment of the current dictionary stack and substitutes the corresponding value for that name. If the name is not found, an **undefined** error occurs.

For the composite objects, there are no enforced restrictions against multiple references to the same value or to recursive or self-referential arrays. However, such structures cannot be expressed directly in the ASCII or binary token encodings of the language; their use violates the interchangeability of the encodings. The recommended structure of a binary object sequence is for each composite object to refer to a distinct value. There is one exception: references from multiple name objects to the same string value are encouraged, because name objects are unique by definition.

The scanner generates a **syntaxerror** when it encounters a binary object sequence that is malformed in any way. Possible causes include:

- An object type that is undefined
- An "unused" field that is not 0
- Lengths and offsets that, combined, would refer outside the bounds of the binary object sequence
- An array offset that is not a multiple of 8 or that refers beyond the earliest string offset

When a **syntaxerror** occurs, the PostScript interpreter pushes the object that caused the error onto the operand stack. For an error detected by the scanner, however, there is no such object, because the error occurs before the scanner has finished creating one. Instead, the scanner fabricates a string object consisting of the characters encountered so far in the current token. If a binary token or binary object sequence was being scanned, the string object produced is a description of the token, such as

```
(bin obj seq, type=128, elements=23, size=234, array out of bounds)
```

rather than the literal characters, which would be gibberish if printed as part of an error message.

3.14.3 Encoded System Names

Both the binary token and binary object sequence encodings provide optional means for representing certain names as small integers rather than as full text strings. Such an integer is referred to as a *system name index*.

A name index is a reference to an element of a name table already known to the PostScript interpreter. When the scanner encounters a name token that specifies a name index rather than a text name, it immediately substitutes the corresponding element of the table. This substitution occurs at scan time, not at execution time. The result of the substitution is an ordinary PostScript name object.

The system name table contains standard operator names, single-letter names, and miscellaneous other useful names. The contents of this table are documented in Appendix F. They are also available as a machine-readable file for use by drivers, translators, and other programs that deal with binary encodings; contact the Adobe Developers Association.

If there is no name associated with a specified system name index, the scanner generates an **undefined** error. The offending command is system*n*, where *n* is the decimal representation of the index.

An encoded system name specifies, as part of the encoding, whether the name is to be literal or executable. A given element of the system name table can be treated as either literal or executable when referenced from a binary token or object sequence.

In the binary object sequence encoding, an immediately evaluated name object analogous to //name can be specified. When such an object specifies a name index, there are *two* substitutions: the first obtains a name object from the table, and the second looks up that name object in the current dictionary stack. The literal or executable attribute of the immediately evaluated name object is disregarded; it has no influence on the corresponding attribute of the resulting object.

A program can depend on a given system name index representing a particular name object. Applications that generate binary-encoded PostScript programs are encouraged to take advantage of encoded system names, because they save both space and time.

Note: *The binary token encoding can reference only the first 256 elements of the system name table. Therefore, this table is organized so that the most commonly used names are in the first 256 elements. The binary object sequence encoding does not have this limitation.*

3.14.4 Number Representations

Binary tokens and binary object sequences use various representations for numbers. Some numbers are the values of number objects (integer and real). Others provide structural information, such as lengths and offsets within binary object sequences.

Different machine architectures use different representations for numbers. The two most common variations are the byte order within multiple-byte integers and the format of real (floating-point) numbers.

Rather than specify a single convention for representing numbers, the language provides a choice of representations. The application program chooses whichever convention is most appropriate for the machine on which it is running. The PostScript language scanner accepts numbers conforming to any of the conventions, translating to its own internal representation when necessary. This translation is needed only when the application and the PostScript interpreter are running on machines with different architectures.

The number representation to be used is specified as part of the token type—the initial character of the binary token or binary object sequence. There are two in-

dependent choices, one for byte order and one for real format. The byte order choices are:

- High-order byte first in a multiple-byte integer or fixed-point number. The high-order byte comes first, followed by successively lower-order bytes.
- Low-order byte first in a multiple-byte integer or fixed-point number. The low-order byte comes first, followed by successively higher-order bytes.

The real format choices are:

- *IEEE standard.* A real number is represented in the 32-bit floating-point format defined in the IEEE *Standard 754-1985 for Binary Floating-Point Arithmetic*. The order of bytes is the same as the integer byte order. For example, if the high-order byte of an integer comes first, then the byte containing the sign and first 7 exponent bits of an IEEE standard real number comes first.
- *Native.* A real number is represented in the native format for the machine on which the PostScript interpreter is running. This may be a standard format or something completely different. The choice of byte order is not relevant. The application program is responsible for finding out the correct format. In general, this choice is useful only in environments where the application and the PostScript interpreter are running on the same machine or on machines with compatible architectures. PostScript programs that use this real number representation are not portable.

Because each binary token or binary object sequence specifies its own number representation, binary encoded programs with different number representations can be mixed. This is a convenience for applications that obtain portions of PostScript programs from different sources.

The **ByteOrder** and **RealFormat** system parameters indicate the native byte order and real number representation of the machine on which the PostScript interpreter is running (see Appendix C). An interactive application can query **RealFormat** to determine whether the interpreter's native real number format is the same as that of the application. If so, translation to and from IEEE format can be avoided.

3.14.5 Encoded Number Strings

Several operators require as operands an indefinitely long sequence of numbers to be used as coordinate values, either absolute or relative. The operators include those dealing with user paths, rectangles, and explicitly positioned text. In the most common use of these operators, all of the numbers are provided as literal values by the applications rather than being computed by the PostScript program.

To facilitate this common use and to streamline the generation and interpretation of numeric operand sequences, these operators permit their operands to be presented in either of two ways:

- As an array object whose elements are numbers to be used successively
- As a string object to be interpreted as an *encoded number string*

An encoded number string is a string that contains a single *homogeneous number array* according to the binary token encoding. That is, the first 4 bytes are treated as a header. The remaining bytes are treated as a sequence of numbers encoded as described in the header. (See Figure 3.2 on page 161.)

An encoded number string is a compact representation of a number sequence both in its external form *and in VM*. Syntactically, it is simply a string object. It remains in that form after being scanned and placed in VM. It is interpreted as a sequence of numbers only when it is used as an operand of an operator that is expecting a number array. Furthermore, even then it is neither processed by the scanner nor expanded into an array object; instead, the numbers are consumed directly by the operator. This arrangement is compact and efficient, particularly for large number sequences.

Example 3.11 shows equivalent ways of invoking **rectfill**, which is one of the LanguageLevel 2 operators that expect number sequences as operands.

Example 3.11

```
[100 200 40 50] rectfill
<95 200004 0064 00c8 0028 0032> rectfill
```

The first line constructs an ordinary PostScript array object containing the numbers and passes it to **rectfill**. This is the most general form, because the **[** and **]**

could enclose an arbitrary computation that produces the numbers and pushes them on the stack.

On the second line, a string object appears in the program. When **rectfill** notices that it has been given a string object, it interprets the value of the string, expecting to find the binary token encoding of a homogeneous number array.

Example 3.11 does not use encoded number strings to best advantage. In this example, it is an ASCII-encoded hexadecimal string enclosed in < and >. A real application would use a more efficient encoding, such as a binary string token or an ASCII base-85 string literal. An ordinary ASCII string enclosed in (and) is unsuitable because of the need to quote special characters.

Operators that use encoded number strings include **rectfill**, **rectstroke**, **rectclip**, **xshow**, **yshow**, and **xyshow**. An encoded user path can represent its numeric operands as an encoded number string; the relevant operators are **ufill**, **ueofill**, **ustroke**, **uappend**, **inufill**, **inueofill**, and **inustroke**.

3.14.6 Structured Output

In some environments, a PostScript program can transmit information back to the application program that generated it. This information includes the values of objects produced by queries, error messages, and unstructured text generated by the **print** operator.

A PostScript program writes all of this data to its standard output file. The application requires a way to distinguish among these different kinds of information received from the PostScript interpreter. To serve this need, the language includes operators to write output in a *structured output format*. This format is basically the same as the binary object sequence representation for input, described in Section 3.14.2, "Binary Object Sequences."

A program that writes structured output should take care when using unstructured output operators, such as **print** and **=**. Because the start of a binary object sequence is indicated by a character whose code is in the range 128 to 131, unstructured output should consist only of character codes outside that range; otherwise, confusion will ensue in the application. Of course, this is only a convention. By prior arrangement, a program can send arbitrary unstructured data to the application.

The operator **printobject** writes an object as a binary object sequence to the standard output file. A similar operator, **writeobject**, writes to any file. The binary object sequence contains a top-level array consisting of one element that is the object being written (see Section 3.14.2, "Binary Object Sequences"). That object, however, can be composite, so the binary object sequence may include subsidiary arrays and strings.

In the binary object sequences produced by **printobject** and **writeobject**, the number representation is controlled by the **setobjectformat** operator. The binary object sequence has a token type that identifies the representation used.

Accompanying the top-level object in the object sequence is a 1-byte *tag*, which is specified as an operand of **printobject** and **writeobject**. This tag is carried in the second byte of the object, which is otherwise unused (see Figure 3.3 on page 164). Only the top-level object receives a tag; the second byte of subsidiary objects is 0. Despite its physical position, the tag is logically associated with the object sequence as a whole.

The purpose of the tag is to enable the PostScript program to specify the intended disposition of the object sequence. A few tag values are reserved for reporting errors (see below). The remaining tag values may be used arbitrarily.

Tag values 0 through 249 are available for general use. Tag values 250 through 255 are reserved for identifying object sequences that have special significance. Of these, only tag value 250 is presently defined; it is used to report errors.

Errors are initiated as described in Section 3.11, "Errors." Normally, when an error occurs, control automatically passes from the PostScript program to a built-in procedure that catches errors. That procedure invokes **handleerror**. Subsequent behavior depends on the definition of **handleerror**. The following description applies to the standard definition of **handleerror**.

If the value of **binary** in the **$error** dictionary is *true* and binary encoding is enabled, **handleerror** writes a binary object sequence with a tag value of 250. But if **binary** is *false* or binary encoding is disabled, **handleerror** writes a human-readable text message whose format is product-dependent.

The binary object sequence that reports an error contains a four-element array as its top-level object. The array elements, ordered as they appear, are:

1. The name Error, which indicates an ordinary error detected by the PostScript interpreter. A different name could indicate another class of errors, in which case the meanings of the other array elements might be different.
2. The name that identifies the specific error—for example, **typecheck**.
3. The object that was being executed when the error occurred. If the object that raised the error is not printable, some suitable substitute is provided—for example, an operator name in place of an operator object.
4. A boolean object (used in the Display PostScript extension), whose normal value is *false*.

CHAPTER 4

Graphics

THE POSTSCRIPT GRAPHICS OPERATORS describe the appearance of pages that are to be reproduced on a raster output device. The facilities described here are intended for both printer and display applications.

The graphics operators form seven main groups:

- *Graphics state operators.* These operators manipulate the data structure called the *graphics state*, which is the global framework within which the other graphics operators execute.
- *Coordinate system and matrix operators.* The graphics state includes the *current transformation matrix* (CTM), which maps coordinates specified by the PostScript program into output device coordinates. The operators in this group manipulate the CTM to achieve any combination of translation, scaling, rotation, reflection, and skewing of user coordinates onto device coordinates.
- *Path construction operators.* The graphics state includes the *current path*, which defines shapes and line trajectories. Path construction operators begin a new path, add line segments and curves to the current path, and close the current path. All of these operators implicitly reference the CTM parameter in the graphics state.
- *Painting operators.* The operators in this group paint graphical elements, such as lines, filled areas, and sampled images, into the raster memory of the output device. These operators are controlled by the current path, current color, and many other parameters in the graphics state.
- *Glyph and font operators.* These operators select and paint character *glyphs* from *fonts* (descriptions of typefaces for representing text characters). Because the PostScript language treats glyphs as general graphical shapes, many of the font operators should be grouped with the path construction or painting oper-

ators. However, the data structures and mechanisms for dealing with glyph and font descriptions are sufficiently specialized that Chapter 5 focuses on them.

- *Device setup operators.* These operators establish the association between raster memory and a physical output device, such as a printer or a display. They are discussed in detail in Chapter 6.
- *Output operators.* Once a page has been completely described, executing an output operator transmits the page to the output device.

This chapter presents general information about device-independent graphics in the PostScript language: how a program describes the abstract appearance of a page. *Rendering*—the device-dependent part of graphics—is covered in Chapter 7.

4.1 Imaging Model

The Adobe imaging model is a simple and unified view of two-dimensional graphics borrowed from the graphic arts. A PostScript program builds an image by placing "paint" on a "page" in selected areas.

- The painted figures may be in the form of letter shapes, general filled shapes, lines, or digitally sampled representations of photographs.
- The paint may be in color or in black, white, or any shade of gray.
- The paint may take the form of a repeating pattern *(LanguageLevel 2)* or a smooth transition between colors *(LanguageLevel 3)*.
- Any of these elements may be clipped to appear within other shapes as they are placed onto the page.
- Once a page has been built up to the desired form, it may be transmitted to an output device.

The PostScript interpreter maintains an implicit *current page* that accumulates the marks made by the *painting operators*. When a program begins, the current page is completely blank. As each painting operator executes, it places marks on the current page. Each new mark completely obscures any marks it may overlay (subject to the effects of the overprint parameter in the graphics state; see Section 4.8.5). This method is known as a *painting model*: no matter what color a mark has—white, black, gray, or color—it is put onto the current page as if it were applied with opaque paint. Once the page has been completely composed,

invoking the **showpage** operator renders the accumulated marks on the output media and then clears the current page to white again.

The principal painting operators (among many others) are as follows:

- **fill** paints an area.
- **stroke** paints lines.
- **image** paints a sampled image.
- **show** paints glyphs representing character shapes.

The painting operators require various parameters, some explicit and others implicit. Chief among the implicit parameters is the *current path* used by **fill**, **stroke**, and **show**. A path consists of a sequence of connected and disconnected points, lines, and curves that together describe shapes and their positions. It is built up through the sequential application of the *path construction operators*, each of which modifies the current path in some way, usually by appending one new element.

Path construction operators include **newpath**, **moveto**, **lineto**, **curveto**, **arc**, and **closepath**. None of the path construction operators places marks on the current page; the painting operators do that. Path construction operators create the shapes that the painting operators paint. Some operators, such as **ufill** and **ustroke**, combine path construction and painting in a single operation for efficiency.

Implicit parameters to the painting operators include the current color, current line width, current font (typeface and size), and many others. There are operators that examine and set each implicit parameter in the graphics state. The values used for implicit parameters are those in effect at the time an operator is invoked.

PostScript programs contain many instances of the following typical sequence of steps:

1. Build a path using path construction operators.
2. Set any implicit parameters if their values need to change.
3. Perform a painting operation.

There is one additional implicit element in the Adobe imaging model that modifies this description: the *current clipping path* outlines the area of the current page on which paint may be placed. Initially, this path outlines the entire imageable area of the current page. By using the **clip** operator, a PostScript program can shrink the path to any shape desired. Although painting operators may attempt to place marks anywhere on the current page, only those marks falling within the current clipping path will affect the page; those falling outside it will not.

4.2 Graphics State

The PostScript interpreter maintains an internal data structure called the *graphics state* that holds current graphics control parameters. These parameters define the global framework within which the graphics operators execute. For example, the **stroke** operator implicitly uses the *current line width* parameter from the graphics state, and the **fill** operator implicitly uses the *current color* parameter.

Most graphics state parameters are ordinary PostScript objects that can be read and altered by the appropriate graphics state operators. For example, the operator **setlinewidth** changes the current line width parameter, and **currentlinewidth** reads that parameter from the graphics state. In general, the operators that set graphics state parameters simply store them unchanged for later use by other graphics operators. However, certain parameters have special properties or behavior:

- Most parameters must be of the correct type or have values that fall into a certain range.
- Parameters that are numeric values, such as color, line width, and miter limit, are forced into legal range, if necessary, and stored as real numbers. If they are later read out, they are always real, regardless of how they were originally specified. However, they are *not* adjusted to reflect capabilities of the raster output device, such as resolution or number of distinguishable colors. Graphics rendering operators perform such adjustments, but the adjusted values are not stored back into the graphics state.
- Certain parameters are composite objects, such as arrays or dictionaries. Graphics operators consult the values of these objects at unpredictable times and may cache them for later use, so altering them can have unpredictable results. A PostScript program should treat the values of graphics state parameters (including those in saved graphics states) as if they were read-only.

- The current path, clipping path, and device parameters are internal objects that are not directly accessible to a PostScript program.

Table 4.1 lists those graphics state parameters that are device-independent and are appropriate to specify in page descriptions. The parameters listed in Table 4.2 control details of the rendering (scan conversion) process and are device-dependent. A page description that is intended to be device-independent should not alter these parameters.

TABLE 4.1 Device-independent parameters of the graphics state

PARAMETER	TYPE	VALUE
CTM	array	The current transformation matrix, which maps positions from user coordinates to device coordinates. This matrix is modified by each application of the coordinate system operators. Initial value: a matrix that transforms default user coordinates to device coordinates.
position	two numbers	The coordinates of the *current point* in user space, the last element of the current path. Initial value: undefined.
path	(internal)	The current path as built up by the path construction operators. Used as an implicit argument by operators such as **fill**, **stroke**, and **clip**. Initial value: empty.
clipping path	(internal)	A path defining the current boundary against which all output is to be cropped. Initial value: the boundary of the entire imageable portion of the output page.
clipping path stack	(internal)	*(LanguageLevel 3)* A stack holding clipping paths that have been saved with the **clipsave** operator and not yet restored with **cliprestore**.
color space	array	*(LanguageLevel 2)* The color space in which color values are to be interpreted. Initial value: **DeviceGray**.
color	(various)	The color to use during painting operations. The type and interpretation of this parameter depends on the current color space. For most color spaces, a color value consists of one to four numbers. Initial value: black.
font	dictionary	The set of graphic shapes (glyphs) that represent characters in the current typeface. Initial value: an invalid font dictionary.
line width	number	The thickness (in user coordinate units) of lines to be drawn by the **stroke** operator. Initial value: 1.0.

line cap	integer	A code that specifies the shape of the endpoints of any open path that is stroked. Initial value: 0 for a square butt end.
line join	integer	A code that specifies the shape of joints between connected segments of a stroked line. Initial value: 0 for mitered joins.
miter limit	number	The maximum length of mitered line joins for the **stroke** operator. This limits the length of "spikes" produced when line segments join at sharp angles. Initial value: 10.0 for a miter cutoff below 11 degrees.
dash pattern	array and number	A description of the dash pattern to be used when lines are painted by the **stroke** operator. Initial value: a normal solid line.
stroke adjustment	boolean	*(LanguageLevel 2)* A flag that specifies whether to compensate for resolution effects that may be noticeable when line thickness is a small number of device pixels. Initial value: *false*.

TABLE 4.2 Device-dependent parameters of the graphics state

PARAMETER	TYPE	VALUE
color rendering	dictionary	*(LanguageLevel 2)* A collection of parameters that determine how to transform CIE-based color specifications to device color values. Initial value: installation-dependent.
overprint	boolean	*(LanguageLevel 2)* A flag that specifies (on output devices that support the overprint control feature) whether painting in one set of colorants cause the corresponding areas of other colorants to be erased (*false*) or left unchanged (*true*). Initial value: *false*.
black generation	procedure	*(LanguageLevel 2)* A procedure that calculates the amount of black to use when converting RGB colors to CMYK. Initial value: installation-dependent.
undercolor removal	procedure	*(LanguageLevel 2)* A procedure that calculates the reduction in the amount of cyan, magenta, and yellow components to compensate for the amount of black added by black generation. Initial value: installation-dependent.
transfer	procedure	A transfer function that adjusts device gray or color component values to correct for nonlinear response in a particular device. Support for four transfer functions is a LanguageLevel 2 feature. Initial value: installation-dependent.
halftone	(various)	A halftone screen for gray and color rendering, specified either as frequency, angle, and spot function or as a halftone dictionary. Halftone dictionaries, as well as support for four halftone screens, are LanguageLevel 2 features. Initial value: installation-dependent.

flatness	number	The precision with which curves are to be rendered on the output device. This number gives the maximum error tolerance, measured in output device pixels. Smaller numbers give smoother curves at the expense of more computation and memory use. Initial value: 1.0.
smoothness	number	*(LanguageLevel 3)* The precision with which color gradients are to be rendered on the output device. This number gives the maximum error tolerance between a shading approximated by piecewise linear interpolation and the true value of a (possibly nonlinear) shading function, expressed as a fraction of the range of each color component. Smaller numbers give smoother color transitions at the expense of more computation and memory use. Initial value: installation-dependent.
device	(internal)	An internal data structure representing the current output device. Initial value: installation-dependent.

Although it contains many objects, the graphics state is not itself a PostScript object and cannot be accessed directly from within a PostScript program. However, there are two mechanisms for saving and later restoring the entire graphics state. One is the *graphics state stack*, managed by the following operators:

- **gsave** pushes a copy of the entire graphics state onto the stack.
- **grestore** restores the entire graphics state to its former value by popping it from the stack.

The graphics state stack, with its LIFO (last in, first out) organization, serves the needs of PostScript programs that are page descriptions. A well-structured document typically contains many graphical elements that are essentially independent of each other and sometimes nested to multiple levels. The **gsave** and **grestore** operators can be used to encapsulate these elements so that they can make local changes to the graphics state without disturbing the graphics state of the surrounding environment.

In some interactive applications, however, a program must switch its attention among multiple, more-or-less independent imaging contexts in an unpredictable order. The second mechanism, available in LanguageLevels 2 and 3, uses *gstate*

objects in virtual memory that contain saved copies of the graphics state. The following LanguageLevel 2 operators manipulate gstate objects:

- **gstate** creates a new gstate object.
- **currentgstate** copies the entire current graphics state into a gstate object.
- **setgstate** replaces the entire current graphics state by the value of a gstate object.

Interactive programs can use these operators to create a separate gstate object for each imaging context and switch among them dynamically as needed.

Note: *Saving a graphics state, with either* ***gsave*** *or* ***currentgstate****, captures every parameter, including such things as the current path and current clipping path. For example, if a nonempty current path exists at the time that* ***gsave****,* ***gstate****, or* ***currentgstate*** *is executed, that path will be reinstated by the corresponding* ***grestore*** *or* ***setgstate****. Unless this effect is specifically desired, it is best to minimize storage demands by saving a graphics state only when the current path is empty and the current clipping path is in its default state.*

4.3 Coordinate Systems and Transformations

Paths and shapes are defined in terms of pairs of *coordinates* on the Cartesian plane. A coordinate pair is a pair of real numbers *x* and *y* that locate a point horizontally and vertically within a Cartesian (two-axis) coordinate system superimposed on the current page. The PostScript language defines a default coordinate system that PostScript programs can use to locate any point on the page.

4.3.1 User Space and Device Space

Coordinates specified in a PostScript program refer to locations within a coordinate system that always bears the same relationship to the current page, regardless of the output device on which printing or displaying will be done. This coordinate system is called *user space.*

Output devices vary greatly in the built-in coordinate systems they use to address pixels within their imageable areas. A particular device's coordinate system is a *device space.* A device space *origin* can be anywhere on the output page. This is because the paper moves through different printers and imagesetters in different

directions. On displays, the origin can vary depending on the window system. Different devices have different resolutions. Some devices even have resolutions that are different in the horizontal and vertical directions.

The operands of the path operators are coordinates expressed in user space. The PostScript interpreter automatically transforms user space coordinates into device space. For the most part, this transformation is hidden from the PostScript program. A program must consider device space only rarely, for certain special effects. This independence of user space from device space is essential to the device-independent nature of PostScript page descriptions.

A coordinate system can be defined with respect to the current page by stating:

- The location of the origin
- The orientation of the *x* and *y* axes
- The lengths of the units along each axis

Initially, the user space origin is located at the lower-left corner of the output page or display window, with the positive *x* axis extending horizontally to the right and the positive *y* axis extending vertically upward, as in standard mathematical practice. The length of a unit along both the *x* and *y* axes is 1/72 inch. This coordinate system is the *default user space.* In default user space, all points within the current page have positive *x* and *y* coordinate values.

Note: *The default unit size (1/72 inch) is approximately the same as a "point," a unit widely used in the printing industry. It is not exactly the same as a point, however; there is no universal definition of a point.*

The default user space origin coincides with the lower-left corner of the *physical* page. Portions of the physical page may not be imageable on certain output devices. For example, many laser printers cannot place marks at the extreme edges of their physical page areas. It may not be possible to place marks at or near the default user space origin. The physical correspondence of page corner to default origin ensures that marks within the imageable portion of the output page will be consistently positioned with respect to the edges of the page.

Coordinates in user space may be specified as either integers or real numbers. Therefore, *the unit size in default user space does not constrain locations to any arbitrary grid.* The resolution of coordinates in user space is not related in any way to the resolution of pixels in device space.

The default user space provides a consistent, dependable starting place for PostScript programs regardless of the output device used. If necessary, the PostScript program may then modify user space to be more suitable to its needs by applying *coordinate transformation operators*, such as **translate**, **rotate**, and **scale**.

What may appear to be absolute coordinates in a PostScript program are not absolute with respect to the current page, because they are expressed in a coordinate system that may slide around and shrink or expand. Coordinate system transformation not only enhances device independence but is a useful tool in its own right. For example, a page description originally composed to occupy an entire page can be incorporated without change as an element of another page description by shrinking the coordinate system in which it is drawn.

Conceptually, user space is an infinite plane. Only a small portion of this plane corresponds to the imageable area of the output device: a rectangular area above and to the right of the origin in default user space. The actual size and position of the area is device- and media-dependent. An application can request a particular page size or other media properties by using the LanguageLevel 2 operator **setpagedevice**, described in Section 6.1.1, "Page Device Dictionary."

4.3.2 Transformations

A *transformation matrix* specifies how to transform the coordinate pairs of one coordinate space into another coordinate space. The graphics state includes the *current transformation matrix* (CTM), which describes the transformation from user space to device space.

The elements of a matrix specify the coefficients of a pair of linear equations that transform the values of coordinates *x* and *y*. However, in graphical applications, matrices are not often thought of in this abstract mathematical way. Instead, a matrix is considered to capture some sequence of geometric manipulations: translation, rotation, scaling, reflection, and so forth. Most of the PostScript language's matrix operators are organized according to this latter model.

The most commonly used matrix operators are those that *modify* the current transformation matrix in the graphics state. Instead of creating a new transformation matrix from nothing, these operators change the existing transformation matrix in some specific way. Operators that modify user space include the following:

- **translate** moves the user space origin to a new position with respect to the current page, leaving the orientation of the axes and the unit lengths unchanged.
- **rotate** turns the user space axes about the current user space origin by some angle, leaving the origin location and unit lengths unchanged.
- **scale** modifies the unit lengths independently along the current *x* and *y* axes, leaving the origin location and the orientation of the axes unchanged.
- **concat** applies an arbitrary linear transformation to the user coordinate system.

Such modifications have a variety of uses:

- *Changing the user coordinate system conventions for an entire page.* For example, in some applications it might be convenient to express user coordinates in centimeters rather than in 72nds of an inch, or it might be convenient to have the origin in the center of the page rather than in the lower-left corner.
- *Defining each graphical element of a page in its own coordinate system*, independent of any other element. The program can then position, orient, and scale each element to the desired location on the page by temporarily modifying the user coordinate system. This allows the description of an element to be decoupled from its placement on the page.

Example 4.1 may aid in understanding the second type of modification. Comments explain what each operator does.

Example 4.1

```
/box                    % Define a procedure to construct a unit-square path in the
   {  newpath           % current user coordinate system, with its lower-left corner at
         0 0 moveto     % the origin.
         0 1 lineto
         1 1 lineto
         1 0 lineto
      closepath
   } def
```

```
gsave                   % Save the current graphics state and create a new one that we
                        % can modify.

72 72 scale             % Modify the current transformation matrix so that everything
                        % subsequently drawn will be 72 times larger; that is, each unit
                        % will represent an inch instead of 1/72 inch.

box fill                % Draw a unit square with its lower-left corner at the origin and
                        % fill it with black. Because the unit size is now 1 inch, this box
                        % is 1 inch on a side.

2 2 translate           % Change the transformation matrix again so that the origin is
                        % displaced 2 inches in from the left and bottom edges of the
                        % page.

box fill                % Draw the box again. This box has its lower-left corner 2 inches
                        % up from and 2 inches to the right of the lower-left corner of
                        % the page.

grestore                % Restore the saved graphics state. Now we are back to default
                        % user space.
```

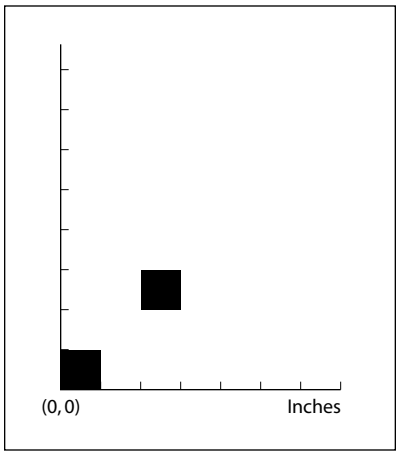


FIGURE 4.1 *The two squares produced by Example 4.1*

Figure 4.1 is a reduction of the entire page containing the two squares painted by Example 4.1, along with scales indicating *x* and *y* positions in inches. This shows how coordinates, such as the ones given to the **moveto** and **lineto** graphics operators, are transformed by the current transformation matrix. By combining trans-

lation, scaling, and rotation, very simple prototype graphics procedures—such as box in the example—can be used to generate an infinite variety of instances.

4.3.3 Matrix Representation and Manipulation

This section presents a brief introduction to the representation and manipulation of matrices. Some knowledge of this topic will make the descriptions of the coordinate system and matrix operators in Chapter 8 easier to understand. It is not essential to understand the details of matrix arithmetic on first reading, but only to obtain a clear geometrical model of the effects of the various transformations.

A two-dimensional transformation is described mathematically by a 3-by-3 matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

In the PostScript language, this matrix is represented as a six-element array object

$$[a\ b\ c\ d\ t_x\ t_y]$$

omitting the matrix elements in the third column, which always have constant values.

This matrix transforms a coordinate pair (x, y) into another coordinate pair (x', y') according to the linear equations

$$\begin{aligned} x' &= ax + cy + t_x \\ y' &= bx + dy + t_y \end{aligned}$$

The common transformations are easily described in this matrix notation. Translation by a specified displacement (t_x, t_y) is described by the matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Scaling by the factor s_x in the horizontal dimension and s_y in the vertical dimension is accomplished by the matrix

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation counterclockwise about the origin by an angle θ is described by the matrix

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 4.2 illustrates the effects of these common transformations.

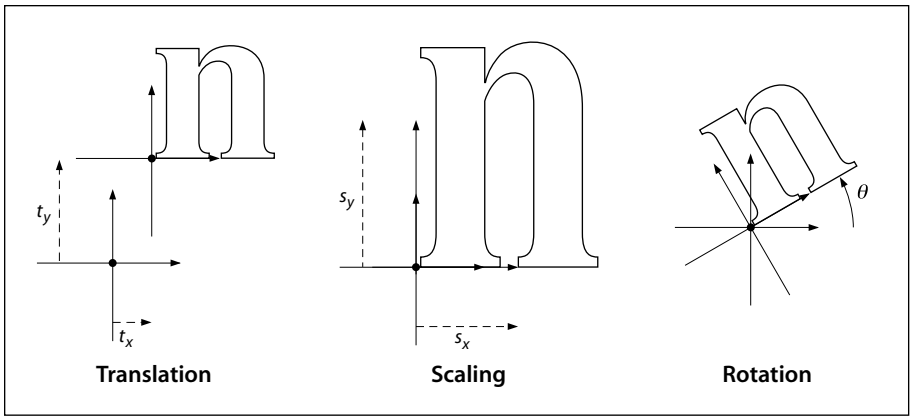


FIGURE 4.2 *Effects of coordinate transformations*

A PostScript program can describe any desired transformation as a sequence of these operations performed in some order. An important property of the matrix notation is that a program can *concatenate* a sequence of operations to form a single matrix that embodies all of them in combination. That is, transforming any pair of coordinates by the single concatenated matrix produces the same result as transforming them by all of the original matrices in sequence. Any linear

transformation from user space to device space can be described by a single transformation matrix, the CTM.

Note: *Concatenation is performed by matrix multiplication. The order in which transformations are concatenated is significant (technically, matrix operations are associative, but not commutative). The requirement that matrices conform during multiplication is what leads to the use of 3-by-3 matrices. Otherwise, 2-by-3 matrices would suffice to describe transformations.*

The operators **translate**, **scale**, and **rotate** each concatenate the CTM with a matrix describing the desired transformation, producing a new matrix that combines the original and additional transformations. This matrix is then established as the new CTM:

$$newCTM = transformation \times originalCTM$$

It is sometimes necessary to perform the *inverse* of a transformation—that is, to find the user space coordinates that correspond to a specific pair of device space coordinates. PostScript programs explicitly do this only occasionally, but it occurs commonly in the PostScript interpreter itself.

Not all transformations are invertible in the way just described. For example, if a matrix contains *a*, *b*, *c*, and *d* elements that are all 0, all user coordinates map to the same device coordinates and there is no unique inverse transformation. Such noninvertible transformations are not very useful and generally arise from unintentional operations, such as scaling by 0. A noninvertible CTM can sometimes cause an **undefinedresult** error to occur during the execution of graphics and font operators.

4.4 Path Construction

In the PostScript language, *paths* define shapes, trajectories, and regions of all sorts. Programs use paths to draw lines, define the shapes of filled areas, and specify boundaries for clipping other graphics.

A path is composed of straight and curved line segments, which may connect to one another or may be disconnected. A pair of segments are said to *connect* only if they are defined consecutively, with the second segment starting where the first one ends. Thus the order in which the segments of a path are defined is signifi-

cant. Nonconsecutive segments that meet or intersect fortuitously are not considered to connect.

A path is made up of one or more disconnected *subpaths*, each comprising a sequence of connected segments. The topology of the path is unrestricted: it may be concave or convex, may contain multiple subpaths representing disjoint areas, and may intersect itself in arbitrary ways. There is an operator, **closepath**, that explicitly connects the end of a subpath back to its starting point; such a subpath is said to be *closed*. A subpath that has not been explicitly closed is *open*.

Paths are represented by data structures internal to the PostScript interpreter. Although a path is not directly accessible as an object, its construction and use are under program control. A path is constructed by sequential application of one or more *path construction operators*. PostScript programs can read out the path or, more commonly, use it to control the application of one of the painting operators described in Section 4.5, "Painting."

Note: *Because the entire set of points defining a path must exist as data simultaneously, there is a limit to the number of segments it may have. Because several paths may also exist simultaneously (the current path and the clipping path, both discussed below, as well as any paths saved by the* ***save***, ***gsave***, ***clipsave***, ***gstate***, *and* ***currentgstate*** *operators), this limit applies to the* total *amount of storage occupied by* all *paths. If a path exhausts the available storage, a* ***limitcheck*** *error occurs. LanguageLevel 1 has a fixed limit for path storage that is implementation-dependent; see Appendix B for more information. In LanguageLevels 2 and 3, there is no such fixed limit; path storage competes with other uses of memory.*

As a practical matter, the limits on path storage are large enough not to impose an unreasonable restriction. It is important, however, that each distinct element of a page be constructed as a separate path, painted, and then discarded *before constructing the next element. Attempting to describe an entire page as a single path is likely to exceed the path storage limit.*

4.4.1 Current Path

The *current path* is part of the graphics state. The path construction operators *modify* the current path, usually by appending to it, and the painting operators implicitly *refer* to the current path. The **gsave** and **grestore** operators respectively save and restore the current path, as they do all components of the graphics state.

A program begins a new path by invoking the **newpath** operator. This initializes the current path to be empty. (Some of the painting operators also reinitialize the current path at the end of their execution.) The program then builds up the definition of the path by applying one or more of the operators that add segments to the current path. These operators may be invoked in any sequence, but the first one invoked must be **moveto**.

The trailing endpoint of the segment most recently added is referred to as the *current point.* If the current path is empty, the current point is undefined. Most operators that add a segment to the current path start at the current point. If the current point is undefined, they generate the error **nocurrentpoint**.

Following is a list of the most common path construction operators. There are other, less common ones as well; see Chapter 8 for complete details.

- **moveto** establishes a new current point without adding a segment to the current path, thereby beginning a new subpath.
- **lineto** adds a straight line segment to the current path, connecting the previous current point to the new one.
- **arc**, **arcn**, **arct**, and **arcto** add an arc of a circle to the current path.
- **curveto** adds a section of a cubic Bézier curve to the current path.
- **rmoveto**, **rlineto**, and **rcurveto** perform the **moveto**, **lineto**, and **curveto** operations, but specify new points via displacements in user space relative to the current point, rather than by absolute coordinates.
- **closepath** adds a straight line segment connecting the current point to the starting point of the current subpath (usually the point most recently specified by **moveto**), thereby *closing* the current subpath.

Note: *Remember that the path construction operators do not place any marks on the page; only the painting operators do that. The usual procedure for painting a graphical element on the page is to define that element as a path and then invoke one of the painting operators. This is repeated for each element on the page.*

All of the points used to describe the path are specified in user space. All coordinates are transformed by the CTM into device space at the time the program adds the point to the current path. Changing the CTM does not affect the coordinates of existing points in device space.

A path that is to be used more than once in a page description can be defined by a PostScript procedure that invokes the operators for constructing the path. Each instance of the path can then be constructed and painted on the page by a three-step sequence:

1. Modify the CTM, if necessary, by invoking coordinate transformation operators to locate, orient, and scale the path to the desired place on the page.
2. Call the procedure to construct the path.
3. Invoke a painting operator to mark the path on the page in the desired manner.

In the common situation that the path description is constant, the LanguageLevel 2 user path operators (described in Section 4.6, "User Paths") can be used to combine steps 2 and 3. The entire sequence can be encapsulated by surrounding it with the operators **gsave** and **grestore**. See Example 4.1 on page 185 for a simple illustration of this technique.

4.4.2 Clipping Path

The graphics state also contains a *clipping path* that limits the regions of the page affected by the painting operators. The closed subpaths of this path define the area that can be painted. Marks falling inside this area will be applied to the page; those falling outside it will not. (Precisely what is considered to be "inside" a path is discussed in Section 4.5.2, "Filling.") The clipping path affects current painting operations only; it has no effect on paths being constructed with the path construction operators listed in Section 4.4.1. When such a path is eventually painted, the results will be limited only by the clipping path current at that time, and not by the one in effect when the path was constructed.

In LanguageLevel 3, the graphics state can also contain a subsidiary stack of saved clipping paths, which are pushed and popped by the **clipsave** and **cliprestore** operators. This enables a program to save and restore just the clipping path without affecting the rest of the graphics state. Because the clipping path stack is an element of the graphics state, wholesale replacement of the graphics state by **grestore** or **setgstate** will replace the entire clipping path stack.

The following operators manage the clipping path:

- **clip** computes a new clipping path from the intersection of the current path with the existing clipping path.
- **clippath** replaces the current path with a copy of the current clipping path.
- **clipsave** *(LanguageLevel 3)* pushes a copy of the current clipping path onto the clipping path stack.
- **cliprestore** *(LanguageLevel 3)* pops the topmost element off the clipping path stack and makes it the current clipping path.

4.5 Painting

The painting operators mark graphical shapes on the current page. This section describes the principal, general-purpose painting operators, **stroke** and **fill**. Variants of these operators combine path construction and painting in a single operation; see Section 4.6, "User Paths." More specialized operators include **shfill**, described in Section 4.9.3, "Shading Patterns"; **image**, described in Section 4.10, "Images"; and the glyph and font operators, described in Chapter 5.

The operators and graphics state parameters described here control the abstract appearance of graphical shapes and are device-independent. Additional, device-dependent facilities for controlling the rendering of graphics in raster memory are described in Chapter 7.

4.5.1 Stroking

The **stroke** operator draws a line along the current path. For each straight or curved segment in the path, the stroked line is centered on the segment with sides parallel to the segment. Each of the path's subpaths is treated separately.

The results of the **stroke** operator depend on the current settings of various parameters in the graphics state. See Section 4.2, "Graphics State," for further information on these parameters, and Chapter 8 for detailed descriptions of the operators that set them.

- The width of the stroked line is determined by the *line width* parameter (see **setlinewidth**).

- The color or pattern of the line is determined by the *color* parameter (see **setgray**, **setrgbcolor**, **sethsbcolor**, **setcmykcolor**, **setcolor**, and **setpattern**; the last three are LanguageLevel 2 operators).
- The line can be drawn either solid or with a program-specified dash pattern, depending on the *dash pattern* parameter (see **setdash**).
- If the subpath is open, the unconnected ends are treated according to the *line cap* parameter, which may be butt, rounded, or square (see **setlinecap**).
- Wherever two consecutive segments are connected, the joint between them is treated according to the *line join* parameter, which may be mitered, rounded, or beveled (see **setlinejoin**). Mitered joins are also subject to the *miter limit* parameter (see **setmiterlimit**).

  Note: *Points at which unconnected segments happen to meet or intersect receive no special treatment. In particular, "closing" a subpath with an explicit* ***lineto*** *rather than with* ***closepath*** *may result in a messy corner, because line caps will be applied instead of a line join.*
- The *stroke adjustment* parameter *(LanguageLevel 2)* requests that coordinates and line widths be adjusted automatically to produce strokes of uniform thickness despite rasterization effects (see **setstrokeadjust** and Section 7.5.2, "Automatic Stroke Adjustment").

4.5.2 Filling

The **fill** operator uses the current color or pattern to paint the entire region enclosed by the current path. If the path consists of several disconnected subpaths, **fill** paints the insides of all subpaths, considered together. Any subpaths that are open are implicitly closed before being filled.

For a simple path, it is intuitively clear what region lies inside. However, for a more complex path—for example, a path that intersects itself or has one subpath that encloses another—the interpretation of "inside" is not always obvious. The path machinery uses one of two rules for determining which points lie inside a path: the *nonzero winding number rule* and the *even-odd rule*, both discussed in detail below.

The nonzero winding number rule is more versatile than the even-odd rule and is the standard rule the **fill** operator uses. Similarly, the **clip** operator uses this rule to determine the inside of the current clipping path. The even-odd rule is occa-

sionally useful for special effects or for compatibility with other graphics systems. The **eofill** and **eoclip** operators invoke this rule.

Nonzero Winding Number Rule

The *nonzero winding number rule* determines whether a given point is inside a path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0 then the point is outside the path; otherwise it is inside.

Note: *The method just described does not specify what to do if a path segment coincides with or is tangent to the chosen ray. Since the direction of the ray is arbitrary, the rule simply chooses a ray that does not encounter such problem intersections.*

For simple convex paths, the nonzero winding number rule defines the inside and outside as one would intuitively expect. The more interesting cases are those involving complex or self-intersecting paths like the ones in Figure 4.3. For a path consisting of a five-pointed star, drawn with five connected straight line segments intersecting each other, the rule considers the inside to be the entire area enclosed by the star, including the pentagon in the center. For a path composed of two concentric circles, the areas enclosed by both circles are considered to be inside, *provided that both are drawn in the same direction.* If the circles are drawn in opposite directions, only the "doughnut" shape between them is inside, according to the rule; the "doughnut hole" is outside.

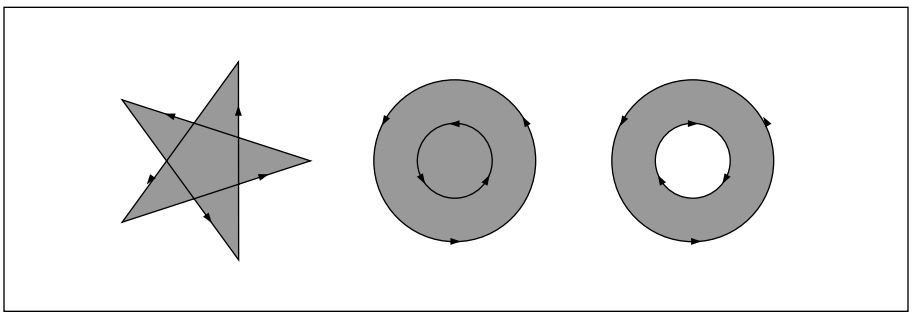


FIGURE 4.3 *Nonzero winding number rule*

Even-Odd Rule

An alternative to the nonzero winding number rule is the *even-odd rule.* This rule determines the "insideness" of a point by drawing a ray from that point in any direction and simply counting the number of path segments that cross the ray, regardless of direction. If this number is odd, the point is inside; if even, the point is outside. This yields the same results as the nonzero winding number rule for paths with simple shapes, but produces different results for more complex shapes.

Figure 4.4 shows the effects of applying the even-odd rule to complex paths. For the five-pointed star, the rule considers the triangular points to be inside the path, but not the pentagon in the center. For the two concentric circles, only the "doughnut" shape between the two circles is considered inside, regardless of the directions in which the circles are drawn.

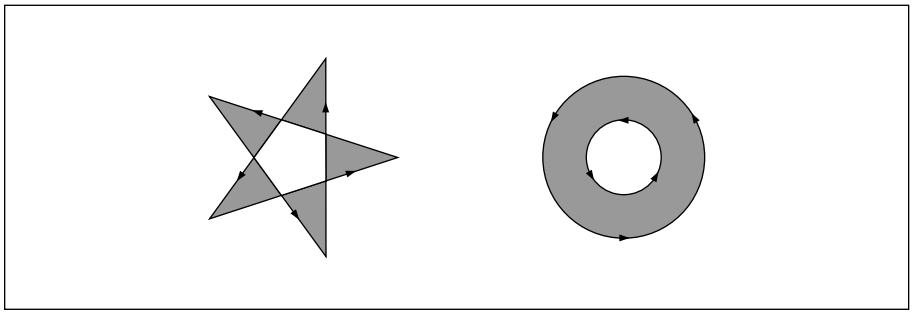


FIGURE 4.4 *Even-odd rule*

4.5.3 Insideness Testing

It is sometimes useful for a program to test whether a point lies inside a path, or whether a path intersects another path, without actually painting anything. The LanguageLevel 2 *insideness-testing* operators can be used for this purpose. They are useful mainly for interactive applications, where they can assist in hit detection; however, they have other uses as well.

There are several insideness-testing operators that vary according to how the paths to be tested are specified. All of the operators return a single boolean result. What it means for a point to be inside a path is that painting the path (by **fill** or

stroke) would cause the device pixel lying under that point to be marked. The insideness tests disregard the current clipping path.

- **infill** tests the current path in the graphics state. There are two forms of this operator. One returns *true* if painting the current path with the **fill** operator would result in marking the device pixel corresponding to a specific point in user space. The second tests whether any pixels within a specified *aperture* would be marked. The aperture is specified by a user path supplied as an operand (see Section 4.6, "User Paths").
- **instroke** is similar to **infill**, but it tests pixels that would be marked by applying the **stroke** operator to the current path, using the current settings of the stroke-related parameters in the graphics state (line width, dash pattern, and so forth).
- **inufill** and **inustroke** are similar to **infill** and **instroke**, but they test a user path supplied as a separate operand, rather than the current path in the graphics state.
- **ineofill** and **inueofill** are similar to **infill** and **inufill**, but they use the even-odd rule instead of the nonzero winding number rule for insideness testing; see Section 4.5.2, "Filling," for more information.

4.6 User Paths

A *user path* is a procedure that is a completely self-contained description of a path in user space. It consists entirely of path construction operators and their coordinate operands expressed as literal numbers. User paths are a LanguageLevel 2 feature.

Special user path painting operators, such as **ustroke** and **ufill**, combine the execution of a user path description with painting operations such as stroking or filling the resulting path. Although these operators can be fully expressed in terms of other path construction and painting operators, they offer a number of advantages in efficiency and convenience:

- They closely match the needs of many application programs.
- Because a user path consists solely of path construction operators and numeric operands, rather than arbitrary computations, it is entirely self-contained: its behavior is guaranteed not to depend on an unpredictable execution environment.

- Every user path carries information about its own bounding box, ensuring that its coordinates will fall within predictable bounds.
- Most of the user path painting operators have no effect on the graphics state. The absence of side effects is a significant reason for the efficiency of the operations. There is no need to build up an explicit current path only to discard it after one use. Although the operators behave as if the path were built up, painted, and discarded in the usual way, their actual implementation is optimized to avoid unnecessary work.
- Because a user path is represented as a self-contained procedure object, the PostScript interpreter can save its output in a cache. This eliminates redundant interpretation of paths that are used repeatedly.

As a result of all these factors, interpreting a user path may be much more efficient than executing an arbitrary PostScript procedure.

4.6.1 User Path Construction

A user path is an array or packed array object consisting of only the following operators and their operands:

	ucache
$ll_x\ ll_y\ ur_x\ ur_y$	**setbbox**
$x\ y$	**moveto**
$dx\ dy$	**rmoveto**
$x\ y$	**lineto**
$dx\ dy$	**rlineto**
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	**curveto**
$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$	**rcurveto**
$x\ y\ r\ angle_1\ angle_2$	**arc**
$x\ y\ r\ angle_1\ angle_2$	**arcn**
$x_1\ y_1\ x_2\ y_2\ r$	**arct**
	closepath

In addition to the special operators **ucache** and **setbbox**, which are used only in constructing user paths, this list includes all standard PostScript operators that append to the current path, with two exceptions: **arcto** is not allowed because it would push results onto the operand stack, and **charpath** is not allowed because the resulting user path would depend on the current font and so would not be self-contained.

Note: *The operators in a user path may be represented either as name objects or as operator objects (such as those associated with the operator names in* ***systemdict****). The latter might result, for example, from applying the* ***bind*** *operator to the user path or to a procedure containing it. Either form of operator designation is acceptable; no advantage is gained by using one in favor of the other.*

The only operands permitted in a user path are literal integers and real numbers. The correct number of operands must be supplied to each operator. The user path must be structured as follows:

1. The optional **ucache** operator places the user path in a special cache, speeding up execution for paths that a program uses frequently. If present, **ucache** must be the first operator invoked in the user path. See Section 4.6.3, "User Path Cache."
2. The next operator invoked must be **setbbox**, which establishes a bounding box in user space enclosing the entire path. Every user path *must* include a call to **setbbox**.
3. The remainder of the user path must consist entirely of path construction operators and their operands. The path is assumed to be empty initially, so the first operator after **setbbox** must be an absolute positioning operator (**moveto**, **arc**, or **arcn**).

All coordinates specified as operands must fall within the bounds specified by **setbbox**, or a **rangecheck** error will occur when the path definition is executed. Any other deviation from the rules above will result in a **typecheck** error.

The user path painting operators interpret a user path as if **systemdict** were the current dictionary. This guarantees that all path construction operators invoked within the path definition will have their standard meanings. To ensure that the definition is self-contained and its meaning independent of its execution environment, aliases are prohibited within a user path definition: it is illegal to use names other than the standard path construction operator names listed above.

To illustrate the construction and use of a user path, Example 4.2 defines a path and paints its interior with the current color.

Example 4.2

```
{   ucache                                  % This is optional
    100 200 400 500 setbbox                 % This is required

    150 200 moveto
    250 200 400 390 400 460 curveto
    400 480 350 500 250 500 curveto
    100 400 lineto

    closepath
} ufill
```

4.6.2 Encoded User Paths

An *encoded user path* is a very compact representation of a user path. It is an array consisting of two string objects or an array and a string, representing the operands and operators of an equivalent user path definition in a compact binary encoding. Encoded user paths are not actually "executed" directly in the same sense as an ordinary PostScript procedure. Rather, user path painting operators such as **ufill** interpret the encoded data structure and perform the operations it encodes.

Note: *The form of operator encoding used in encoded user paths is unrelated to the alternative external encodings of the PostScript language described in Section 3.14, "Binary Encoding Details."*

The elements of an encoded user path are:

- A *data string* or *data array* containing numeric operands. If a string, it is interpreted as an encoded number string according to the representation described in Section 3.14.5, "Encoded Number Strings"; if an array, its elements must all be numbers and are simply used in sequence.
- An *operator string* containing a sequence of encoded path construction operators, one operation code (opcode) per character. Table 4.3 shows the allowed opcode values.

This two-part organization is for the convenience of application programs that generate encoded user paths. In particular, operands always fall on natural addressing boundaries. All characters in both the data and operator strings are interpreted as binary numbers, rather than as ASCII character codes.

TABLE 4.3 Operation codes for encoded user paths

OPCODE	OPERATOR	OPCODE	OPERATOR
0	**setbbox**	6	**rcurveto**
1	**moveto**	7	**arc**
2	**rmoveto**	8	**arcn**
3	**lineto**	9	**arct**
4	**rlineto**	10	**closepath**
5	**curveto**	11	**ucache**
$32 < n \leq 255$	repetition count: repeat next code $n - 32$ times		

Associated with each opcode in the operator string are zero or more operands in the data string or data array. The order of the operands is the same as in an ordinary user path; for example, the **lineto** operator (opcode 3) consumes an *x* and a *y* operand from the data sequence.

Note: *If the encoded user path does not conform to the rules described above, a* ***typecheck*** *error will occur when the path is interpreted. Possible errors include invalid opcodes in the operator string or premature end of the data sequence.*

Example 4.3 shows an encoded version of the user path from Example 4.2, specifying its operands as an ordinary data array encoded in ASCII. Example 4.4 shows the same user path with the operands given as an encoded number string.

Example 4.3

```
{  {  100 200 400 500
      150 200
      250 200 400 390 400 460
      400 480 350 500 250 500
      100 400
   }
   <  0B 00 01 22 05 03 0A >
} ufill
```

Example 4.4

```
{  <  95200014
      0064 00C8 0190 01F4

      0096 00C8
      00FA 00C8 0190 0186 0190 01CC
      0190 01E0 015E 01F4 00FA 01F4
      0064 0190
   >
   <  0B 00 01 22 05 03 0A >
} ufill
```

Example 4.4 illustrates how encoded user paths are likely to be used. Although it does not appear to be more compact than Example 4.3 in its ASCII representation, it occupies less space in virtual memory and executes considerably faster. For clarity of exposition, the example shows the operand as a hexadecimal literal string; an ASCII base-85 string literal or a binary string token would be more compact.

4.6.3 User Path Cache

Some PostScript programs define paths that are repeated many times. To optimize the interpretation of such paths, the PostScript language provides a facility called the *user path cache*. This cache, analogous to the font cache, retains the results from previously interpreted user path definitions. When the PostScript interpreter encounters a user path that is already in the cache, it substitutes the cached results instead of reinterpreting the path definition.

There is a nontrivial cost associated with caching a user path: extra computation is required and existing paths may be displaced from the cache. Because most user paths are used once and immediately thrown away, it does not make sense to place every user path in the cache. Instead, the application program must explicitly identify which user paths are to be cached. It does so by invoking the **ucache** operator as the first operation in a user path definition, before **setbbox**, as shown in Example 4.5.

Example 4.5

```
/Circle1
   {   ucache
       –1 –1 1 1 setbbox
       0 0 1 0 360 arc
   } cvlit def
Circle1 ufill
```

The **ucache** operator notifies the PostScript interpreter that the enclosing user path should be placed in the cache if it is not already there, or retrieved from the cache if it is. (Invoking **ucache** outside a user path has no effect.) This cache management is not performed directly by **ucache**; rather, it is performed by the painting operator applied to the user path (**ufill** in Example 4.5). This is because the results retained in the cache differ according to what painting operation is performed. User path painting operators produce the same effects on the current page whether the cache is accessed or not.

Caching is based on the *value* of a user path object. That is, two user paths are considered the same for caching purposes if all of their corresponding elements are equal, *even if the objects themselves are not.* A user path placed in the cache need not be explicitly retained in virtual memory. An equivalent user path appearing literally later in the program can take advantage of the cached information. Of course, if it is known that a given user path will be used many times, defining it explicitly in VM avoids creating it multiple times.

User path caching, like font caching, is effective across translations of the user coordinate system, but not across other transformations, such as scaling or rotation. In other words, multiple instances of a given user path painted at different places on the page will take advantage of the user path cache when the current transformation matrix has been altered only by **translate**. If the CTM has been altered by **scale** or **rotate**, the instances will be treated as if they were described by different user paths.

Two other features of Example 4.5 are important to note:

- The user path object is explicitly saved for later use (as the value of Circle1 in this example). This is done in anticipation of painting the same path multiple times.

- The **cvlit** operator is applied to the user path object to remove its executable attribute. This ensures that the subsequent reference to Circle1 pushes the object on the operand stack rather than inappropriately executing it as a procedure. It is unnecessary to do this if the user path is to be consumed immediately by a user path painting operator and not saved for later use.

Note: *It is necessary to build the user path as an executable array with* { *and* }*, rather than as a literal array with* [*and*]*, so that the user path construction operators are not executed while the array is being built. Executable arrays have deferred execution.*

4.6.4 User Path Operators

There are three categories of user path operator:

- User path painting operators such as **ustroke**, **ufill**, and **ueofill**, which combine interpretation of a user path with a standard painting operation (**stroke** or **fill**)
- Some of the insideness-testing operators (see Section 4.5.3, "Insideness Testing")
- Miscellaneous operators involving user paths, such as **uappend**, **upath**, and **ustrokepath**

The *userpath* operand to any of these operators is one of the following:

- For an ordinary user path, an array (not necessarily executable) whose length is at least 5.
- For an encoded user path, an array of two elements. The first element is either an array whose elements are all numbers or a string that can be interpreted as an encoded number string (see Section 3.14.5, "Encoded Number Strings"). The second element is a string that encodes a sequence of operators, as described in Table 4.3 on page 201.

In either case, the value of the object must conform to the rules for constructing user paths, as detailed in preceding sections. If the user path is malformed, a **typecheck** error will occur.

The user path painting operators **ustroke**, **ufill**, and **ueofill** interpret a user path as if it were an ordinary PostScript procedure being executed with **systemdict** as the current dictionary; they then perform the corresponding standard painting oper-

ation (**stroke**, **fill**, or **eofill**). The user path operators implicitly invoke **newpath** before interpreting the user path, and enclose the entire operation with **gsave** and **grestore**. The overall effect is to define a path and paint it, leaving no side effects in the graphics state or anywhere else except in raster memory.

Several of the operators take an optional matrix as their final operand. This is a six-element array of numbers describing a transformation matrix. A matrix is distinguished from a user path (which is also an array) by the number and types of its elements.

There is no user path clipping operator. Because the whole purpose of the clipping operation is to alter the current clipping path, there is no way to avoid building the path. The best way to clip with a user path is

```
newpath userpath uappend clip newpath
```

Under favorable conditions, this operation can still take advantage of information in the user path cache.

Note: *The* ***uappend*** *operator and the user path painting operators perform a temporary adjustment to the current transformation matrix as part of their execution, rounding the* t_x *and* t_y *components of the CTM to the nearest integer values. This ensures that scan conversion of the user path produces uniform results when it is placed at different positions on the page through translation. This adjustment is especially important if the user path is cached. The adjustment is not ordinarily visible to a PostScript program, and is not mentioned in the descriptions of the individual operators.*

4.6.5 Rectangles

Because rectangles are used very frequently, it is useful to have a few operators to paint them directly as a convenience to application programs. Also, knowing that the figure will be a rectangle allows execution to be significantly optimized. The rectangle operators are similar to the user path painting operators in that they combine path construction with painting, but their operands are considerably simpler in form.

A rectangle is defined in the user coordinate system, just as if it were constructed as an ordinary path. The LanguageLevel 2 rectangle operators **rectfill**, **rectstroke**, and **rectclip** accept their operands in any of three different forms:

- Four numbers *x*, *y*, *width*, and *height*, describing a single rectangle. The rectangle's sides are parallel to the axes in user space. It has corners located at coordinates (x, y), $(x + width, y)$, $(x + width, y + height)$, and $(x, y + height)$. Note that *width* and *height* can be negative.
- An arbitrarily long sequence of numbers represented as an array.
- An arbitrarily long sequence of numbers represented as an encoded number string, as described in Section 3.14.5, "Encoded Number Strings."

The sequence in the latter two operand forms must contain a multiple of four numbers. Each group of four consecutive numbers is interpreted as the *x*, *y*, *width*, and *height* values defining a single rectangle. The effect produced is equivalent to specifying all the rectangles as separate subpaths of a single combined path, which is then operated on by a single **stroke**, **fill**, or **clip** operator.

The PostScript interpreter draws all rectangles in a counterclockwise direction in user space, regardless of the signs of the *width* and *height* operands. This ensures that when multiple rectangles overlap, all of their interiors are considered to be inside the path according to the nonzero winding number rule.

4.7 Forms

A *form* is a self-contained description of any arbitrary graphics, text, or sampled images that are to be painted multiple times, either on several pages or at several locations on the same page. The appearance of a form is described by a PostScript procedure that invokes graphics operators. Language support for forms is a LanguageLevel 2 feature.

What distinguishes a form from an ordinary procedure is that it is self-contained and behaves according to certain rules. By defining a form, a program declares that each execution of the form will produce the same output, which depends only on the graphics state at the time the form is executed. The form's definition does not refer to variable information in virtual memory, and its execution has no side effects in VM.

These rules permit the PostScript interpreter to save the graphical output of the form in a cache. Later, when the same form is used again, the interpreter substitutes the saved output instead of reexecuting the form's definition. This can significantly improve performance when the form is used many times.

There are various uses for forms:

- As its name suggests, a form can serve as the template for an entire page. For example, a program that prints filled-in tax forms can first paint the fixed template as a form, then paint the variable information on top of it.
- A form can also be any graphical element that is to be used repeatedly. For example, in output from computer-aided design systems, it is common for certain standard components to appear many times. A company's logo can be treated as a form.

4.7.1 Using Forms

The use of forms requires two steps:

1. *Describe the appearance of the form*. Create a *form dictionary* containing descriptive information about the form. A crucial element of the dictionary is the ***PaintProc*** *procedure*, a PostScript procedure that can be executed to paint the form.
2. *Invoke the form*. Invoke the **execform** operator with the form dictionary as the operand. Before doing so, a program should set appropriate parameters in the graphics state; in particular, it should alter the current transformation matrix to control the position, size, and orientation of the form in user space.

Every form dictionary must contain a **FormType** entry, which identifies the particular *form type* that the dictionary describes and determines the format and meaning of its remaining entries. At the time of publication, only one form type, type 1, has been defined. Table 4.4 shows the contents of the form dictionary for this form type. (The dictionary can also contain any additional entries that its **PaintProc** procedure may require.)

TABLE 4.4 Entries in a type 1 form dictionary

KEY	TYPE	VALUE
FormType	integer	*(Required)* A code identifying the form type that this dictionary describes. The only valid value defined at the time of publication is 1.
XUID	array	*(Optional)* An *extended unique ID* that uniquely identifies the form (see Section 5.6.2, "Extended Unique ID Numbers"). The presence of an **XUID** entry in a form dictionary enables the PostScript interpreter to save cached output from the form for later use, even when the form dictionary is loaded into virtual memory multiple times (for instance, by different jobs). To ensure correct behavior, **XUID** values must be assigned from a central registry. This is particularly appropriate for forms treated as named resources. Forms that are created dynamically by an application program should *not* contain **XUID** entries.
BBox	array	*(Required)* An array of four numbers in the form coordinate system, giving the coordinates of the left, bottom, right, and top edges, respectively, of the form's bounding box. These boundaries are used to clip the form and to determine its size for caching.
Matrix	matrix	*(Required)* A transformation matrix that maps the form's coordinate space into user space. This matrix is concatenated with the current transformation matrix before the **PaintProc** procedure is called.
PaintProc	procedure	*(Required)* A PostScript procedure for painting the form.
Implementation	any	An additional entry inserted in the dictionary by the **execform** operator, containing information used by the interpreter to support form caching. The type and value of this entry are implementation-dependent.

The form is defined in its own *form coordinate system*, defined by concatenating the matrix specified by the form dictionary's **Matrix** entry with the current transformation matrix each time the **execform** operator is executed. The form dictionary's **BBox** value is interpreted in the form coordinate system, and the **PaintProc** procedure is executed within that coordinate system.

The **execform** operator first checks whether the form dictionary has previously been used as an operand to **execform**. If not, it verifies that the dictionary contains the required elements and makes the dictionary read-only. It then paints the form, either by invoking the form's **PaintProc** procedure or by substituting cached output produced by a previous execution of the same form.

Whenever **execform** needs to execute the form definition, it does the following:

1. Invokes **gsave**
2. Concatenates the matrix from the form dictionary's **Matrix** entry with the CTM
3. Clips according to the **BBox** entry
4. Invokes **newpath**
5. Pushes the form dictionary on the operand stack
6. Executes the form's **PaintProc** procedure
7. Invokes **grestore**

The **PaintProc** procedure is expected to consume its dictionary operand and to use the information at hand to paint the form. It must obey certain guidelines to avoid disrupting the environment in which it is executed:

- It should not invoke any of the operators listed in Appendix G as unsuitable for use in encapsulated PostScript files.
- It should not invoke **showpage**, **copypage**, or any device setup operator.
- Except for removing its dictionary operand, it should leave the stacks unchanged.
- It should have no side effects beyond painting the form. It should not alter objects in virtual memory or anywhere else. Because of the effects of caching, the **PaintProc** procedure is called at unpredictable times and in unpredictable environments. It should depend only on information in the form dictionary and should produce the same effect every time it is called.

Form caching is most effective when the graphics state does not change between successive invocations of **execform** for a given form. Changes to the translation components of the CTM usually do not influence caching behavior; other changes may require the interpreter to reexecute the **PaintProc** procedure.

4.8 Color Spaces

The PostScript language includes powerful facilities for specifying the colors of graphical objects to be marked on the current page. The color facilities are divided into two parts:

- *Color specification*. A PostScript program can specify abstract colors in a device-independent way. Colors can be described in any of a variety of color systems, or *color spaces*. Some color spaces are related to device color representation (grayscale, RGB, CMYK), others to human visual perception (CIE-based). Certain special features are also modeled as color spaces: patterns, color mapping, separations, and high-fidelity and multitone color.
- *Color rendering*. The PostScript interpreter reproduces colors on the raster output device by a multiple-step process that includes color conversion, gamma correction, halftoning, and scan conversion. Certain aspects of this process are under PostScript language control. However, unlike the facilities for color specification, the color rendering facilities are device-dependent and ordinarily should not be accessed from a page description.

This section describes the color specification facilities of the PostScript language. It covers everything that most PostScript programs need in order to specify colors. Chapter 7 describes the facilities for controlling color rendering; a program should use those facilities only to configure or calibrate an output device or to achieve special device-dependent effects.

Figures 4.5 and 4.6 on pages 212 and 213 illustrate the organization of the PostScript language features for dealing with color, showing the division between (device-independent) color specification and (device-dependent) color rendering.

4.8.1 Types of Color Space

As described in Section 4.5, "Painting," marks placed on the page by operators such as **fill** and **stroke** have a color that is determined by the *current color* parameter of the graphics state. A color value consists of one or more *color components*, which are usually numbers. For example, a gray level can be specified by a single number ranging from 0.0 (black) to 1.0 (white). Full color values can be specified in any of several ways; a common method uses three numbers to specify red, green, and blue components.

In LanguageLevels 2 and 3, color values are interpreted according to the *current color space*, another parameter of the graphics state. A PostScript program first selects a color space by invoking the **setcolorspace** operator. It then selects color values within that color space with the **setcolor** operator. There are also convenience operators—**setgray**, **setrgbcolor**, **sethsbcolor**, **setcmykcolor**, and **setpattern**—that select both a color space and a color value in a single step.

In LanguageLevel 1, this distinction between color spaces and color values is not explicit, and the set of color spaces is limited. Colors can be specified only by **setgray**, **setrgbcolor**, **sethsbcolor**, and (in some implementations) **setcmykcolor**. However, in those color spaces that are supported, the color values produce consistent results from one LanguageLevel to another.

The **image** and **colorimage** operators, introduced in Section 4.10, "Images," enable sampled images to be painted on the current page. Each individual sample in an image is a color value consisting of one or more components to be interpreted in some color space. Since the color values come from the image itself, the current color in the graphics state is not used.

Whether color values originate from the graphics state or from a sampled image, all later stages of color processing treat them the same way. The following sections describe the semantics of color values that are specified as operands to the **setcolor** operator, but the same semantics also apply to color values originating as image samples.

Color spaces can be classified into *color space families*. Spaces within a family share the same general characteristics; they are distinguished by parameter values supplied at the time the space is specified. The families, in turn, fall into three categories:

- *Device color spaces* directly specify colors or shades of gray that the output device is to produce. They provide a variety of color specification methods, including gray level, RGB (red-green-blue), HSB (hue-saturation-brightness), and CMYK (cyan-magenta-yellow-black), corresponding to the color space families **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**. (HSB is merely an alternate convention for specifying RGB colors.) Since each of these families consists of just a single color space with no parameters, they are sometimes loosely referred to as the **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces.

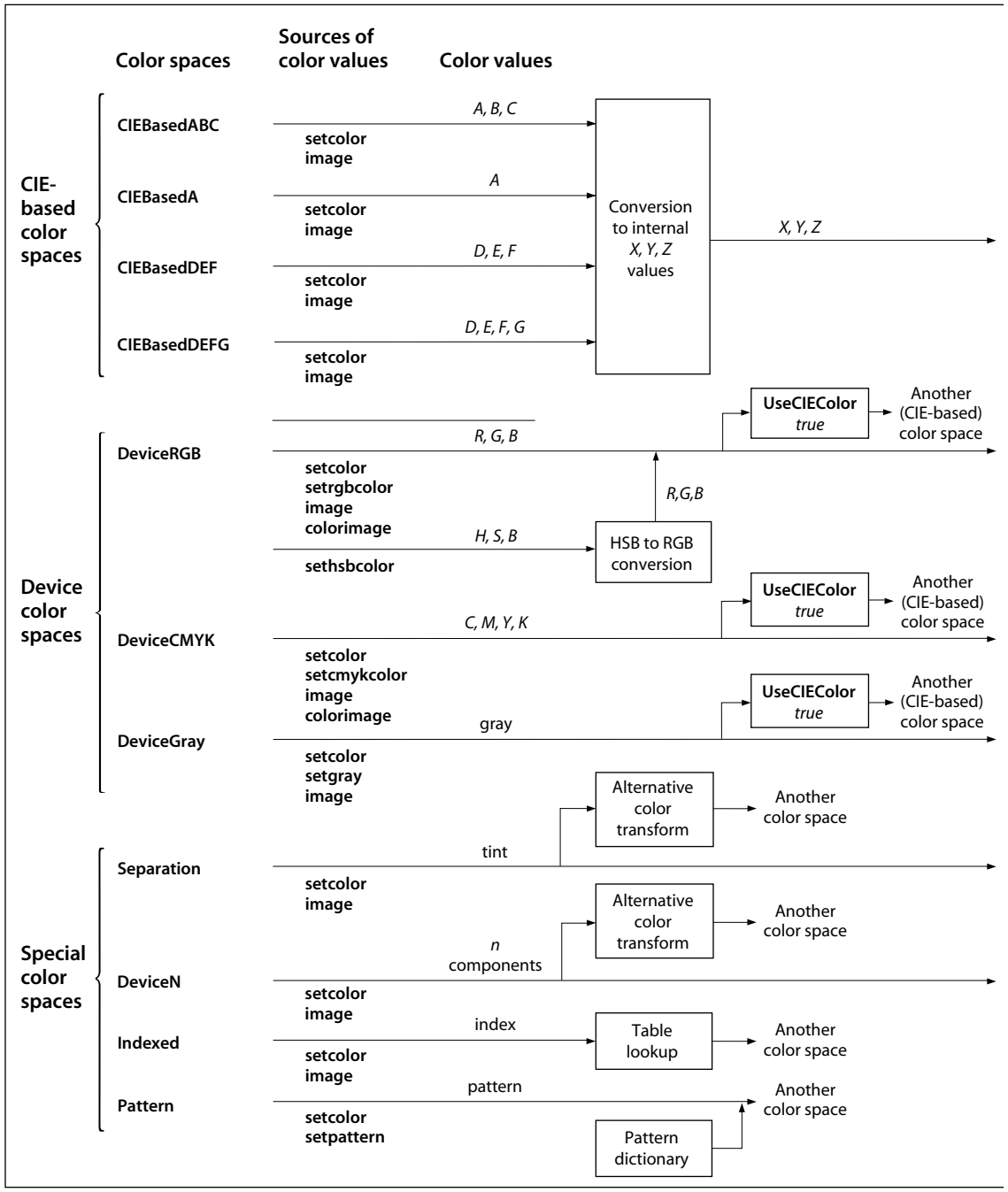


FIGURE 4.5 *Color specification*

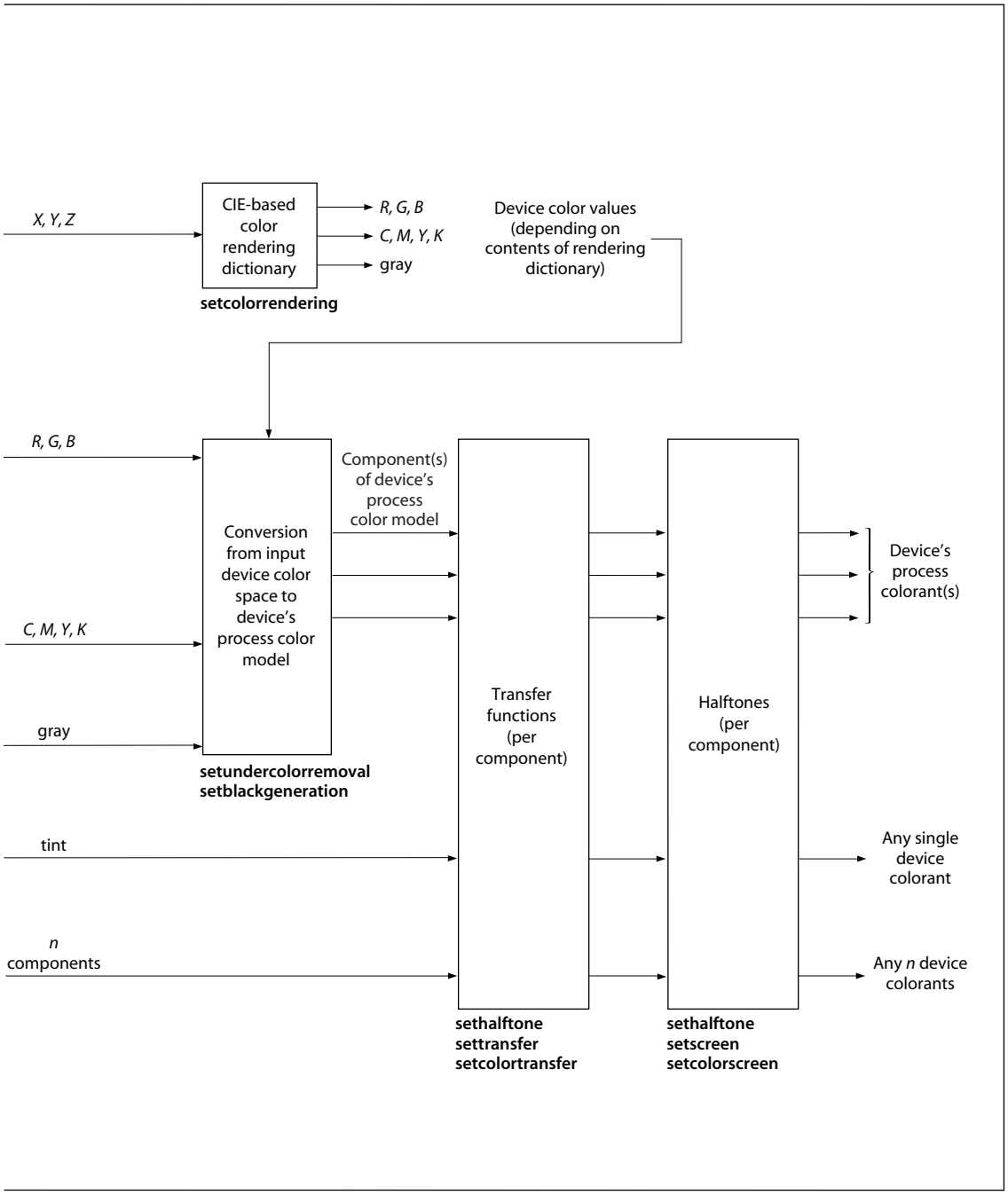


FIGURE 4.6 *Color rendering*

- *CIE-based color spaces* are based on an international standard for color specification created by the Commission Internationale de l'Éclairage (International Commission on Illumination). These spaces allow colors to be specified in a way that is independent of the characteristics of any particular output device. Color space families in this category include **CIEBasedABC**, **CIEBasedA**, **CIEBasedDEF**, and **CIEBasedDEFG**. Individual color spaces within these families are specified by means of dictionaries containing the parameter values needed to define the space.
- *Special color spaces* add features or properties to an underlying color space. They include facilities for patterns, color mapping, separations, and high-fidelity and multitone color. The corresponding color space families are **Pattern**, **Indexed**, **Separation**, and **DeviceN**. Individual color spaces within these families are specified by means of additional parameters.

Whatever type of color space a PostScript program uses to *specify* a color, the process of *rendering* that color on a particular output device is under separate control. Color rendering is discussed in Chapter 7.

The following operators control the selection of color spaces and color values:

- **setcolorspace** sets the color space parameter in the graphics state; **currentcolorspace** returns the current color space parameter.

  The operand to **setcolorspace** is an array object containing as its first element a name object identifying the desired color space. The remaining array elements, if any, are parameters that further characterize the color space; their number and types vary according to the particular color space selected. For color spaces that do not require parameters, the operand to **setcolorspace** can simply be the color space name itself instead of an array; **currentcolorspace** always returns an array.

  The following color space families are standard in LanguageLevel 2:

DeviceGray	**CIEBasedABC**	**Pattern**
DeviceRGB	**CIEBasedA**	**Indexed**
DeviceCMYK		**Separation**

  LanguageLevel 3 supports the following additional families:

CIEBasedDEF	**DeviceN**
CIEBasedDEFG	

- **setcolor** sets the current color parameter in the graphics state; **currentcolor** returns the current color parameter. Depending on the color space, **setcolor** requires one or more operands, each specifying one component of the color value.
- **setgray**, **setrgbcolor**, **sethsbcolor**, **setcmykcolor**, and **setpattern** set the color space implicitly and the current color value as specified by the operands. **currentgray**, **currentrgbcolor**, **currenthsbcolor**, and **currentcmykcolor** return the current color according to an implicit color space; in certain limited cases, the latter operators also perform conversions if the current color space differs from the implicit one.

Note: *Color specification operators such as* ***setcolorspace***, ***setcolor***, *and* ***setpattern*** *sometimes install composite objects, such as arrays or dictionaries, as parameters in the graphics state. To ensure predictable behavior, a PostScript program should thereafter treat all such objects as if they were read-only.*

In certain circumstances, it is illegal to invoke operators that specify colors or other color-related parameters in the graphics state. This restriction occurs when defining graphical figures whose colors are to be specified separately each time they are used. Specifically, the restriction applies:

- After execution of **setcachedevice** or **setcachedevice2** in a **BuildGlyph**, **BuildChar**, or **CharStrings** procedure of a font dictionary (see Sections 5.7, "Type 3 Fonts"; "Type 1 CIDFonts" on page 376; and 5.9.3, "Replacing or Adding Individual Glyphs")
- In the **PaintProc** procedure of an uncolored tiling pattern (see Section 4.9, "Patterns")

In these circumstances, invoking any of the following operators will cause an **undefined** error:

colorimage	**setcolorscreen**	**setpattern**
image	**setcolorspace**	**setrgbcolor**
setblackgeneration	**setcolortransfer**	**setscreen**
setcmykcolor	**setgray**	**settransfer**
setcolor	**sethalftone**	**setundercolorremoval**
setcolorrendering	**sethsbcolor**	**shfill**

Note that the **imagemask** operator is not restricted. This is because it does not specify colors, but rather designates places where the current color is to be painted.

4.8.2 Device Color Spaces

The device color spaces enable a page description to specify color values that are directly related to their representation on an output device. Color values in these spaces map directly—or via simple conversions—to the application of device colorants, such as quantities of ink or intensities of display phosphors. This enables a PostScript program to control colors precisely for a *particular* device, but the results may not be consistent between *different* devices.

Output devices form colors either by adding light sources together or by subtracting light from an illuminating source. Computer displays and film recorders typically add colors, while printing inks typically subtract them. These two ways of forming colors give rise to two complementary forms of color specification: the additive RGB specification and the subtractive CMYK specification. The corresponding device color spaces are as follows:

- **DeviceRGB** controls the intensities of red, green, and blue light, the three additive primary colors used in displays. Colors in this space can alternatively be specified by hue, saturation, and brightness values.
- **DeviceCMYK** controls the concentrations of cyan, magenta, yellow, and black inks, the four subtractive process colors used in printing.
- **DeviceGray** controls the intensity of achromatic light, on a scale from black to white.

Although the notion of explicit color spaces is a LanguageLevel 2 feature, the operators for specifying colors in the **DeviceRGB** and **DeviceGray** color spaces—**setrgbcolor**, **sethsbcolor**, and **setgray**—are available in all LanguageLevels. The **setcmykcolor** operator is also supported by some (but not all) LanguageLevel 1 implementations.

DeviceRGB Color Space

Colors in the **DeviceRGB** color space can be specified according to either of two color models: *red-green-blue* (RGB) and *hue-saturation-brightness* (HSB). Each of these models can specify any reproducible color with three numeric parameters, but the numbers have different meanings in the two models. Example 4.6 shows different ways to select the **DeviceRGB** color space and a specific color within that space.

Example 4.6

```
[/DeviceRGB] setcolorspace red green blue setcolor
/DeviceRGB setcolorspace red green blue setcolor
red green blue setrgbcolor
hue saturation brightness sethsbcolor
```

In the RGB model, a color is described as a combination of the three additive primary colors (red, green, and blue) in particular concentrations. The intensity of each primary color is specified by a number in the range 0.0 to 1.0, where 0.0 denotes no contribution at all and 1.0 denotes maximum intensity of that color.

If all three primary colors have equal intensity, the perceived result theoretically is a pure gray on the scale from black to white. If the intensities are not all equal, the result is some color other than a pure gray.

In the HSB model, a color is described by a combination of three parameters called *hue*, *saturation*, and *brightness*:

- *Hue* corresponds to the property that is intuitively meant by the term "color," such as yellow or blue-green.
- *Saturation* indicates how pure the color is. A saturation of 0.0 means that none of the color's hue is visible; the result is a shade of gray. A saturation of 1.0 denotes a pure color, consisting entirely of the specified hue. Intermediate values represent a mixture between pure hue and pure gray.
- *Brightness* determines how light the color determined by the hue and saturation will be. A brightness of 0.0 is always black. A brightness of 1.0 denotes the lightest color that the given combination of hue and saturation can allow. (For example, pure red can never be as light as the brightest white, because it is missing two components.)

HSB colors are often illustrated as arranged around a *color wheel.* The hue parameter determines the angular position of a color on this wheel, with 0.0 corresponding to pure red, 1/3 (0.333) to pure green, 2/3 (0.666) to pure blue, and 1.0 to red again. The saturation parameter denotes the color's radial position between the center of the wheel (saturation = 0.0) and the edge (saturation = 1.0). The brightness parameter controls the brightness of the colors displayed on the wheel itself.

Note: *HSB is not a color space in its own right. It is simply an alternative convention for specifying color values in the* ***DeviceRGB*** *color space.*

As shown in Example 4.6, the **setcolorspace** and **setcolor** operators select the color space and color value separately; **setrgbcolor** and **sethsbcolor** set them in combination. When the specified color space is **DeviceRGB**, **setcolorspace** sets the red, green, and blue components of the current color to 0.0.

When **DeviceRGB** is the current color space, both **currentcolor** and **currentrgbcolor** return the current color value in the form of its red, green, and blue components, regardless of how it was originally specified. **currenthsbcolor** returns the current color value as a hue, saturation, and brightness, converting among color models as necessary. When one of the other device color spaces (**DeviceCMYK** or **DeviceGray**) is current, **currentcolor** returns the current color value expressed in that space; **currentrgbcolor** and **currenthsbcolor** convert it to **DeviceRGB**. (The conversions are described in Section 7.2, "Conversions among Device Color Spaces.") These operators cannot convert from CIE-based or special color spaces.

Note: *Of the operators just described, only* ***setrgbcolor, sethsbcolor, currentrgbcolor,*** *and* ***currenthsbcolor*** *are supported in LanguageLevel 1.*

DeviceCMYK Color Space

The **DeviceCMYK** color space allows colors to be specified according to the subtractive CMYK model typical of printers and other paper-based output devices. Each color component in a **DeviceCMYK** color value specifies the amount of light that the corresponding ink or other colorant *absorbs*. In theory, each of the three standard *process colors* used in printing (cyan, magenta, and yellow) absorbs one of the additive primary colors (red, green, and blue, respectively). Black, a fourth standard process color, absorbs all additive primaries in equal amounts. Each of the four components in a CMYK color specification is a number between 0.0 and

1.0, where 0.0 represents no ink (that is, absorbs no light) and 1.0 represents the maximum quantity of ink (absorbs all the light it can). Note that the sense of these numbers is opposite to that of RGB color components.

Example 4.7 shows different ways to select the **DeviceCMYK** color space and a specific color within that space.

Example 4.7

```
[/DeviceCMYK] setcolorspace cyan magenta yellow black setcolor
/DeviceCMYK setcolorspace cyan magenta yellow black setcolor
cyan magenta yellow black setcmykcolor
```

The **setcolorspace** and **setcolor** operators select the color space and color value separately; **setcmykcolor** sets them in combination. When the specified color space is **DeviceCMYK**, **setcolorspace** sets the cyan, magenta, and yellow components of the current color to 0.0 and the black component to 1.0.

When **DeviceCMYK** is the current color space, both **currentcolor** and **currentcmykcolor** return the current color value in the form of its cyan, magenta, yellow, and black components. When one of the other device color spaces (**DeviceRGB** or **DeviceGray**) is current, **currentcolor** returns the current color value expressed in that space; **currentcmykcolor** converts it to **DeviceCMYK**. (The conversions are described in Section 7.2, "Conversions among Device Color Spaces.") This operator cannot convert from CIE-based or special color spaces.

Note: *The* ***setcmykcolor*** *and* ***currentcmykcolor*** *operators are supported by some, but not all, LanguageLevel 1 implementations.*

DeviceGray Color Space

Black, white, and intermediate shades of gray are special cases of full color. A grayscale value is represented by a single number in the range 0.0 to 1.0, where 0.0 corresponds to black, 1.0 to white, and intermediate values to different gray levels. Example 4.8 shows different ways to select the **DeviceGray** color space and a specific gray level within that space.

Example 4.8

```
[/DeviceGray] setcolorspace gray setcolor
/DeviceGray setcolorspace gray setcolor
gray setgray
```

The **setcolorspace** and **setcolor** operators select the color space and color value separately; **setgray** sets them in combination. When the specified color space is **DeviceGray**, **setcolorspace** sets the current color to 0.0.

When **DeviceGray** is the current color space, both **currentcolor** and **currentgray** return the current color value in the form of a single gray component. When one of the other device color spaces (**DeviceRGB** or **DeviceCMYK**) is current, **currentcolor** returns the current color value expressed in that space; **currentgray** converts it to **DeviceGray**. (The conversions are described in Section 7.2, "Conversions among Device Color Spaces.") This operator cannot convert from CIE-based or special color spaces.

Note: *The **setgray** and **currentgray** operators are supported by all PostScript implementations.*

4.8.3 CIE-Based Color Spaces

CIE-based color is defined relative to an international standard used in the graphic arts, television, and printing industries. It enables a page description to specify color values in a way that is related to human visual perception. The goal of this standard is for a given CIE-based color specification to produce consistent results on different output devices, up to the limitations of each device.

Note: *The detailed semantics of the CIE colorimetric system and the theory on which it is based are beyond the scope of this book. See the Bibliography for sources of further information.*

The semantics of the CIE-based color spaces are defined in terms of the relationship between the space's components and the tristimulus values *X*, *Y*, and *Z* of the CIE 1931 XYZ space. LanguageLevel 2 supports two CIE-based color space families, named **CIEBasedABC** and **CIEBasedA**; LanguageLevel 3 adds two more such families, **CIEBasedDEF** and **CIEBasedDEFG**. CIE-based color spaces are normally selected by

```
[name dictionary] setcolorspace
```

where *name* is the name of one of the CIE-based color space families and *dictionary* is a dictionary containing parameters that further characterize the color space. The entries in this dictionary have specific interpretations that vary depending on the color space; some entries are required and some are optional.

Having selected a color space, a PostScript program can then specify color values using the **setcolor** operator. Color values consist of a single component in a **CIEBasedA** color space, three components in a **CIEBasedABC** or **CIEBasedDEF** color space, and four components in a **CIEBasedDEFG** color space. The interpretation of these values varies depending on the specific color space.

Note: *To use any of the CIE-based color spaces with the* ***image*** *operator requires using the one-operand (dictionary) form of that operator, which interprets sample values according to the current color space. See Section 4.10.5, "Image Dictionaries."*

CIE-based color spaces are a feature of LanguageLevel 2 (**CIEBasedABC**, **CIEBasedA**) and LanguageLevel 3 (**CIEBasedDEF**, **CIEBasedDEFG**). Such spaces are entirely separate from device color spaces. Operators that refer to device color spaces implicitly, such as **setrgbcolor** and **currentrgbcolor**, have no connection with CIE-based color spaces; they do not perform conversions between CIE-based and device color spaces. (Note, however, that the PostScript interpreter may perform such conversions internally under the control of the **UseCIEColor** parameter in the page device dictionary; see "Remapping Device Colors to CIE" on page 237.) The **setrgbcolor** operator changes the color space to **DeviceRGB**. When the current color space is CIE-based, **currentrgbcolor** returns the initial value of the **DeviceRGB** color space, which has no relation to the current color in the graphics state.

CIEBasedABC Color Spaces

A **CIEBasedABC** color space *(LanguageLevel 2)* is defined in terms of a two-stage, nonlinear transformation of the CIE 1931 XYZ space. The formulation of **CIEBasedABC** color spaces models a simple *zone theory* of color vision, consisting of a nonlinear trichromatic first stage combined with a nonlinear opponent-color second stage. This formulation allows colors to be digitized with minimum loss of fidelity, an important consideration in sampled images.

The **CIEBasedABC** family includes a variety of interesting and useful color spaces, such as the CIE 1931 XYZ space, a class of calibrated RGB spaces, and a class of opponent-color spaces such as the CIE 1976 L*a*b* space and the NTSC, SECAM, and PAL television spaces.

Color values in **CIEBasedABC** color spaces have three components, arbitrarily named *A*, *B*, and *C*. They can represent a variety of independent color components, depending on how the space is parameterized. For example, *A*, *B*, and *C* may represent:

- *X*, *Y*, and *Z* in the CIE 1931 XYZ space
- *R*, *G*, and *B* in a calibrated RGB space
- *L**, *a**, and *b** in the CIE 1976 L*a*b* space
- *Y*, *I*, and *Q* in the NTSC television space
- *Y*, *U*, and *V* in the SECAM and PAL television spaces

The initial values of *A*, *B*, and *C* are 0.0 unless the range of valid values for a color component does not include 0.0, in which case the nearest valid value is substituted.

The parameters for a **CIEBasedABC** color space must be provided in a dictionary that is the second element of the array operand to the **setcolorspace** operator. Table 4.5 describes the contents of this dictionary; Figure 4.7 illustrates the transformations involved.

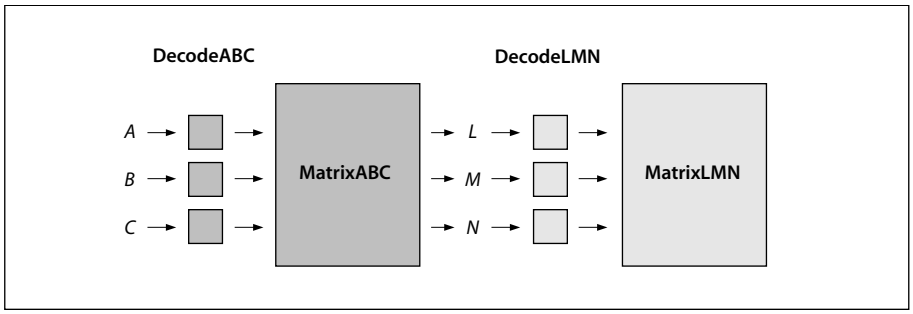


FIGURE 4.7 *Component transformations in the **CIEBasedABC** color space*

TABLE 4.5 Entries in a CIEBasedABC color space dictionary

KEY	TYPE	VALUE
RangeABC	array	*(Optional)* An array of six numbers $[A_0\ A_1\ B_0\ B_1\ C_0\ C_1]$ specifying the range of valid values for the *A*, *B*, and *C* components of the color space—that is, $A_0 \le A \le A_1$, $B_0 \le B \le B_1$, and $C_0 \le C \le C_1$. Component values falling outside the specified range will be adjusted to the nearest valid value without error indication. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].
DecodeABC	array	*(Optional)* An array of three PostScript procedures $[D_A\ D_B\ D_C]$ that decode the *A*, *B*, and *C* components of the color space into values that are linear with respect to an intermediate LMN representation; see **MatrixABC** below for further explanation. Default value: the array of identity procedures [{} {} {}]. Each of the three procedures is called with an encoded *A*, *B*, or *C* component on the operand stack and must return the corresponding decoded value. The result must be a monotonically nondecreasing function of the operand. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeABC** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.
MatrixABC	array	*(Optional)* An array of nine numbers $[L_A\ M_A\ N_A\ L_B\ M_B\ N_B\ L_C\ M_C\ N_C]$ specifying the linear interpretation of the decoded *A*, *B*, and *C* components of the color space with respect to the intermediate LMN representation. Default value: the identity matrix [1 0 0 0 1 0 0 0 1]. The transformation defined by the **DecodeABC** and **MatrixABC** entries is $L = D_A(A) \times L_A + D_B(B) \times L_B + D_C(C) \times L_C$ $M = D_A(A) \times M_A + D_B(B) \times M_B + D_C(C) \times M_C$ $N = D_A(A) \times N_A + D_B(B) \times N_B + D_C(C) \times N_C$ In other words, the *A*, *B*, and *C* components of the color space are first decoded individually by the **DecodeABC** procedures. The results are treated as a three-element vector and multiplied by **MatrixABC** (a 3-by-3 matrix) to obtain the *L*, *M*, and *N* components of the intermediate representation.
RangeLMN	array	*(Optional)* An array of six numbers $[L_0\ L_1\ M_0\ M_1\ N_0\ N_1]$ specifying the range of valid values for the *L*, *M*, and *N* components of the intermediate representation: that is, $L_0 \le L \le L_1$, $M_0 \le M \le M_1$, and $N_0 \le N \le N_1$. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].
DecodeLMN	array	*(Optional)* An array of three PostScript procedures $[D_L\ D_M\ D_N]$ that decode the *L*, *M*, and *N* components of the intermediate representation into values that are linear with respect to the CIE 1931 XYZ space; see **MatrixLMN** below

for further explanation. Default value: the array of identity procedures [{} {} {}].

Each of the three procedures is called with an encoded *L*, *M*, or *N* component on the operand stack and must return the corresponding decoded value. The result must be a monotonically nondecreasing function of the operand. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeLMN** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.

MatrixLMN array *(Optional)* An array of nine numbers $[X_L\ Y_L\ Z_L\ X_M\ Y_M\ Z_M\ X_N\ Y_N\ Z_N]$ specifying the linear interpretation of the decoded *L*, *M*, and *N* components of the intermediate representation with respect to the CIE 1931 XYZ space. Default value: the identity matrix [1 0 0 0 1 0 0 0 1].

The transformation defined by the **DecodeLMN** and **MatrixLMN** entries is

$$\begin{aligned} X &= D_L(L) \times X_L + D_M(M) \times X_M + D_N(N) \times X_N \\ Y &= D_L(L) \times Y_L + D_M(M) \times Y_M + D_N(N) \times Y_N \\ Z &= D_L(L) \times Z_L + D_M(M) \times Z_M + D_N(N) \times Z_N \end{aligned}$$

WhitePoint array *(Required)* An array of three numbers $[X_W\ Y_W\ Z_W]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse white point; see below for further explanation. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.

BlackPoint array *(Optional)* An array of three numbers $[X_B\ Y_B\ Z_B]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse black point; see below for further explanation. All three of these numbers must be nonnegative. Default value: [0.0 0.0 0.0].

The **WhitePoint** and **BlackPoint** entries in the color space dictionary control the overall effect of the CIE-based gamut mapping function described in Section 7.1, "CIE-Based Color to Device Color." Typically, the colors specified by **WhitePoint** and **BlackPoint** are mapped to the nearly lightest and nearly darkest achromatic colors that the output device is capable of rendering in a way that preserves color appearance and visual contrast.

WhitePoint is assumed to represent the diffuse achromatic highlight, not a specular highlight. Specular highlights, achromatic or otherwise, are often reproduced lighter than the diffuse highlight. **BlackPoint** is assumed to represent the diffuse achromatic shadow; its value is typically limited by the dynamic range of the in-

put device. In images produced by a photographic system, the values of **WhitePoint** and **BlackPoint** vary with exposure, system response, and artistic intent; hence, their values are image-dependent.

The following PostScript program fragments illustrate various interesting and useful special cases of **CIEBasedABC**. Example 4.9 establishes the CIE 1931 XYZ space with the CCIR XA/11–recommended D65 white point.

Example 4.9

```
[   /CIEBasedABC
      <<    /RangeABC [0.0  0.9505  0.0  1.0  0.0  1.0890]
            /RangeLMN [0.0  0.9505  0.0  1.0  0.0  1.0890]
            /WhitePoint [0.9505  1.0  1.0890]
      >>
] setcolorspace
```

Example 4.10 establishes the sRGB color space found in the IEC 61966 Standard (see Bibliography). It uses the ITU-R BT.709-2 reference primaries and the CCIR XA/11–recommended D65 white point.

Example 4.10

```
[   /CIEBasedABC
      <<    /DecodeLMN
               [   {    dup  0.03928  le
                           {12.92321  div}
                           {0.055  add  1.055  div  2.4  exp}
                        ifelse
                   } bind dup dup
               ]

            /MatrixLMN  [0.412457  0.212673  0.019334
                         0.357576  0.715152  0.119192
                         0.180437  0.072175  0.950301]

            /WhitePoint [0.9505  1.0  1.0890]
      >>
] setcolorspace
```

In many cases, the parameters of calibrated RGB color spaces are specified in terms of the CIE 1931 chromaticity coordinates (x_R, y_R), (x_G, y_G), (x_B, y_B) of the red, green, and blue phosphors, respectively, and the chromaticity (x_W, y_W) of the diffuse white point corresponding to some linear RGB value (R, G, B), where usually $R = G = B = 1.0$. Note that standard CIE notation uses lowercase letters to specify chromaticity coordinates and uppercase letters to specify tristimulus values. Given this information, **MatrixLMN** and **WhitePoint** can be found as follows:

$$z = y_W \times ((x_G - x_B) \times y_R - (x_R - x_B) \times y_G + (x_R - x_G) \times y_B)$$

$$Y_L = \frac{y_R}{R} \times \frac{(x_G - x_B) \times y_W - (x_W - x_B) \times y_G + (x_W - x_G) \times y_B}{z}$$

$$X_L = Y_L \times \frac{x_R}{y_R} \qquad Z_L = Y_L \times \left(\frac{1 - x_R}{y_R} - 1 \right)$$

$$Y_M = -\frac{y_G}{G} \times \frac{(x_R - x_B) \times y_W - (x_W - x_B) \times y_R + (x_W - x_R) \times y_B}{z}$$

$$X_M = Y_M \times \frac{x_G}{y_G} \qquad Z_M = Y_M \times \left(\frac{1 - x_G}{y_G} - 1 \right)$$

$$Y_N = \frac{y_B}{B} \times \frac{(x_R - x_G) \times y_W - (x_W - x_G) \times y_R + (x_W - x_R) \times y_G}{z}$$

$$X_N = Y_N \times \frac{x_B}{y_B} \qquad Z_N = Y_N \times \left(\frac{1 - x_B}{y_B} - 1 \right)$$

$$X_W = R \times X_L + G \times X_M + B \times X_N$$
$$Y_W = R \times Y_L + G \times Y_M + B \times Y_N$$
$$Z_W = R \times Z_L + G \times Z_M + B \times Z_N$$

Example 4.11 establishes the CIE 1976 L*a*b* space with the CCIR XA/11–recommended D65 white point. The a^* and b^* components, although theoretically unbounded, are defined to lie in the useful range −128 to +127. The transformation from L^*, a^*, and b^* component values to tristimulus values X, Y, and Z in the CIE 1931 XYZ space is defined as

$$X = X_W \times g\left(\frac{L^* + 16}{116} + \frac{a^*}{500}\right)$$

$$Y = Y_W \times g\left(\frac{L^* + 16}{116}\right)$$

$$Z = Z_W \times g\left(\frac{L^* + 16}{116} - \frac{b^*}{200}\right)$$

where the function $g(x)$ is defined as

$$g(x) = x^3 \qquad \text{if } x \geq \frac{6}{29}$$

$$g(x) = \frac{108}{841} \times \left(x - \frac{4}{29}\right) \qquad \text{otherwise}$$

Example 4.11

```
[   /CIEBasedABC
      <<    /RangeABC [0 100 −128 127 −128 127]

            /DecodeABC
               [   {16 add 116 div} bind
                   {500 div} bind
                   {200 div} bind
               ]

            /MatrixABC  [1  1  1
                         1  0  0
                         0  0 −1]

            /DecodeLMN
               [   {   dup 6 29 div ge
                              {dup dup mul mul}
                              {4 29 div sub 108 841 div mul}
                           ifelse
                       0.9505 mul
                   } bind
```

```
                {   dup 6 29 div ge
                        {dup dup mul mul}
                        {4 29 div sub 108 841 div mul}
                    ifelse
                } bind

                {   dup 6 29 div ge
                        {dup dup mul mul}
                        {4 29 div sub 108 841 div mul}
                    ifelse
                  1.0890 mul
                } bind
            ]

        /WhitePoint [ 0.9505  1.0  1.0890 ]
    >>
] setcolorspace
```

CIEBasedA Color Spaces

The **CIEBasedA** color space family *(LanguageLevel 2)* is the one-dimensional (and usually achromatic) analog of **CIEBasedABC**. Color values in **CIEBasedA** have a single component, arbitrarily named *A*. It can represent a variety of color components, depending on how the space is parameterized. For example, *A* may represent:

- The luminance *Y* component of the CIE 1931 XYZ space
- The gray component of a calibrated gray space
- The CIE 1976 psychometric lightness L^* component of the CIE 1976 L*a*b* space
- The luminance *Y* component of the NTSC, SECAM, and PAL television spaces

The initial value of *A* is 0.0 unless the range of valid values does not include 0.0, in which case the nearest valid value is substituted.

The parameters for a **CIEBasedA** color space must be provided in a dictionary that is the second element of the array operand to the **setcolorspace** operator. Table 4.6 describes the contents of this dictionary; Figure 4.8 illustrates the transformations involved.

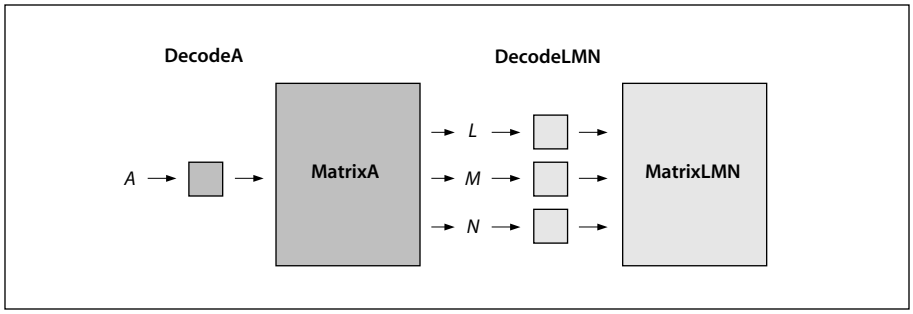


FIGURE 4.8 *Component transformations in the **CIEBasedA** color space*

TABLE 4.6 Entries in a CIEBasedA color space dictionary

KEY	TYPE	VALUE
RangeA	array	*(Optional)* An array of two numbers $[A_0\ A_1]$ specifying the range of valid values for the A component of the color space—that is, $A_0 \le A \le A_1$. Component values falling outside the specified range will be adjusted to the nearest valid value without error indication. Default value: [0.0 1.0].
DecodeA	procedure	*(Optional)* A PostScript procedure D_A that decodes the A component of the color space into a value that is linear with respect to an intermediate LMN representation; see **MatrixA** below for further explanation. Default value: the identity procedure {}. The procedure is called with an encoded A component on the operand stack and must return the corresponding decoded value. The result must be a monotonically nondecreasing function of the operand. The procedure must be prepared to accept operand values outside the range specified by the **RangeA** entry and to deal with such values in a robust way. Because this procedure is called at unpredictable times and in unpredictable environments, it must operate as a pure function without side effects.
MatrixA	array	*(Optional)* An array of three numbers $[L_A\ M_A\ N_A]$ specifying the linear interpretation of the decoded A component of the color space with respect to the intermediate LMN representation. Default value: the matrix [1 1 1]. The transformation defined by the **DecodeA** and **MatrixA** entries is $L = D_A(A) \times L_A$ $M = D_A(A) \times M_A$ $N = D_A(A) \times N_A$

		In other words, the *A* component of the color space is first decoded by the **DecodeA** procedure. The result is then multiplied by **MatrixA** (a three-element vector) to obtain the *L*, *M*, and *N* components of the intermediate representation.
RangeLMN	array	*(Optional)* An array of six numbers $[L_0\ L_1\ M_0\ M_1\ N_0\ N_1]$ specifying the range of valid values for the *L*, *M*, and *N* components of the intermediate representation—that is, $L_0 \le L \le L_1$, $M_0 \le M \le M_1$, and $N_0 \le N \le N_1$. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].
DecodeLMN	array	*(Optional)* An array of three PostScript procedures $[D_L\ D_M\ D_N]$ that decode the *L*, *M*, and *N* components of the intermediate representation into values that are linear with respect to the CIE 1931 XYZ space; see **DecodeLMN** and **MatrixLMN** in Table 4.5 for further explanation. Default value: the array of identity procedures [{} {} {}].
MatrixLMN	array	*(Optional)* An array of nine numbers $[X_L\ Y_L\ Z_L\ X_M\ Y_M\ Z_M\ X_N\ Y_N\ Z_N]$ specifying the linear interpretation of the decoded *L*, *M*, and *N* components of the intermediate representation with respect to the CIE 1931 XYZ space; see **MatrixLMN** in Table 4.5 for further explanation. Default value: the identity matrix [1 0 0 0 1 0 0 0 1].
WhitePoint	array	*(Required)* An array of three numbers $[X_W\ Y_W\ Z_W]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse white point; see the discussion following Table 4.5 for further explanation. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.
BlackPoint	array	*(Optional)* An array of three numbers $[X_B\ Y_B\ Z_B]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the diffuse black point; see the discussion following Table 4.5 for further explanation. All three of these numbers must be nonnegative. Default value: [0.0 0.0 0.0].

The following PostScript program fragments illustrate various interesting and useful special cases of **CIEBasedA**. Example 4.12 establishes a space consisting of the *Y* dimension of the CIE 1931 XYZ space with the CCIR XA/11–recommended D65 white point.

Example 4.12

```
[   /CIEBasedA
        <<      /MatrixA [0.9505  1.0  1.0890]
                /RangeLMN [0.0  0.9505  0.0  1.0  0.0  1.0890]
                /WhitePoint [0.9505  1.0  1.0890]
        >>
] setcolorspace
```

Example 4.13 establishes a calibrated gray space with the CCIR XA/11–recommended D65 white point and opto-electronic transfer function.

Example 4.13

```
[   /CIEBasedA
        <<      /DecodeA {1  0.45  div  exp} bind
                /MatrixA [0.9505  1.0  1.0890]
                /RangeLMN [0.0  0.9505  0.0  1.0  0.0  1.0890]
                /WhitePoint [0.9505  1.0  1.0890]
        >>
] setcolorspace
```

Example 4.14 establishes a space consisting of the L^* dimension of the CIE 1976 L*a*b* space with the CCIR XA/11–recommended D65 white point. See the discussion of Example 4.11 on page 227 for further explanation.

Example 4.14

```
[   /CIEBasedA
        <<      /RangeA [0  100]
                /DecodeA
                   {    16 add 116 div dup 6 29 div ge
                                {dup  dup  mul  mul}
                                {4  29  div  sub  108  841  div  mul}
                            ifelse
                   } bind
                /MatrixA [0.9505  1.0  1.0890]
                /RangeLMN [0.0  0.9505  0.0  1.0  0.0  1.0890]
                /WhitePoint [0.9505  1.0  1.0890]
        >>
] setcolorspace
```

CIEBasedDEF and CIEBasedDEFG Color Spaces

The **CIEBasedDEF** and **CIEBasedDEFG** color space families *(LanguageLevel 3)* extend the PostScript language's CIE-based color capabilities to support additional color spaces, including:

- The CIE 1976 L*u*v space
- Calibrated RGB from scanners
- Calibrated CMYK

The first two of these are three-component spaces and can be represented by **CIEBasedDEF** color spaces; the last has four components and can be represented by a **CIEBasedDEFG** space.

Both **CIEBasedDEF** and **CIEBasedDEFG** are simple pre-extensions to the **CIEBasedABC** color space family. Figure 4.9 illustrates the relationship between **CIEBasedDEFG** and **CIEBasedABC**. The components (*D*, *E*, *F*, *G*) of a color in a **CIEBasedDEFG** space are first transformed by applying a PostScript decoding procedure, **DecodeDEFG**; the resulting values are then used to look up and interpolate in a four-dimensional mapping table. The table lookup yields a color value with components *A*, *B*, and *C*, which is then mapped into the CIE 1931 XYZ space in the same way as for a **CIEBasedABC** color space; see "CIEBasedABC Color Spaces" on page 221. (The equivalent diagram for a three-component **CIEBasedDEF** space would look the same as Figure 4.9, except that the lookup table would have three inputs instead of four and the initial decoding procedure would be named **DecodeDEF** instead of **DecodeDEFG**.)

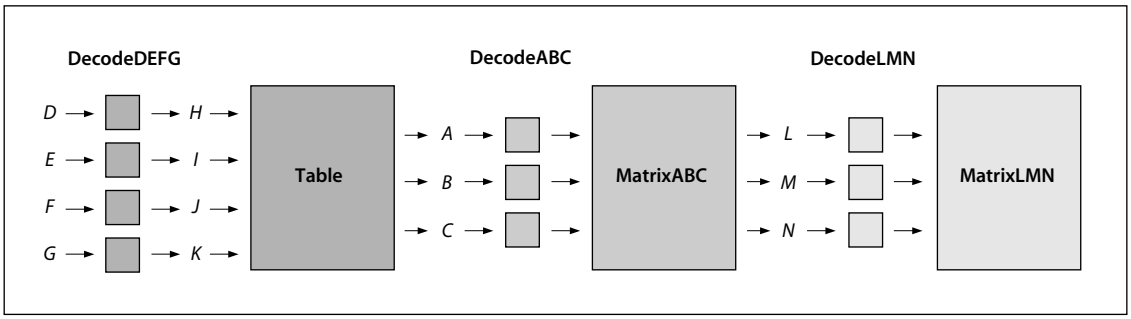


FIGURE 4.9 ***CIEBasedDEFG** pre-extension to the **CIEBasedABC** color space*

The parameters for a **CIEBasedDEF** or **CIEBasedDEFG** color space must be provided in a dictionary that is the second element of the array operand to the **setcolorspace** operator. This dictionary must contain all the same entries required for a **CIEBasedABC** space, as listed in Table 4.5 on page 223, and may also include any of the entries listed there as optional. Tables 4.7 and 4.8 show the additional dictionary entries, both required and optional, specific to **CIEBasedDEF** and **CIEBasedDEFG** color spaces, respectively. When one of these spaces is selected with the **setcolorspace** operator, the initial values of the color components *D*, *E*, and *F* (or *D*, *E*, *F*, and *G*) are set to 0.0 unless the range of valid values for a component does not include 0.0, in which case the nearest valid value is substituted.

TABLE 4.7 Additional entries specific to a CIEBasedDEF color space dictionary

KEY	TYPE	VALUE
RangeDEF	array	*(Optional)* An array of six numbers [D_0 D_1 E_0 E_1 F_0 F_1] specifying the range of valid values for the *D*, *E*, and *F* components of the color space—that is, $D_0 \le D \le D_1$, $E_0 \le E \le E_1$, and $F_0 \le F \le F_1$. Component values falling outside the specified range will be adjusted to the nearest valid value without error indication. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].
DecodeDEF	array	*(Optional)* An array of three PostScript procedures [D_D D_E D_F] that decode the *D*, *E*, and *F* components of the color space into intermediate values *H*, *I*, and *J*, respectively, that are more suitable for performing a table lookup. Default value: the array of identity procedures [{} {} {}]. Each of the three procedures is called with an encoded *D*, *E*, or *F* component on the operand stack and must return the corresponding decoded value *H*, *I*, or *J*. The result must be a monotonically nondecreasing function of the operand and must be within the range for the corresponding component, as specified in the **RangeHIJ** entry. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeDEF** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.
RangeHIJ	array	*(Optional)* An array of six numbers [H_0 H_1 I_0 I_1 J_0 J_1] specifying the range of valid values for the table lookup components *H*, *I*, and *J*—that is, $H_0 \le H \le H_1$, $I_0 \le I \le I_1$, and $J_0 \le J \le J_1$. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].

Table array *(Required)* An array of the form [$N_H\, N_I\, N_J\, table$] defining a three-dimensional lookup table that maps colors in the intermediate HIJ color space into a three-dimensional target space with components *A*, *B*, and *C*. The ABC space is then mapped into the CIE 1931 XYZ space in the same way as for a **CIEBasedABC** color space, and guided by the same dictionary entries; see Table 4.5 on page 223 for further explanation.

The first three elements of the **Table** array, N_H, N_I, and N_J, must be integers greater than 1 specifying the dimensions of the lookup table. The table contains $N_H \times N_I \times N_J$ entries, each consisting of three components making up an ABC color value.

The fourth array element, *table*, holds the contents of the lookup table itself in the form of an array of N_H strings, each containing $3 \times N_I \times N_J$ bytes. Like all PostScript arrays, the *table* array is indexed from 0; index *h* thus varies from 0 to $N_H - 1$, index *i* from 0 to $N_I - 1$, and index *j* from 0 to $N_J - 1$. The table entry for coordinates (h, i, j) is found in the string at index *h* in the lookup table, in the 3 bytes starting at position $3 \times (i \times N_J + j)$. This entry corresponds to the following color value in the HIJ space:

$$H = H_0 + \frac{h \times (H_1 - H_0)}{N_H - 1}$$

$$I = I_0 + \frac{i \times (I_1 - I_0)}{N_I - 1}$$

$$J = J_0 + \frac{j \times (J_1 - J_0)}{N_J - 1}$$

The limiting values H_0, H_1, I_0, I_1, J_0, and J_1 are specified by the color space dictionary's **RangeHIJ** entry.

The ABC component values corresponding to a given color in HIJ space are computed by locating the nearest adjacent table entries and then interpolating among the encoded byte values contained in those entries. The resulting interpolated, encoded components are mapped linearly to the range of valid values for the corresponding ABC component, as defined by the dictionary's **RangeABC** entry (see Table 4.5 on page 223). For example, a byte value of 0 for the *A* color component denotes a component value of A_0 as defined by **RangeABC**, a byte value of 255 denotes a component value of A_1, and similarly for the *B* and *C* components.

TABLE 4.8 Additional entries specific to a CIEBasedDEFG color space dictionary

KEY	TYPE	VALUE
RangeDEFG	array	*(Optional)* An array of eight numbers [D_0 D_1 E_0 E_1 F_0 F_1 G_0 G_1] specifying the range of valid values for the *D*, *E*, *F*, and *G* components of the color space—that is, $D_0 \le D \le D_1$, $E_0 \le E \le E_1$, $F_0 \le F \le F_1$, and $G_0 \le G \le G_1$. Component values falling outside the specified range will be adjusted to the nearest valid value without error indication. Default value: [0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0].
DecodeDEFG	array	*(Optional)* An array of four PostScript procedures [D_D D_E D_F D_G] that decode the *D*, *E*, *F*, and *G* components of the color space into intermediate values *H*, *I*, *J*, and *K*, respectively, that are more suitable for performing a table lookup. Default value: the array of identity procedures [{} {} {} {}]. Each of the four procedures is called with an encoded *D*, *E*, *F*, or *G* component on the operand stack and must return the corresponding decoded value *H*, *I*, *J*, or *K*. The result must be a monotonically nondecreasing function of the operand and must be within the range for the corresponding component, as specified in the **RangeHIJK** entry. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeDEFG** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.
RangeHIJK	array	*(Optional)* An array of eight numbers [H_0 H_1 I_0 I_1 J_0 J_1 K_0 K_1] specifying the range of valid values for the table lookup components *H*, *I*, *J*, and *K*—that is, $H_0 \le H \le H_1$, $I_0 \le I \le I_1$, $J_0 \le J \le J_1$, and $K_0 \le K \le K_1$. Default value: [0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0].
Table	array	*(Required)* An array of the form [N_H N_I N_J N_K *table*] defining a four-dimensional lookup table that maps colors in the intermediate HIJK color space into a three-dimensional target space with components *A*, *B*, and *C*. The ABC space is then mapped into the CIE 1931 XYZ space in the same way as for a **CIEBasedABC** color space, and guided by the same dictionary entries; see Table 4.5 on page 223 for further explanation. The first four elements of the **Table** array, N_H, N_I, N_J, and N_K, must be integers greater than 1 specifying the dimensions of the lookup table. The table contains $N_H \times N_I \times N_J \times N_K$ entries, each of which consists of three components making up an ABC color value. The fifth array element, *table*, holds the contents of the lookup table itself in the form of an array of N_H arrays, each in turn containing N_I strings of $3 \times N_J \times N_K$ bytes. Like all PostScript arrays, the *table* array and each of the arrays that are its elements are indexed from 0; index *h* thus varies from 0 to

$N_H - 1$, index i from 0 to $N_I - 1$, and so on. The table entry for coordinates (h, i, j, k) is found in the array at index h in the lookup table, in the ith string of the array, in the 3 bytes starting at position $3 \times (j \times N_K + k)$. This entry corresponds to the following color value in the HIJK space:

$$H = H_0 + \frac{h \times (H_1 - H_0)}{N_H - 1}$$

$$I = I_0 + \frac{i \times (I_1 - I_0)}{N_I - 1}$$

$$J = J_0 + \frac{j \times (J_1 - J_0)}{N_J - 1}$$

$$K = K_0 + \frac{k \times (K_1 - K_0)}{N_K - 1}$$

The limiting values H_0, H_1, I_0, I_1, J_0, J_1, K_0, and K_1 are specified by the color space dictionary's **RangeHIJK** entry.

The ABC component values corresponding to a given color in HIJK space are computed by locating the nearest adjacent table entries and then interpolating among the encoded byte values contained in those entries. The resulting interpolated, encoded components are mapped linearly to the range of valid values for the corresponding ABC component, as defined by the dictionary's **RangeABC** entry (see Table 4.5 on page 223). For example, a byte value of 0 for the A color component denotes a component value of A_0 as defined by **RangeABC**, a byte value of 255 denotes a component value of A_1, and similarly for the B and C components.

Example 4.15 illustrates in schematic form the use of a **CIEBasedDEFG** color space.

Example 4.15

```
[   /CIEBasedDEFG
      <<    /DecodeLMN [{1  0.45  div  exp} bind  dup  dup]

            /MatrixLMN  [0.4124  0.2126  0.0193
                         0.3576  0.7152  0.1192
                         0.1805  0.0722  0.9505]

            /WhitePoint [0.9505  1.0  1.0890]
```

```
            /Table [15  30  25  20
                    [   [   < … String containing 3×25×20 bytes … >
                            < … String containing 3×25×20 bytes … >
                            … 28 additional strings omitted …
                        ]
                        … 14 additional arrays omitted …
                    ]
                ]
        >>
    ] setcolorspace
```

Remapping Device Colors to CIE

Specifying colors in a device color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**) makes them device-dependent. By setting the **UseCIEColor** parameter *(LanguageLevel 3)* in the page device dictionary (see Section 6.2.5, "Color Support"), a PostScript program can request that such colors be systematically transformed into a device-independent CIE-based color space. This capability can be useful in a variety of circumstances, such as the following:

- A page description originally intended for one device is redirected to a different device.
- An application generates LanguageLevel 1 output only, and thus is unable to specify CIE-based colors directly. This is especially common for encapsulated PostScript (EPS) files.
- Color corrections or rendering intents need to be applied to device colors. (Rendering intents allow the application to set priorities regarding which color properties to preserve and which to compromise in order to compensate for the limitations of a particular output device; see Section 7.1.3, "Rendering Intents," for details.)

When **UseCIEColor** is *true*, all colors specified in the **DeviceGray**, **DeviceRGB**, or **DeviceCMYK** color space are remapped into a corresponding target color space, which must previously have been defined as a resource in the **ColorSpace** resource category. This substitution occurs at the moment a device color space is selected by **setcolorspace** (or implicitly by **setgray**, **setrgbcolor**, or **setcmykcolor**); it is unaffected by subsequent changes to the **ColorSpace** resource category. The resources specifying the target spaces for the three device color spaces are named

DefaultGray, **DefaultRGB**, and **DefaultCMYK**, respectively. Their values must be as follows; if they are absent, an error will occur.

- **DefaultGray** must be a **CIEBasedA** color space or [/DeviceGray] (for no remapping).
- **DefaultRGB** must be a **CIEBasedABC** color space, a **CIEBasedDEF** color space, or [/DeviceRGB] (for no remapping).
- **DefaultCMYK** must be a **CIEBasedDEFG** color space or [/DeviceCMYK] (for no remapping).

If the color space in use is a special color space based on an underlying device color space, **UseCIEColor** will remap the underlying space. This applies to the following:

- The base color space of an **Indexed** color space
- The underlying color space of a **Pattern** color space
- The alternative color space of a **Separation** or **DeviceN** color space (but only if the alternative color space is actually selected)

See Section 4.8.4, "Special Color Spaces," for details on these color spaces.

Enabling **UseCIEColor** does not alter the current color space or current color values in the graphics state. The remapping of device colors into CIE-based colors is entirely internal to the implementation of the PostScript color operators. Once transformed, the colors are then processed according to the current color rendering dictionary, as is normally done for CIE-based colors. See Section 7.1, "CIE-Based Color to Device Color," for more information.

4.8.4 Special Color Spaces

Special color spaces add features or properties to an underlying color space. There are four special color space families: **Pattern**, **Indexed**, **Separation**, and **DeviceN**.

Pattern Color Space

A **Pattern** color space *(LanguageLevel 2)* enables PostScript programs to paint an area with a "color" defined as a *pattern*, which may be either a graphical figure re-

peated indefinitely to fill the area *(LanguageLevel 2)* or a *gradient fill* defining a smooth color transition across the area *(LanguageLevel 3)*. Section 4.9, "Patterns," discusses patterns in detail.

Indexed Color Space

An **Indexed** color space *(LanguageLevel 2)* allows a PostScript program to select from a *color map* or *color table* of arbitrary colors in some other space, using small integers as indices. The PostScript interpreter treats each sample value as an index into the color table and uses the color value it finds there. This technique can considerably reduce the amount of data required to represent a sampled image—for example, by using 8-bit index values as samples instead of 24-bit RGB color values.

An **Indexed** color space is selected as follows:

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

In other words, the operand to **setcolorspace** is a four-element array. The first element is the color space family name **Indexed**; the remaining elements are the parameters *base*, *hival*, and *lookup*, which the **Indexed** color space requires. **setcolorspace** sets the current color to 0.

The *base* parameter is an array or name that identifies the *base color space* in which the values in the color table are to be interpreted. It can be any device or CIE-based color space or (in LanguageLevel 3) a **Separation** or **DeviceN** space, but not a **Pattern** or another **Indexed** space. For example, if the base color space is **DeviceRGB**, the values in the color table are to be interpreted as red, green, and blue components; if the base color space is a **CIEBasedABC** space, the values are to be interpreted as *A*, *B*, and *C* components. The *base* parameter should be specified in the same way as if it were being used directly as an operand to the **setcolorspace** operator.

Note: *Attempting to use a* ***Separation*** *or* ***DeviceN*** *color space as the base for an* ***Indexed*** *color space will generate an* ***undefined*** *error in LanguageLevel 2.*

The *hival* parameter is an integer that specifies the maximum valid index value. In other words, the color table is to be indexed by integers in the range 0 to *hival*. *hival* can be no greater than 4095, which is what would be required to index a table with 12-bit color sample values.

The color table is defined by the *lookup* parameter, which can be either a procedure or a string. It provides the mapping between index values and the corresponding colors in the base color space.

If *lookup* is a procedure, the PostScript interpreter calls it to transform an index value into corresponding color component values in the base color space. The procedure is called with the index on the operand stack and must return the color component values in a form acceptable to the **setcolor** operator in the base color space. The number of components and the interpretation of their values depend on the base color space. Because the *lookup* procedure is called by the **setcolor** and **image** operators at unpredictable times, it must operate as a pure function without side effects. It must be able to return color component values for any integer from 0 to *hival*.

If *lookup* is a string object, it must be of length $m \times (hival + 1)$, where m is the number of color components in the base color space. Each byte in the string is interpreted as an integer. To look up an index, the PostScript interpreter multiplies the index by m and uses the result to access the *lookup* string. The m bytes beginning at that position in the string are interpreted as coded values for the m color components of the base color space. Those bytes are treated as 8-bit integers in the range 0 to 255, which are then divided by 255, yielding component values in the range 0.0 to 1.0.

Note: *This method of specification is useful only when the range of all color components in the base color space is 0.0 to 1.0. For color spaces with different ranges, such as a* ***CIEBasedABC*** *space representing L*a*b* (see Example 4.11 on page 227),* ***lookup*** *should be a procedure.*

Example 4.16 illustrates the specification of an **Indexed** color space that maps 8-bit index values to three-component color values in the **DeviceRGB** color space.

Example 4.16

```
[   /Indexed
      /DeviceRGB 255
      <000000 FF0000 00FF00 0000FF B57342 … >
] setcolorspace
```

The example shows only the first five color values in the *lookup* string; in all, there should be 256 color values and the string should be 768 bytes long. Having established this color space, the program can now specify colors using single-

component values in the range 0 to 255. For example, a color value of 4 selects an RGB color whose components are coded as the hexadecimal integers B5, 73, and 42. Dividing these by 255 yields a color whose red, green, and blue components are 0.710, 0.451, and 0.259, respectively.

Note: *To use an* ***Indexed*** *color space with the* ***image*** *operator requires using the one-operand (dictionary) form of that operator, which interprets sample values according to the current color space. See Section 4.10.5, "Image Dictionaries."*

Although an **Indexed** color space is useful mainly for images, index values can also be used with the **setcolor** operator. For example,

```
123 setcolor
```

selects the same color as does an image sample value of 123. The index value should be an integer in the range 0 to *hival*. If it is a real number, it is rounded to the nearest integer; if it is outside the range 0 to *hival*, it is clipped to the nearest bound.

Separation Color Spaces

Color output devices produce full color by combining *primary* or *process colors* in varying amounts. On a display, the primary colors consist of red, green, and blue phosphors; on a printer, they consist of cyan, magenta, yellow, and sometimes black inks. In addition, some devices can apply special colorants, often called *spot colors*, to produce effects that cannot be achieved with the primary colors alone. Examples include metallic and fluorescent colors and special textures.

When the **showpage** or **copypage** operator is invoked, most devices produce a single *composite* page on which all primary colors (and spot colors, if any) are combined. However, some devices, such as imagesetters, produce a separate, monochromatic rendition of the page, called a *separation*, for each individual colorant. When the separations are later combined—on a printing press, for example—and the proper inks or other colorants are applied to them, a full-color page results.

A **Separation** color space *(LanguageLevel 2)* provides a means for PostScript programs to specify the use of additional colorants or to isolate the control of individual color components of a device color space. When such a space is the current

color space, the current color is a single-component value, called a *tint*, that controls the application of the given colorant or color component only.

Note: *The term* separation *is often misused as a synonym for an individual device colorant. In the context of this discussion, a printing system that produces separations generates a separate piece of physical medium (generally film) for each colorant. It is these pieces of physical medium that are correctly referred to as separations. A particular colorant properly constitutes a separation only if the device is generating physical separations, one of which corresponds to the given colorant. The* ***Separation*** *color space is so named for historical reasons, but it has evolved to the broader purpose of controlling the application of individual colorants in general, whether or not they are actually realized as physical separations.*

Note also that the operation of a ***Separation*** *color space itself is independent of the characteristics of any particular output device. Depending on the device, the space may or may not correspond to a true, physical separation or to an actual colorant. For example, a* ***Separation*** *color space could be used to control the application of a single process colorant (such as cyan) on a composite device that does not produce physical separations, or could represent a color (such as orange) for which no specific colorant exists on the device. A* ***Separation*** *color space provides consistent, predictable behavior, even on devices that cannot directly generate the requested color.*

A **Separation** color space is selected as follows:

[/Separation *name alternativeSpace tintTransform*] setcolorspace

In other words, the operand to **setcolorspace** is a four-element array whose first element is the color space family name **Separation**. The remaining elements are parameters that a **Separation** color space requires; their meanings are discussed below.

A color value in a **Separation** color space consists of a single tint component in the range 0.0 to 1.0. The value 0.0 represents the minimum amount of colorant that can be applied; 1.0 represents the maximum. Tints are always treated as *subtractive* colors, even if the device produces output for the designated component by an additive method. Thus a tint value of 0.0 denotes the lightest color that can be achieved with the given colorant, and 1.0 the darkest. (Note that this is the same as the convention for **DeviceCMYK** color components, but opposite to the one for **DeviceRGB** and **DeviceGray**.) The **setcolor** operator sets the current color in the graphics state to a tint value; the initial value is 1.0. A sampled image with single-component samples can also be used as a source of tint values.

Note: *To use a* ***Separation*** *color space with the* ***image*** *operator requires using the one-operand (dictionary) form of that operator, which interprets sample values according to the current color space. See Section 4.10.5, "Image Dictionaries."*

The *name* parameter in the parameter array passed to **setcolorspace** is a name or string object specifying the name of the colorant that this **Separation** color space is intended to represent (or one of the special names **All** or **None**; see below). Such colorant names are arbitrary, and there can be any number of them, subject to implementation limits. Name and string objects can be used interchangeably as names; a string may be more convenient if the desired name contains spaces or other special characters.

The set of available colorant names is determined from the **ProcessColorModel** and **SeparationColorNames** entries in the page device dictionary (Section 6.2.5, "Color Support"). **ProcessColorModel** defines the native color space of the device, and hence the set of available process colorants (**Cyan**, **Magenta**, **Yellow**, and **Black** for **DeviceCMYK**; **Red**, **Green**, and **Blue** for **DeviceRGB**; **Gray** for **DeviceGray**). **SeparationColorNames** lists additional available colorants.

Note: *In LanguageLevel 2 implementations lacking the* ***ProcessColorModel*** *and* ***SeparationColorNames*** *page device parameters, the set of available colorants is implicit and cannot be queried by a PostScript program.*

The special colorant name **All** refers collectively to all colorants available on a device, including those for the standard process colorants as well as any additional colorants named explicitly in the page device dictionary's **SeparationColorNames** entry. When a **Separation** space with this colorant name is the current color space, painting operators apply tint values to all available colorants at once. This is useful for purposes such as painting registration marks in the same place on every separation. A program would typically paint such marks as the last step in composing a page, immediately before invoking **showpage**, to ensure that they are not overwritten by subsequent drawing operations.

The special colorant name **None** will never produce any visible output. Painting operations in a **Separation** space with this colorant name have no effect on the current page.

All devices support **Separation** color spaces with the colorant names **All** and **None**, even if they do not support any others. **Separation** spaces with either of these colorant names ignore the *alternativeSpace* and *tintTransform* parameters (discussed below), though dummy values must still be provided.

At the moment the color space is set to a **Separation** space, the **setcolorspace** operator checks the **ProcessColorModel** and **SeparationColorNames** entries in the page device dictionary to determine whether the device has an available colorant corresponding to the name of the requested space. If so, **setcolorspace** ignores its *alternativeSpace* and *tintTransform* parameters; subsequent painting operations within the space will apply the designated colorant directly, according to the tint values supplied.

If the colorant name associated with a **Separation** color space does not correspond to a colorant available on the device, **setcolorspace** arranges instead for subsequent painting operations to be performed in an *alternative color space*. This enables the intended colors to be approximated by colors in some device or CIE-based color space, which are then rendered using the usual primary or process colors. This works as follows:

- The *alternativeSpace* parameter must be an array or name object that identifies the alternative color space. This can be any device or CIE-based color space, but not another special color space (**Pattern**, **Indexed**, **Separation**, or **DeviceN**). The *alternativeSpace* parameter should be specified in the same way as if it were being used directly as an operand to the **setcolorspace** operator.
- The *tintTransform* parameter must be a PostScript procedure. During subsequent painting operations, the PostScript interpreter will call this procedure to transform a tint value into color component values in the alternative color space. The procedure is called with the tint value on the operand stack and must return the corresponding color component values in a form acceptable to the **setcolor** operator in the alternative color space. The number of components and the interpretation of their values depend on the alternative color space. Because the *tintTransform* procedure is called by the **setcolor** and **image** operators at unpredictable times, it must operate as a pure function without side effects.

Example 4.17 illustrates the specification of a **Separation** color space that is intended to produce a color named LogoGreen. If the output device has no colorant corresponding to this color, **DeviceCMYK** will be used as the alternative color

space; the *tintTransform* procedure provided maps tint values linearly into shades of a CMYK color value approximating the "logo green" color.

Example 4.17

```
[   /Separation
        (LogoGreen)
        /DeviceCMYK
        {   dup 0.84 mul
            exch 0.0 exch dup 0.44 mul
            exch 0.21 mul
        }
] setcolorspace
```

DeviceN Color Spaces

DeviceN color spaces *(LanguageLevel 3)* support the use of high-fidelity and multitone color. *High-fidelity* color is the use of more than the standard CMYK process colorants to produce an extended *gamut*, or range of colors. A popular example of such a system is the PANTONE® Hexachrome™ system, which uses six colorants: the usual cyan, magenta, yellow, and black, plus orange and green.

Multitone color systems use a single-component image to specify multiple color components. In a *duotone*, for example, a single-component image can be used to specify both the black component and a spot color component. The tone reproduction is generally different for the different components; for example, the black component might be painted with the exact sample data from the single-component image, while the spot color component might be generated as a non-linear function of the image data in a manner that emphasizes the shadows.

DeviceN color spaces allow any subset of the available device colorants to be treated as a device color space with multiple components. This provides greater flexibility than is possible with standard device color spaces such as **DeviceCMYK** or with individual **Separation** color spaces. For example, it is possible to create a **DeviceN** color space consisting of only the cyan, magenta, and yellow color components, while excluding the black component. If overprinting is enabled (see Section 4.8.5, "Overprint Control"), painting in this color space will leave the black component unchanged.

A **DeviceN** color space is selected as follows:

[/DeviceN *names alternativeSpace tintTransform*] setcolorspace

In other words, the operand to **setcolorspace** is a four-element array whose first element is the color space family name **DeviceN**. The remaining elements are parameters that a **DeviceN** color space requires; their meanings are discussed below.

A **DeviceN** color space works almost the same as a **Separation** color space—in fact, a **DeviceN** color space with only one component is exactly equivalent to a **Separation** color space. The following are the only differences between **DeviceN** and **Separation**:

- Color values in a **DeviceN** color space consist of multiple tint components, rather than only one.
- The *names* parameter in the color space array passed to **setcolorspace** is an array of colorant names, specified as name or string objects. (The special colorant names **All** and **None** are not allowed.) The length of the array determines the number of components, and hence the number of operands required by the **setcolor** operator when this space is the current color space. Operand values supplied to **setcolor** are interpreted as color component values in the order in which the colors are given in the *names* array.
- The **setcolorspace** operator will select the requested set of colorants only if all of them are available on the device; otherwise, it will select the alternative color space designated by the *alternativeSpace* parameter.
- The *tintTransform* procedure is called with *n* tint values on the operand stack and must return the corresponding *m* color component values, where *n* is the number of components needed to specify a color in the **DeviceN** color space and *m* is the number required by the alternative color space.

The following PostScript program fragments illustrate various interesting and useful special cases of **DeviceN**. Example 4.18 shows the specification of an **Indexed** color space that maps 8-bit index values to a duotone **DeviceN** space in cyan and black.

Example 4.18

```
[   /Indexed
      [   /DeviceN
            [/Cyan /Black]
            /DeviceCMYK
            {0 0 3 –1 roll}
      ]
      255
      <6605 6806 6907 6B09 6C0A … >
] setcolorspace
```

Example 4.19 defines an **Indexed** color space that maps 8-bit index values to a tritone **DeviceN** space in magenta, yellow, and black. Because all of the corresponding magenta and yellow components in the lookup table are equal, the effect is equivalent to a duotone in red and black.

Example 4.19

```
[   /Indexed
      [   /DeviceN
            [/Magenta /Yellow /Black]
            /DeviceCMYK
            {0 4 1 roll}
      ]
      255
      <4C4C05 4E4E06 505007 525208 545409 … >
] setcolorspace
```

4.8.5 Overprint Control

The graphics state contains an *overprint* parameter, controlled by the **setoverprint** operator. Overprint control is useful mainly on devices that produce true physical separations, but it is available on some composite devices as well. Although the operation of this parameter is device-dependent, it is described here, rather than in the chapter on color rendering, because it pertains to an aspect of painting in device color spaces that is important to many applications.

Any painting operation marks some specific set of device colorants, depending on the color space in which the painting takes place. In a **Separation** or **DeviceN** color space, the colorants to be marked are specified explicitly; in a device or CIE-based space, they are implied by the process color model of the output device (see

"Process Color Model" on page 422). The overprint parameter is a boolean flag that determines how painting operations affect colorants other than those explicitly or implicitly specified by the current color space.

If the overprint flag is *false* (the default value), painting a color in any color space causes the corresponding areas of unspecified colorants to be erased (painted with a tint value of 0.0). The effect is that the color marked at any position on the page is whatever was painted there last; this is consistent with the normal opaque painting behavior of the Adobe imaging model.

If the overprint flag is *true* and the output device supports overprinting, no such erasing actions are performed; anything previously painted in other colorants is left undisturbed. Consequently, the color at a given position on the page may be a combined result of several painting operations in different colorants. The effect produced by such overprinting is device-dependent and is not defined by the PostScript language.

Note: *Not all devices support overprinting. Furthermore, many LanguageLevel 2 implementations support it only when separations are being produced, not for composite output. If overprinting is not supported, the value of the overprint parameter is ignored.*

4.9 Patterns

When operators such as **fill**, **stroke**, and **show** paint an area of the page with the current color, they ordinarily apply a single color that covers the area uniformly. Sometimes, however, it is desirable to apply "paint" that consists of a repeating figure or a smoothly varying color gradient instead of a simple color. Such a repeating figure or smooth gradient is called a *pattern.* Patterns are quite general, and have many uses. They can be used to create various graphical textures, such as weaves, brick walls, sunbursts, and similar geometrical and chromatic effects.

PostScript patterns come in two varieties:

- *Tiling patterns* consist of a small graphical figure (called a *pattern cell*) that is replicated at fixed horizontal and vertical intervals to fill the area to be painted.
- *Shading patterns* define a *gradient fill* that produces a smooth transition between colors across the area.

Note: *The ability to paint with patterns is a feature of LanguageLevels 2 (tiling patterns) and 3 (shading patterns). With some effort, it is possible to achieve a limited form of tiling patterns in LanguageLevel 1 by defining them as character glyphs in a special font and painting them repeatedly with the* ***show*** *operator. Another technique, defining patterns as halftone screens, is* not *recommended, because the effects produced are device-dependent.*

Patterns are specified in a special color space family named **Pattern**, whose "color values" are *pattern dictionaries* instead of the numeric component values used with other color spaces. This section describes **Pattern** color spaces and the specification of color values within them; see Section 4.8, "Color Spaces," for information about color spaces and color values in general.

4.9.1 Using Patterns

A pattern dictionary contains descriptive information defining the appearance and properties of a pattern. All pattern dictionaries contain an entry named **PatternType**, whose value identifies the kind of pattern the dictionary describes: type 1 denotes a tiling pattern, type 2 a shading pattern. The remaining contents of the dictionary depend on the pattern type, and are detailed below in the sections on each pattern type.

Painting with a pattern is a five-step procedure:

1. *Define the prototype pattern.* Create a pattern dictionary containing descriptive information about the pattern's appearance and other properties.
2. *Instantiate the pattern.* Pass the prototype pattern dictionary to the **makepattern** operator. This produces a copy of the dictionary representing an instance of the pattern that is locked to current user space. The copy may contain an optional additional entry, named **Implementation**, containing implementation-dependent information to be used by the interpreter in painting the pattern.
3. *Select a* ***Pattern*** *color space.* Use the **setcolorspace** operator to set the current color space to a **Pattern** space. The initial color value in this color space is a *null* object, which is treated as if it were a pattern dictionary representing an empty tiling pattern. Painting with this pattern produces no marks on the current page.

4. *Make the pattern the current color.* Invoke **setcolor** with the instantiated pattern dictionary from step 2 as an operand (and possibly other operands as well) to select the pattern as the current color.

5. *Invoke painting operators*, such as **fill**, **stroke**, **imagemask**, or **show**. All areas that normally would be painted with a uniform color will instead be filled with the selected pattern.

A convenience operator, **setpattern**, combines steps 3 and 4 above into a single operation: it takes a pattern dictionary as an operand, selects a **Pattern** color space, and sets the specified pattern as the current color. **setpattern** is the normal method for selecting patterns; in practice, it is rarely necessary to set the color space and color value separately. For purposes of exposition, however, the examples presented here will separate the two steps for maximum clarity.

4.9.2 Tiling Patterns

A tiling pattern consists of a small graphical figure called a *pattern cell*. Painting with the pattern replicates the cell at fixed horizontal and vertical intervals to fill an area. The effect is as if the figure were painted on the surface of a clear glass tile, identical copies of which were then laid down in an array covering the area and trimmed to its boundaries. This is called *tiling* the area.

The pattern cell can include graphical elements such as filled areas, text, and sampled images. Its shape need not be rectangular, and the spacing of tiles can differ from the dimensions of the cell itself. The cell's appearance is defined by an arbitrary PostScript procedure, the ***PaintProc*** *procedure*, which paints a single instance of the cell. The PostScript interpreter obtains the **PaintProc** procedure from the pattern dictionary and calls it (with the graphics state altered in certain ways) to obtain the pattern cell. When performing painting operations such as **fill** or **stroke**, the interpreter then paints the cell on the current page as many times as necessary to fill an area. To optimize execution, the interpreter maintains a cache of recently used pattern cells.

Tiling patterns are defined by pattern dictionaries of type 1. Table 4.9 lists the entries in this type of dictionary. (The dictionary can also contain any additional entries that its **PaintProc** procedure may require.) All entries except **Implementation** can appear in a prototype pattern dictionary supplied as an operand to **makepattern**. The pattern dictionary instantiated and returned by

makepattern may contain an **Implementation** entry in addition to those included in the prototype.

TABLE 4.9 Entries in a type 1 pattern dictionary

KEY	TYPE	VALUE
PatternType	integer	*(Required)* A code identifying the pattern type that this dictionary describes; must be 1 for a tiling pattern.
XUID	array	*(Optional)* An *extended unique ID* that uniquely identifies the pattern (see Section 5.6.2, "Extended Unique ID Numbers"). The presence of an **XUID** entry in a pattern dictionary enables the PostScript interpreter to save cached instances of the pattern for later use, even when the pattern dictionary is loaded into virtual memory multiple times (for instance, by different jobs). To ensure correct behavior, **XUID** values must be assigned from a central registry. This is particularly appropriate for patterns treated as named resources. Patterns that are created dynamically by an application program should *not* contain **XUID** entries.
PaintProc	procedure	*(Required)* A PostScript procedure for painting the pattern cell.
BBox	array	*(Required)* An array of four numbers in the pattern coordinate system, giving the coordinates of the left, bottom, right, and top edges, respectively, of the pattern cell's bounding box. These boundaries are used to clip the pattern cell and to determine its size for caching.
XStep	number	*(Required)* The desired horizontal spacing between pattern cells, measured in the pattern coordinate system.
YStep	number	*(Required)* The desired vertical spacing between pattern cells, measured in the pattern coordinate system. Note that **XStep** and **YStep** may differ from the dimensions of the pattern cell implied by the **BBox** entry. This allows tiling with irregularly shaped figures. **XStep** and **YStep** may be either positive or negative, but not zero.
PaintType	integer	*(Required)* A code that determines how the color of the pattern cell is to be specified: 1 *Colored tiling pattern*. The **PaintProc** procedure itself specifies the colors used to paint the pattern cell. When the **PaintProc** procedure begins execution, the current color is the one that was in effect at the time the tiling pattern was instantiated with **makepattern**. 2 *Uncolored tiling pattern*. The **PaintProc** procedure does not specify any color information. Instead, the entire pattern cell is painted with a separately specified color each time the tiling pattern is used. Essen-

		tially, the **PaintProc** procedure describes a *stencil* through which the current color is to be poured. The **PaintProc** procedure must not invoke operators that specify colors or other color-related parameters in the graphics state; otherwise, an **undefined** error will occur (see Section 4.8.1, "Types of Color Space"). Use of the **imagemask** operator is permitted, however, since it does not specify any color information.
TilingType	integer	*(Required)* A code that controls adjustments to the spacing of tiles relative to the device pixel grid: 1 *Constant spacing*. Pattern cells are spaced consistently—that is, by a multiple of a device pixel. To achieve this, **makepattern** may need to distort the pattern cell slightly by making small adjustments to **XStep**, **YStep**, and the transformation matrix. The amount of distortion does not exceed 1 device pixel. 2 *No distortion*. The pattern cell is not distorted, but the spacing between pattern cells may vary by as much as 1 device pixel, both horizontally and vertically, when the tiling pattern is painted. This achieves the spacing requested by **XStep** and **YStep** *on average*, but not for each individual pattern cell. 3 *Constant spacing and faster tiling*. Pattern cells are spaced consistently as in tiling type 1, but with additional distortion permitted to enable a more efficient implementation.
Implementation	any	An additional entry inserted in the dictionary by the **makepattern** operator, containing information used by the interpreter to achieve proper tiling of the pattern. The type and value of this entry are implementation-dependent.

The pattern cell is described in its own coordinate system, defined when the tiling pattern is instantiated from its prototype with the **makepattern** operator. This *pattern coordinate system* is formed by concatenating the operator's *matrix* operand with the current transformation matrix at the time of instantiation. The pattern dictionary's **XStep**, **YStep**, and **BBox** values are interpreted in the pattern coordinate system, and the **PaintProc** procedure is executed within that coordinate system.

The placement of pattern cells in the tiling is based on the location of one *key pattern cell*, which is then displaced by multiples of **XStep** and **YStep** to replicate the pattern. The origin of the key pattern cell coincides with the origin of the pattern coordinate system; the phase of the tiling can be controlled by the translation

components of the **makepattern** operator's *matrix* operand. Because the pattern coordinate system is locked into user space at the time of instantiation, properties of the pattern that depend on the coordinate system, such as the size of the pattern cell and the phase of the tiling in device space, are frozen at that time and are unaffected by subsequent changes in the CTM or other graphics state parameters.

PaintProc Procedure

As described above in Section 4.9.1, "Using Patterns," the first step in painting with a pattern is to establish the pattern dictionary as the current color in the graphics state. In the case of a tiling pattern, subsequent painting operations will tile the painted areas with the pattern cell described in the dictionary. Whenever it needs to obtain the pattern cell, the interpreter does the following:

1. Invokes **gsave**
2. Installs the graphics state that was in effect at the time the tiling pattern was instantiated, with certain parameters altered as documented in the description of the **makepattern** operator in Chapter 8
3. Pushes the pattern dictionary on the operand stack
4. Executes the pattern's **PaintProc** procedure
5. Invokes **grestore**

The **PaintProc** procedure is expected to consume its dictionary operand and to use the information at hand to paint the pattern cell. It must obey certain guidelines to avoid disrupting the environment in which it is executed:

- It should not invoke any of the operators listed in Appendix G as unsuitable for use in encapsulated PostScript files.
- It should not invoke **showpage**, **copypage**, or any device setup operator.
- Except for removing its dictionary operand, it should leave the stacks unchanged.
- It should have no side effects beyond painting the pattern cell. It should not alter objects in virtual memory or anywhere else. Because of the effects of caching, the **PaintProc** procedure is called at unpredictable times and in unpredictable environments. It should depend only on information in the pattern dictionary and should produce the same effect every time it is called.

Colored Tiling Patterns

A *colored tiling pattern* is one whose color is self-contained. In the course of painting the pattern cell, the **PaintProc** procedure explicitly sets the color of each graphical element it paints. A single pattern cell can contain elements that are painted different colors; it can also contain sampled grayscale or color images.

A **Pattern** color space representing a colored tiling pattern requires no additional parameters and can be specified with just the color space family name **Pattern**. The color space operand to **setcolorspace** can be either the name **Pattern** or a one-element array containing the name **Pattern**. A second parameter, the *underlying color space*—required as a second element of the array for uncolored tiling patterns—may optionally be included, but is ignored when using colored tiling patterns.

A color value operand to **setcolor** in such a color space has a single component, a pattern dictionary whose **PaintType** value is 1. Example 4.20 shows how to establish a colored tiling pattern as the current color, where *pattern* is a pattern dictionary of paint type 1.

Example 4.20

```
[/Pattern] setcolorspace                % Alternatively, /Pattern setcolorspace
pattern setcolor
```

Subsequent executions of painting operators, such as **fill**, **stroke**, **show**, and **imagemask**, will use the designated pattern to tile the areas to be painted.

Note: *The **image** operator in its five-operand form and the **colorimage** operator use a predetermined color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**) for interpreting their color samples, regardless of the current color space. Setting a **Pattern** color space has no effect on these operators. The one-operand (dictionary) form of **image** is not allowed, since numeric color components are not meaningful in a **Pattern** color space. The **imagemask** operator is allowed, however, because the image samples do not represent colors, but rather designate places where the current color is to be painted.*

Example 4.21 defines a colored tiling pattern and then uses it to paint a rectangle and a character glyph; Figure 4.10 shows the results.

Example 4.21

```
<<    /PatternType 1                          % Tiling pattern
      /PaintType 1                            % Colored
      /TilingType 1
      /BBox [0 0 60 60]
      /XStep 60
      /YStep 60

      /star
         {   gsave                            % Private procedure used by PaintProc
               0 12 moveto
               4 {144 rotate 0 12 lineto} repeat
               closepath fill
             grestore
         } bind

      /PaintProc
         {   begin                            % Push pattern on dictionary stack
               0.3 setgray                    % Set color for dark gray stars
                  15 15 translate star
                  30 30 translate star
               0.7 setgray                    % Set color for light gray stars
                  –30 0 translate star
                  30 –30 translate star
             end
         } bind
>>                                            % End prototype pattern dictionary

matrix                                        % Identity matrix
makepattern                                   % Instantiate the pattern
/Star4 exch def

120 120 184 120 4 copy                        % Two copies of rectangle operands

/Pattern setcolorspace
Star4 setcolor rectfill                       % Fill rectangle with stars
0.0 setgray rectstroke                        % Stroke black outline

/Times-Roman 270 selectfont
160 100 translate
0.9 setgray 0 0 moveto (A) show               % Paint glyph with gray
Star4 setpattern 0 0 moveto (A) show          % Paint glyph with stars
```

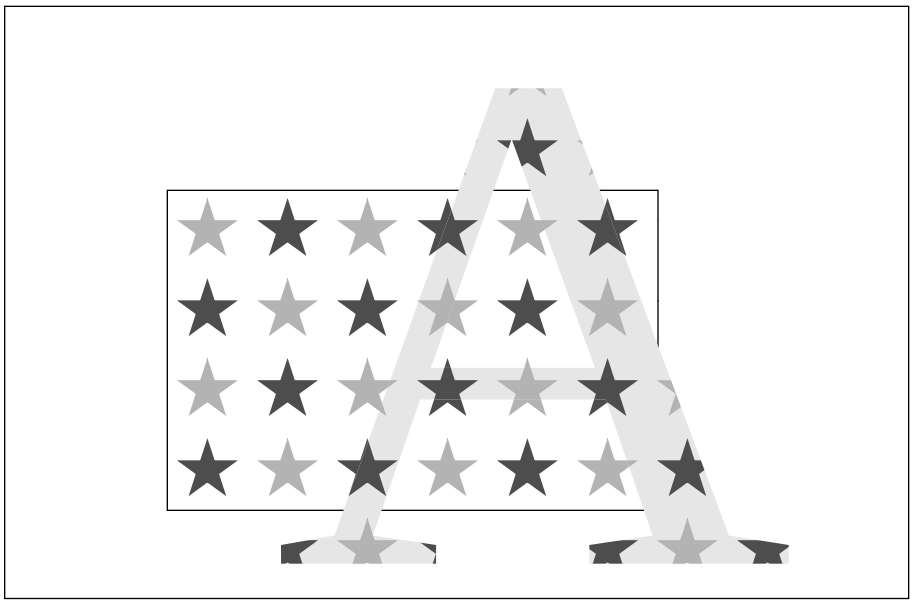


FIGURE 4.10 *Output from Example 4.21*

The pattern consists of four stars in two different colors. The **PaintProc** procedure specifies the colors of the stars. Several features of Example 4.21 are noteworthy:

- After constructing the prototype pattern dictionary, the program immediately invokes **makepattern** on it. The value assigned to Star4 is the *instantiated* pattern returned by **makepattern**. There is no need to save the prototype pattern unless it is to be instantiated in multiple ways, perhaps with different sizes or orientations.
- The program illustrates both methods of selecting a pattern for painting. The first time, it invokes the **setcolorspace** and **setcolor** operators separately. The second time, it uses the convenience operator **setpattern**. Note that the calls to **setgray** also change the color space to **DeviceGray**.
- The rectangle and the glyph representing the letter A are painted with the same instantiated pattern (the pattern dictionary returned by a single execution of **makepattern**). The pattern cells align, even though the current transformation matrix is altered between the two uses of the pattern.
- The pattern cell does not completely cover the tile: it leaves the spaces between the stars unpainted. When the tiling pattern is used as a color, the existing

background shows through these unpainted areas, as the appearance of the A glyph in Figure 4.10 demonstrates. The letter is first painted solid gray; when it is painted again with the star pattern, the gray continues to show between the stars.

Uncolored Tiling Patterns

An *uncolored tiling pattern* is one that has no inherent color: the color must be specified separately whenever the pattern is used. This provides a way to tile different regions of the page with pattern cells having the same shape but different colors. The pattern's **PaintProc** procedure does not explicitly specify any colors; it can use the **imagemask** operator, but not **image** or **colorimage**. (See Section 4.8.1, "Types of Color Space," for further discussion.)

A **Pattern** color space representing an uncolored tiling pattern requires a parameter: an array or name that identifies the *underlying color space* in which the actual color of the pattern is to be specified. Operands supplied to **setcolor** in such a color space must include both a color value in the underlying color space, specified by one or more numeric color components, and a pattern dictionary whose paint type is 2.

Note: *The underlying color space of a* ***Pattern*** *color space cannot itself be a* ***Pattern*** *color space.*

Example 4.22 establishes an uncolored tiling pattern as the current color, using **DeviceRGB** as the underlying color space. The component values *r*, *g*, and *b* specify a color in **DeviceRGB** space; *pattern* is a pattern dictionary with a paint type of 2.

Example 4.22

```
[ /Pattern
    [/DeviceRGB]
] setcolorspace
r g b pattern setcolor
```

Subsequent executions of painting operators, such as **fill**, **stroke**, **show**, and **imagemask**, will tile the areas to be painted with the pattern cell defined by *pattern*, using the color specified by the components *r*, *g*, and *b*.

Example 4.23 defines an uncolored tiling pattern and then uses it to paint a rectangle and a circle in different colors; Figure 4.11 shows the results.

Example 4.23

```
<<    /PatternType 1                          % Tiling pattern
      /PaintType 2                            % Uncolored
      /TilingType 1
      /BBox [–12 –12 12 12]
      /XStep 30
      /YStep 30

      /PaintProc
         {  pop                               % Pop pattern dictionary
            0 12 moveto
            4 {144 rotate 0 12 lineto} repeat
            closepath
            fill
         } bind
>>                                            % End prototype pattern dictionary

matrix                                        % Identity matrix
makepattern                                   % Instantiate the pattern
/Star exch def

140 110 170 100 4 copy                        % Two copies of rectangle operands

0.9 setgray rectfill                          % Fill rectangle with gray
[/Pattern /DeviceGray] setcolorspace
1.0 Star setcolor rectfill                    % Fill rectangle with white stars

225 185 60 0 360 arc                          % Build circular path
0.0 Star setpattern gsave fill grestore       % Fill circle with black stars
0.0 setgray stroke                            % Stroke black outline
```

The pattern consists of a single star, which the **PaintProc** procedure paints without first specifying a color. Most of the remarks after Example 4.21 on page 255 also apply to Example 4.23. Additionally:

- The program paints the rectangle twice, first with gray, then with the tiling pattern. To paint with the pattern, it supplies two operands to **setcolor**: the number 1.0, denoting white in the underlying **DeviceGray** color space, and the pattern dictionary.

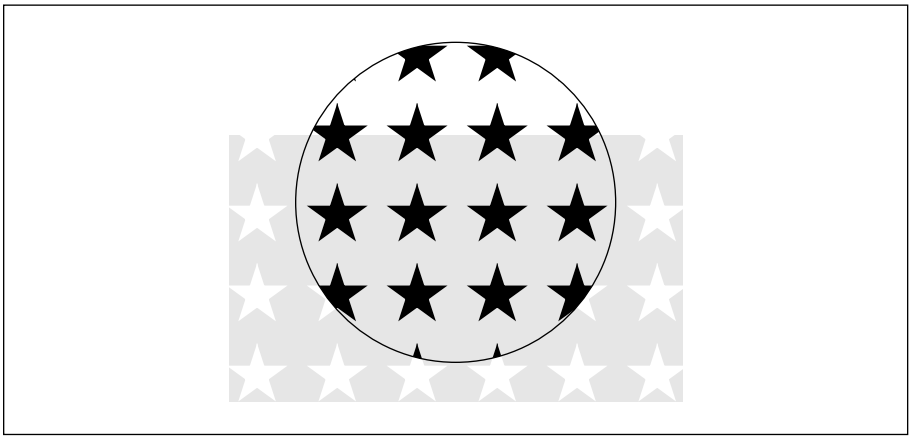


FIGURE 4.11 *Output from Example 4.23*

- The program paints the circle with the same pattern, but with the color set to 0.0 (black). Note that in this instance, **setpattern** inherits parameters from the existing color space (see the description of the **setpattern** operator in Chapter 8 for details).

4.9.3 Shading Patterns

Shading patterns *(LanguageLevel 3)* provide a smooth transition between colors across an area to be painted, independent of the resolution of any particular output device and without specifying the number of steps in the color transition. Patterns of this type are described by pattern dictionaries with a pattern type of 2. Table 4.10 shows the contents of this type of dictionary, most of whose entries are identical to corresponding entries in the type 1 pattern dictionary (Table 4.9). The most significant entry is **Shading**, whose value is a *shading dictionary* defining the properties of the shading pattern's *gradient fill.* This is a complex "paint" that determines the type of color transition the shading pattern produces when painted across an area.

TABLE 4.10 Entries in a type 2 pattern dictionary

KEY	TYPE	VALUE
PatternType	integer	*(Required)* A code identifying the pattern type that this dictionary describes; must be 2 for a shading pattern.
Shading	dictionary	*(Required)* A shading dictionary defining the shading pattern's gradient fill. The contents of the dictionary consist of the entries in Table 4.11 plus those in one of Tables 4.12 to 4.17. To ensure predictable behavior, once the pattern has been instantiated with the **makepattern** operator, this shading dictionary and all of its contents should be treated as if they were read-only.
XUID	array	*(Optional)* An *extended unique ID* that uniquely identifies the pattern (see Section 5.6.2, "Extended Unique ID Numbers). The presence of an **XUID** entry in a pattern dictionary enables the PostScript interpreter to save cached instances of the pattern for later use, even when the pattern dictionary is loaded into virtual memory multiple times (for instance, by different jobs). To ensure correct behavior, **XUID** values must be assigned from a central registry. This is particularly appropriate for patterns treated as named resources. Patterns that are created dynamically by an application program should *not* contain **XUID** entries.
Implementation	any	An additional entry inserted in the dictionary by the **makepattern** operator, containing information used by the interpreter to achieve proper shading of the pattern. The type and value of this entry are implementation-dependent.

By setting a shading pattern as the current color in the graphics state, a PostScript program can use it with painting operators such as **fill**, **stroke**, **show**, or **imagemask** to paint a path, glyph, or mask with a smooth color transition. When a shading is used in this way, the geometry of the gradient fill is independent of the geometry of the object being painted.

When the area to be painted is a relatively simple shape whose geometry is the same as that of the gradient fill itself, the **shfill** operator can be used instead. **shfill** accepts a shading dictionary as an operand and applies the corresponding gradient fill directly to current user space. This operator does not require the creation of a pattern dictionary and works without reference to the current path or current color in the graphics state. See the description of the **shfill** operator in Chapter 8 for details.

Note: *Patterns defined by type 2 pattern dictionaries do not tile. To create a tiling pattern containing a gradient fill, invoke the* ***shfill*** *operator from the* ***PaintProc*** *procedure of a type 1 (tiling) pattern.*

Shading Dictionaries

A shading dictionary specifies details of a particular gradient fill, including the type of shading to be used, the geometry of the area to be shaded, and the geometry of the gradient fill itself. Various shading types are available, depending on the value of the dictionary's **ShadingType** entry:

- *Function-based shadings* (type 1) define the color of every point in the domain using a mathematical function (not necessarily smooth or continuous).
- *Axial shadings* (type 2) define a color blend along a line between two points, optionally extended beyond the boundary points by continuing the boundary colors.
- *Radial shadings* (type 3) define a blend between two circles, optionally extended beyond the boundary circles by continuing the boundary colors. This type of shading is commonly used to represent three-dimensional spheres and cones.
- *Free-form Gouraud-shaded triangle meshes* (type 4) define a common construct used by many three-dimensional applications to represent complex colored and shaded shapes. Vertices are specified in free-form geometry.
- *Lattice-form Gouraud-shaded triangle meshes* (type 5) are based on the same geometrical construct as type 4, but with vertices specified as a pseudorectangular lattice.
- *Coons patch meshes* (type 6) construct a shading from one or more color patches, each bounded by four Bézier curves.
- *Tensor-product patch meshes* (type 7) are similar to type 6, but with 16 control points in each patch instead of 12.

Table 4.11 shows the entries that all shading dictionaries share in common; entries specific to particular shading types are described in the relevant sections below.

Note: *Many of the following descriptions refer to "the coordinate space into which the shading is painted." For shadings used with a type 2 pattern dictionary, this is the*

pattern coordinate system established at the time the pattern is instantiated with ***makepattern****. For shadings used directly with the* ***shfill*** *operator, it is the current user space.*

TABLE 4.11 Entries common to all shading dictionaries

KEY	TYPE	VALUE
ShadingType	integer	*(Required)* The shading type: 1 Function-based shading 2 Axial shading 3 Radial shading 4 Free-form Gouraud-shaded triangle mesh 5 Lattice-form Gouraud-shaded triangle mesh 6 Coons patch mesh 7 Tensor-product patch mesh
ColorSpace	name or array	*(Required)* The color space in which color values are expressed. May be any device, CIE-based, or special color space except a **Pattern** space. All color values in the shading are interpreted relative to this color space. See "Color Space: Special Considerations" on page 263 for further information.
Background	array	*(Optional)* An array of color components appropriate to the color space, specifying a single background color value. If present, this color is used before any painting operation involving the shading, to fill the entire area to be painted. The effect is as if the painting operation were performed twice: first with the background color and then again with the shading.
BBox	array	*(Optional)* An array of four numbers giving the left, bottom, right, and top coordinates, respectively, of the shading's bounding box. The coordinates are interpreted in the coordinate space into which the shading is painted. If present, this bounding box is applied as a temporary clipping boundary when the shading is painted, in addition to the current clipping path and any other clipping boundaries in effect at that time.
AntiAlias	boolean	*(Optional)* A flag indicating whether to filter the shading function to prevent aliasing artifacts. The shading operators sample shading functions at a rate determined by the resolution of the output device. Aliasing can occur if the function is not smooth—that is, if it has a high spatial frequency relative to the sampling rate. Anti-aliasing can be computationally expensive and is usually unnecessary, since most shading functions are smooth enough, or are sampled at a high enough frequency, to avoid aliasing effects. This feature may not be implemented on some devices, in which case this flag is ignored. Default value: *false*.

Some types of shading dictionary include a **DataSource** entry, whose value is an array, string, or file containing arbitrary amounts of descriptive data characterizing the shading's gradient fill. Since a shading pattern may access its shading dictionary multiple times, the descriptive data must be provided in reusable form. An array or string is reusable, but its length is subject to an implementation limit. A file can be of unlimited length, but only positionable files are reusable. In-line data obtained from **currentfile** is not reusable, and must be converted into reusable form by means of the **ReusableStreamDecode** filter. Nonreusable data sources may be used only with the **shfill** operator.

In addition, some shading dictionaries also include a function dictionary defining how colors vary across the area to be shaded. In such cases, the shading dictionary usually defines the geometry of the shading, while the function dictionary defines the color transitions across that geometry. The function dictionary is required for some types of shading and optional for others. Function dictionaries are described in detail in Section 3.10.1, "Function Dictionaries."

Note: *Discontinuous color transitions, or those with high spatial frequency, may exhibit aliasing effects when painted at low effective resolutions.*

Color Space: Special Considerations

Conceptually, a shading determines a color value for each individual point within the area to be painted. In practice, however, the shading may actually be used to compute color values only for some subset of the points in the target area, with the colors of the intervening points determined by interpolation between the ones computed. PostScript implementations are free to use this strategy as long as the interpolated color values approximate those defined by the shading to within the tolerance specified by the smoothness parameter in the graphics state (see the description of the **setsmoothness** operator in Chapter 8). The **ColorSpace** entry common to all shading dictionaries not only defines the color space in which the shading specifies its color values, but also determines the color space in which color interpolation is performed.

Note: *Some shading types (4, 5, 6, and 7) perform interpolation on a parametric value supplied as* input *to the shading's color mapping function, as described in the relevant sections below. This form of interpolation is conceptually distinct from the interpolation described here, which operates on the* output *color values produced by the color mapping function and takes place within the shading's target color space.*

Gradient fills between colors defined by most shadings are implemented using a variety of interpolation algorithms, and these algorithms are sensitive to the characteristics of the color space. Linear interpolation, for example, may have observably different results when applied in CMYK color space than in the CIE 1976 L*a*b* color space, even if the starting and ending colors are perceptually identical. The difference arises because the two color spaces are not linear relative to each other. Shadings are rendered according to the following rules:

- If **ColorSpace** is a device color space different from the native color space of the output device, color values in the shading will be converted to the native color space using the standard conversion formulas described in Section 7.2, "Conversions among Device Color Spaces." To optimize performance, these conversions may take place at any time (either before or after any interpolation on the color values in the shading). Thus, shadings defined with device color spaces may have color gradient fills that are less accurate and somewhat device-dependent. (This does not apply to axial and radial shadings—shading types 2 and 3—because those shading types perform gradient fill calculations on a single variable and then convert to parametric colors.)
- If **ColorSpace** is a CIE-based color space, all gradient fill calculations will be performed in that space. Conversion to device colors will occur only after all interpolation calculations are performed. Thus, the color gradients will be device-independent for the colors generated at each point.
- If **ColorSpace** is a **Separation** or **DeviceN** color space and the specified colorants are supported, no color conversion calculations are needed. If the specified colorants are not supported (so that the color space's *alternativeSpace* parameter must be used), gradient fill calculations will be performed in the designated **Separation** or **DeviceN** color space before conversion to the alternative space. Thus, nonlinear *tintTransform* functions will be accommodated for the best possible representation of the shading.
- If **ColorSpace** is an **Indexed** color space, all color values specified in the shading will be immediately converted to the base color space. Depending on whether the base color space is a device or CIE-based space, gradient fill calculations will be performed as stated above. Interpolation never occurs in an **Indexed** color space, which is quantized and inappropriate for calculations that assume a continuous range of colors. For similar reasons, an **Indexed** color space is not allowed in any shading whose color values are generated by a function; this applies to any shading dictionary that contains a **Function** entry.

Shading Types

In addition to the entries listed in Table 4.11, all shading dictionaries must have entries specific to the type of shading they represent, as indicated by the value of their **ShadingType** key. The following sections describe the available shading types and the dictionary entries specific to each.

Type 1 (Function-Based) Shadings

In type 1 (function-based) shadings, the color of every point in the domain is defined by a specified mathematical function. The function is not necessarily smooth or continuous. This is the most general of the available shading types, and is useful for shadings that cannot be adequately described with any of the other types. In addition to the entries in Table 4.11, a type 1 shading dictionary includes the entries listed in Table 4.12.

Note: *This type of shading may not be used with an* ***Indexed*** *color space.*

TABLE 4.12 Additional entries specific to a type 1 shading dictionary

KEY	TYPE	VALUE
Domain	array	*(Optional)* An array of four numbers specifying the rectangular domain of coordinates over which the color function(s) are defined. Default value: [0 1 0 1].
Matrix	array	*(Optional)* A transformation matrix mapping the coordinate space specified by the **Domain** entry into the coordinate space in which the shading is painted. For example, to map the domain rectangle [0 1 0 1] to a 1-inch square with lower-left corner at coordinates (100, 100) in default user space, the **Matrix** value would be [72 0 0 72 100 100]. Default value: the identity matrix [1 0 0 1 0 0].
Function	dictionary or array	*(Required)* A 2-in, *n*-out function dictionary or an array of *n* 2-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). Each function dictionary's domain must be a superset of that of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.

The domain rectangle (**Domain**) establishes an internal coordinate space for the shading that is independent of the coordinate space in which it is to be painted. The color function(s) (**Function**) specify the color of the shading at each point within this domain rectangle. The transformation matrix (**Matrix**) then maps the domain rectangle into a corresponding rectangle or parallelogram in the coordinate space in which the shading is painted. Points within the shading's bounding box (**BBox**) that fall outside this transformed domain rectangle will be painted with the shading's background color (**Background**); if the shading dictionary has no **Background** entry, such points will be left unpainted. If the function is undefined at any point within the declared domain rectangle, an **undefinedresult** error may occur, even if the corresponding transformed point falls outside the shading's bounding box.

Type 2 (Axial) Shadings

Type 2 (axial) shadings define a color blend that varies along a linear axis between two endpoints and extends indefinitely perpendicular to that axis. The shading may optionally be extended beyond either or both endpoints by continuing the boundary colors indefinitely. In addition to the entries in Table 4.11 on page 262, a type 2 shading dictionary includes the entries listed in Table 4.13.

Note: *This type of shading may not be used with an* ***Indexed*** *color space.*

TABLE 4.13 Additional entries specific to a type 2 shading dictionary

KEY	TYPE	VALUE
Coords	array	*(Required)* An array of four numbers [x_0 y_0 x_1 y_1] specifying the starting and ending coordinates of the axis, expressed in the coordinate space in which the shading is painted.
Domain	array	*(Optional)* An array of two numbers [t_0 t_1] specifying the limiting values of a parametric variable *t*. The variable is considered to vary linearly between these two values as the color gradient varies between the starting and ending points of the axis. The variable *t* becomes the argument with which the color function(s) are called. Default value: `[0 1]`.
Function	dictionary or array	*(Required)* A 1-in, *n*-out function dictionary or an array of *n* 1-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). The function(s) are called with values of the parametric variable *t* in the domain defined by the shading dictionary's **Domain** entry. Each function dictionary's domain must be a superset of that

		of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.
Extend	array	*(Optional)* An array of two boolean values specifying whether to extend the shading beyond the starting and ending points of the axis, respectively. Default value: [*false false*].

The color blend is accomplished by linearly mapping each point (x, y) along the axis between the endpoints (x_0, y_0) and (x_1, y_1) to a corresponding point in the domain specified by the shading dictionary's **Domain** entry. The point $(0, 0)$ in the domain corresponds to (x_0, y_0) on the axis, and $(1, 0)$ corresponds to (x_1, y_1). Since all points along a line in domain space perpendicular to the line from $(0, 0)$ to $(1, 0)$ will have the same color, only the new value of x needs to be computed:

$$x' = \frac{(x_1 - x_0)(x - x_0) + (y_1 - y_0)(y - y_0)}{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

The value of the parametric variable t is then determined from x' as follows:

- For $0 \le x' \le 1$, $t = t_0 + (t_1 - t_0)x'$.
- For $x' < 0$, if the first value in the **Extend** array is *true*, then $t = t_0$; otherwise, t is undefined and the point is left unpainted.
- For $x' > 1$, if the second value in the **Extend** array is *true*, then $t = t_1$; otherwise, t is undefined and the point is left unpainted.

The value of t is then passed as the input argument to the function(s) defined by the shading dictionary's **Function** entry, yielding the component values of the color with which to paint the point (x, y).

Type 3 (Radial) Shadings

Type 3 (radial) shadings define a color blend that varies between two circles. They are commonly used to depict three-dimensional spheres and cones. In addition to the entries in Table 4.11 on page 262, a type 3 shading dictionary includes the entries listed in Table 4.14.

Note: *This type of shading may not be used with an* ***Indexed*** *color space.*

TABLE 4.14 Additional entries specific to a type 3 shading dictionary

KEY	TYPE	VALUE
Coords	array	*(Required)* An array of six numbers [x_0 y_0 r_0 x_1 y_1 r_1] specifying the centers and radii of the starting and ending circles, expressed in the coordinate space in which the shading is painted. The radii r_0 and r_1 must both be greater than or equal to 0. If one radius is 0, the corresponding circle is treated as a point; if both are 0, nothing is painted.
Domain	array	*(Optional)* An array of two numbers [t_0 t_1] specifying the limiting values of a parametric variable *t*. The variable is considered to vary linearly between these two values as the color gradient varies between the starting and ending circles. The variable *t* becomes the argument with which the color function(s) are called. Default value: [0 1].
Function	dictionary or array	*(Required)* A 1-in, *n*-out function dictionary or an array of *n* 1-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). The function(s) are called with values of the parametric variable *t* in the domain defined by the shading dictionary's **Domain** entry. Each function dictionary's domain must be a superset of that of the shading dictionary. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.
Extend	array	*(Optional)* An array of two boolean values specifying whether to extend the shading beyond the starting and ending circles, respectively. Default value: [*false false*].

The color blend is based on a family of *blend circles* interpolated between the starting and ending circles that are defined by the shading dictionary's **Coords** entry. The blend circles are defined in terms of a subsidiary parametric variable

$$s = \frac{t - t_0}{t_1 - t_0}$$

which varies linearly between 0.0 and 1.0 as t varies across the domain from t_0 to t_1, as specified by the dictionary's **Domain** entry. The center and radius of each blend circle are given by the parametric equations

$$\begin{aligned} x_c(s) &= x_0 + s \times (x_1 - x_0) \\ y_c(s) &= y_0 + s \times (y_1 - y_0) \\ r(s) &= r_0 + s \times (r_1 - r_0) \end{aligned}$$

Each value of s between 0.0 and 1.0 determines a corresponding value of t, which is then passed as the input argument to the function(s) defined by the shading dictionary's **Function** entry. This yields the component values of the color with which to fill the corresponding blend circle. For values of s not lying between 0.0 and 1.0, the boolean values in the shading dictionary's **Extend** entry determine whether and how the shading will be extended. If the first of the two boolean values is *true*, the shading is extended beyond the defined starting circle to values of s less than 0.0; if the second boolean value is *true*, the shading is extended beyond the defined ending circle to s values greater than 1.0.

Note that either of the starting or ending circles may be larger than the other. If the shading is extended at the smaller end, the family of blend circles continues as far as that value of s for which the radius of the blend circle $r(s) = 0$; if the shading is extended at the larger end, the blend circles continue as far as that s value for which $r(s)$ is large enough to encompass the shading's entire bounding box (**BBox**). Extending the shading can thus cause painting to extend beyond the areas defined by the two circles themselves.

Conceptually, all of the blend circles are painted in order of increasing values of s, from smallest to largest. Blend circles extending beyond the starting circle are painted in the same color defined by the shading dictionary's **Function** entry for the starting circle ($s = 0.0$, $t = t_0$); those extending beyond the ending circle are painted in the color defined for the ending circle ($s = 1.0$, $t = t_1$). The painting is opaque, with the color of each circle completely overlaying those preceding it; thus if a point lies within more than one blend circle, its final color will be that of the last of the enclosing circles to be painted, corresponding to the greatest value of s. Note the following points:

- If one of the starting and ending circles entirely contains the other, the shading will depict a sphere.

- If neither circle contains the other, the shading will depict a cone. If the starting circle is larger, the cone will appear to point out of the page; if the ending circle is larger, the cone will appear to point into the page.

Type 4 Shadings (Free-Form Gouraud-Shaded Triangle Meshes)

Type 4 shadings (free-form Gouraud-shaded triangle meshes) are commonly used to represent complex colored and shaded three-dimensional shapes. The area to be shaded is defined by a path composed entirely of triangles. The color at each vertex of the triangles is specified, and a technique known as Gouraud interpolation is used to color the interiors. The interpolation functions defining the shading may be linear or nonlinear. In addition to the entries in Table 4.11 on page 262, a type 4 shading dictionary includes the entries listed in Table 4.15.

TABLE 4.15 Additional entries specific to a type 4 shading dictionary

KEY	TYPE	VALUE
DataSource	array, string, or file	*(Required)* The sequence of vertex coordinates and colors defining the free-form triangle mesh.
BitsPerCoordinate	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each vertex coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16.
BitsPerFlag	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent the edge flag for each vertex (see below). Allowed values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Allowed values for the edge flag itself are 0, 1, and 2.
Decode	array	*(Required, unless **DataSource** is an array)* An array of numbers describing how to map vertex coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows: $[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max} \cdots c_{n,\min}\ c_{n,\max}]$ Note that only one pair of c values should be specified if a **Function** entry is present.

Function	dictionary or array	*(Optional)* A 1-in, *n*-out function dictionary or an array of *n* 1-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by *n* separate color components; the designated function(s) will be called with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value. This entry may not be used with an **Indexed** color space.

The shading dictionary's **DataSource** entry provides a sequence of vertex coordinates and color data that defines the triangle mesh. Each vertex is specified by the following values, in the order shown:

$$f\ x\ y\ c_1 \ldots c_n$$

where

f is the vertex's edge flag (discussed below)
x and *y* are its horizontal and vertical coordinates
$c_1 \ldots c_n$ are its color components

All vertex coordinates are expressed in the coordinate space in which the shading is painted. If the shading dictionary includes a **Function** entry, then only a single parametric value, *t*, is permitted for each vertex in place of the color components $c_1 \ldots c_n$.

The *edge flag* associated with each vertex determines the way it connects to the other vertices of the triangle mesh. A vertex v_a with an edge flag value $f_a = 0$ begins a new triangle, unconnected to any other. At least two more vertices (v_b and v_c) must be provided, but their edge flags will be ignored. These three vertices define a triangle (v_a, v_b, v_c), as shown in Figure 4.12.

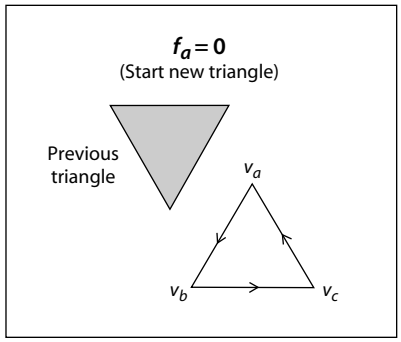


FIGURE 4.12 *Starting a new triangle in a free-form Gouraud-shaded triangle mesh*

Subsequent triangles are defined by a single new vertex combined with two vertices of the preceding triangle. Given triangle (v_a, v_b, v_c), where vertex v_a precedes vertex v_b in the data source and v_b precedes v_c, a new vertex v_d can form a new triangle on side v_{bc} or side v_{ac}, as shown in Figure 4.13. (Side v_{ab} is assumed to be shared with a preceding triangle and so is not available for continuing the mesh.) If the edge flag is $f_d = 1$ (side v_{bc}), the next vertex forms the triangle (v_b, v_c, v_d); if the edge flag is $f_d = 2$ (side v_{ac}), the next vertex forms the triangle (v_a, v_c, v_d). An edge flag of $f_d = 0$ would start a new triangle, as described above.

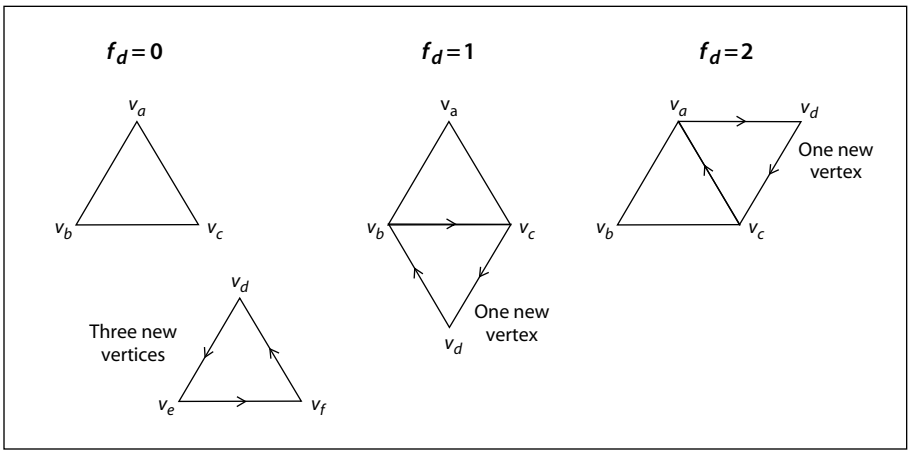


FIGURE 4.13 *Connecting triangles in a free-form Gouraud-shaded triangle mesh*

Complex shapes can be created by using the edge flags to control the edge on which subsequent triangles are formed. Figure 4.14 shows two simple examples. Mesh 1 begins with triangle 1 and uses the following edge flags to draw each succeeding triangle:

1 ($f_a = f_b = f_c = 0$)
2 ($f_d = 1$)
3 ($f_e = 1$)
4 ($f_f = 1$)
5 ($f_g = 1$)
6 ($f_h = 1$)
7 ($f_i = 2$)
8 ($f_j = 2$)
9 ($f_k = 2$)
10 ($f_l = 1$)
11 ($f_m = 1$)

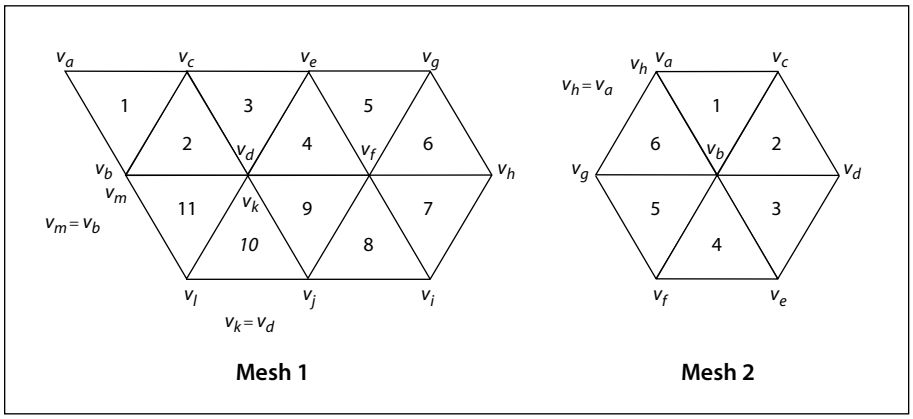


FIGURE 4.14 *Varying the value of the edge flag to create different shapes*

Mesh 2 again begins with triangle 1 and uses the edge flags

1 ($f_a = f_b = f_c = 0$)
2 ($f_d = 1$)
3 ($f_e = 2$)
4 ($f_f = 2$)
5 ($f_g = 2$)
6 ($f_h = 2$)

The data source must provide vertex data for a whole number of triangles with appropriate edge flags; otherwise, a **rangecheck** error will occur. If the mesh contains only a few vertices, they may be represented by an array of numeric values (integers for edge flags, integers or real numbers for coordinates and colors); in

this case, only the **ShadingType**, **DataSource**, and **Function** entries in the shading dictionary are relevant. If the mesh contains many vertices, the data should be encoded compactly and drawn from a string or a file. The encoding is specified by the dictionary's **BitsPerFlag**, **BitsPerCoordinate**, **BitsPerComponent**, and **Decode** entries.

The data for each vertex consists of the following items, reading in sequence from higher-order to lower-order bit positions:

- An edge flag, expressed in **BitsPerFlag** bits
- A pair of horizontal and vertical coordinates, each expressed in **BitsPerCoordinate** bits
- A set of *n* color components (where *n* is the number of components in the shading's color space), each expressed in **BitsPerComponent** bits, in the order expected by the **setcolor** operator

Each set of vertex data must occupy a whole number of bytes; if the total number of bits required is not divisible by 8, the last data byte for each vertex is padded at the end with extra bits, which are ignored. The coordinates and color values are decoded according to the **Decode** array in the same way as in an image dictionary; see "Sample Decoding" on page 299 for details.

If the shading dictionary contains a **Function** entry, the color data for each vertex must be specified by a single parametric value *t*, rather than by *n* separate color components. All linear interpolation within the triangle mesh is done using the *t* values; after interpolation, the results are passed to the function(s) specified in the **Function** entry to determine the color of each point.

Type 5 Shadings (Lattice-Form Gouraud-Shaded Triangle Meshes)

Type 5 shadings (lattice-form Gouraud-shaded triangle meshes) are similar to type 4, but instead of using free-form geometry, their vertices are arranged in a *pseudorectangular lattice*, which is topologically equivalent to a rectangular grid. The vertices are organized into rows, which need not be geometrically linear (see Figure 4.15). In addition to the entries in Table 4.11 on page 262, a type 5 shading dictionary includes the entries listed in Table 4.16.

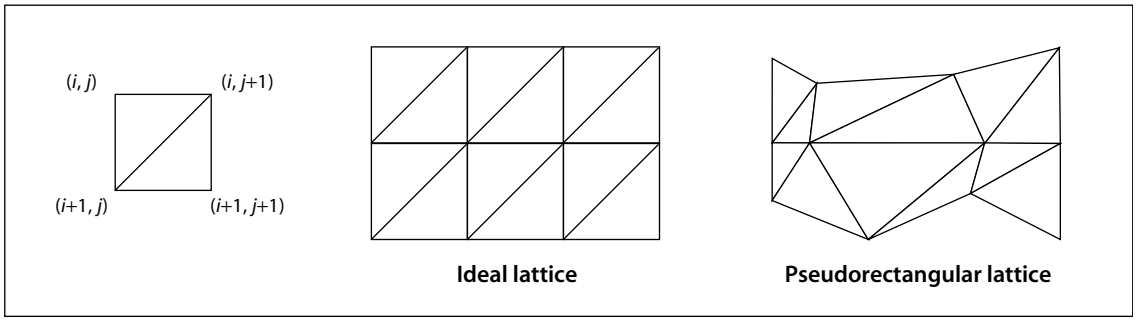


FIGURE 4.15 *Lattice-form triangular meshes*

TABLE 4.16 Additional entries specific to a type 5 shading dictionary

KEY	TYPE	VALUE
DataSource	array, string, or file	*(Required)* The sequence of vertex coordinates and colors defining the lattice-form triangle mesh.
BitsPerCoordinate	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each vertex coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16.
VerticesPerRow	integer	*(Required)* The number of vertices in each row of the lattice; must be greater than or equal to 2. The number of rows need not be specified.
Decode	array	*(Required, unless **DataSource** is an array)* An array of numbers describing how to map vertex coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows: $[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max} \cdots\ c_{n,\min}\ c_{n,\max}]$ Note that only one pair of *c* values should be specified if a **Function** entry is present.
Function	dictionary or array	*(Optional)* A 1-in, *n*-out function dictionary or an array of *n* 1-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by *n* separate color components; the designated function(s) will be called

with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value.

This entry may not be used with an **Indexed** color space.

The data source for a type 5 shading has the same format as for type 4, except that it does not use edge flags to define the geometry of the triangle mesh. The data for each vertex thus consists of the following values, in the order shown:

$x\ y\ c_1 \ldots c_n$

where

x and y are the vertex's horizontal and vertical coordinates
$c_1 \ldots c_n$ are its color components

All vertex coordinates are expressed in the coordinate space in which the shading is painted. If the shading dictionary includes a **Function** entry, then only a single parametric value, t, is permitted for each vertex in place of the color components $c_1 \ldots c_n$.

The **VerticesPerRow** entry in the shading dictionary gives the number of vertices in each row of the lattice. All of the vertices in a row are specified sequentially, followed by those for the next row. Given m rows of k vertices each, the triangles of the mesh are constructed using the following triplets of vertices, as shown in Figure 4.15:

$(V_{i,j}, V_{i,j+1}, V_{i+1,j})$ for $0 \le i \le m-2,\ 0 \le j \le k-2$
$(V_{i,j+1}, V_{i+1,j}, V_{i+1,j+1})$

See "Type 4 Shadings (Free-Form Gouraud-Shaded Triangle Meshes)" on page 270 for further details on the format of the vertex data.

Type 6 Shadings (Coons Patch Meshes)

Type 6 shadings (Coons patch meshes) are constructed from one or more color patches, each bounded by four Bézier curves. Degenerate Bézier curves are allowed and are useful for certain graphical effects. At least one complete patch must be specified.

A Coons patch generally has two independent aspects:

- Colors are specified for each corner of the unit square, and bilinear interpolation is used to fill in colors over the entire unit square.
- Coordinates are mapped from the unit square into a four-sided patch whose sides are not necessarily linear. The mapping is continuous: the corners of the unit square map to corners of the patch, and the sides of the unit square map to sides of the patch, as shown in Figure 4.16.

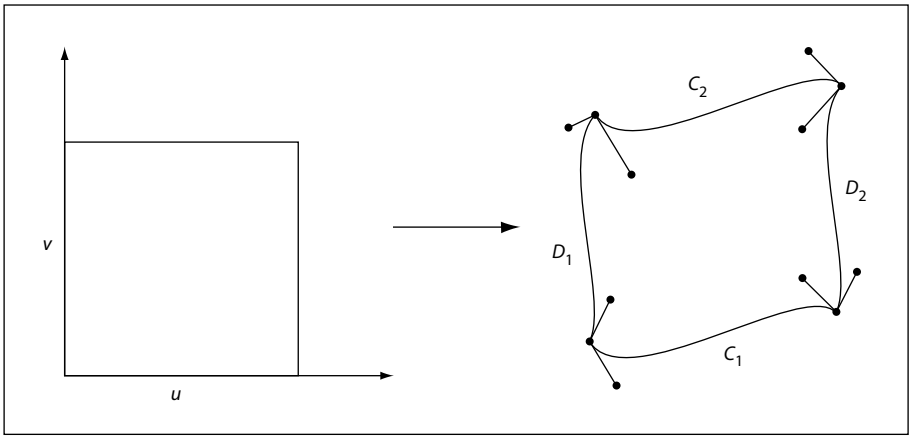


FIGURE 4.16 *Coordinate mapping from a unit square to a four-sided Coons patch*

The sides of the patch are given by four cubic Bézier curves, C_1, C_2, D_1, and D_2, defined over a pair of parametric variables u and v that vary horizontally and ver-

tically across the unit square. The four corners of the Coons patch satisfy the equations

$$C_1(0) = D_1(0)$$
$$C_1(1) = D_2(0)$$
$$C_2(0) = D_1(1)$$
$$C_2(1) = D_2(1)$$

Two surfaces can be described that are linear interpolations between the boundary curves. Along the u axis, the surface S_C is defined by

$$S_C(u, v) = (1 - v) \times C_1(u) + v \times C_2(u)$$

Along the v axis, the surface S_D is given by

$$S_D(u, v) = (1 - u) \times D_1(v) + u \times D_2(v)$$

A third surface is the bilinear interpolation of the four corners:

$$\begin{aligned} S_B(u, v) = {} & (1 - v) \times [(1 - u) \times C_1(0) + u \times C_1(1)] \\ & + v \times [(1 - u) \times C_2(0) + u \times C_2(1)] \end{aligned}$$

The coordinate mapping for the shading is given by the surface S, defined as

$$S = S_C + S_D - S_B$$

This defines the geometry of each patch. A patch mesh is constructed from a sequence of one or more such colored patches.

Patches can sometimes appear to fold over on themselves—for example, if a boundary curve intersects itself. As the value of parameter u or v increases in parameter space, the location of the corresponding pixels in device space may change direction, so that new pixels are mapped onto previous pixels already mapped. If more than one point (u, v) in parameter space is mapped to the same point in device space, the point selected will be the one with the largest value of v; if multiple points have the same v, the one with the largest value of u will be selected. If one patch overlaps another, the patch that appears later in the data source paints over the earlier one.

Note also that the patch is a control surface, rather than a painting geometry. The outline of a projected square (that is, the painted area) may not be the same as the

patch boundary if, for example, the patch folds over on itself, as shown in Figure 4.17.

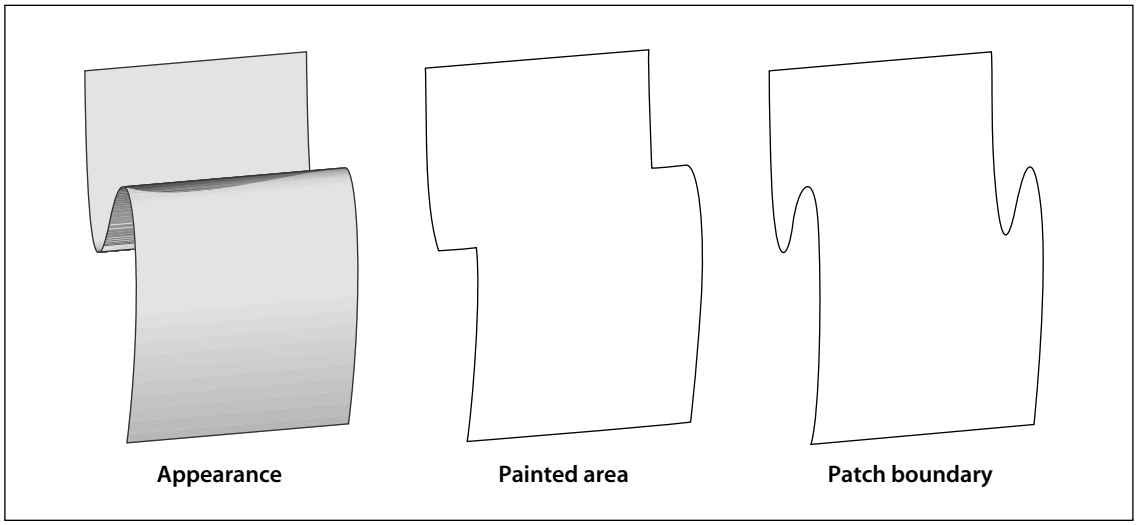


FIGURE 4.17 *Painted area and boundary of a Coons patch*

In addition to the entries in Table 4.11 on page 262, a type 6 shading dictionary includes the entries listed in Table 4.17.

TABLE 4.17 Additional entries specific to a type 6 shading dictionary

KEY	TYPE	VALUE
DataSource	array, string, or file	*(Required)* The sequence of coordinates and colors defining the patch mesh.
BitsPerCoordinate	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each geometric coordinate. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent each color component. Allowed values are 1, 2, 4, 8, 12, and 16.
BitsPerFlag	integer	*(Required, unless **DataSource** is an array)* The number of bits used to represent the edge flag for each patch (see below). Allowed values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Allowed values for the edge flag itself are 0, 1, 2, and 3.

Decode	array	*(Required, unless **DataSource** is an array)* An array of numbers describing how to map coordinates and color components into the appropriate ranges of values. The decoding method is similar to that used in image dictionaries (see "Sample Decoding" on page 299). The ranges are specified as follows: $[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max} \cdots\ c_{n,\min}\ c_{n,\max}]$ Note that only one pair of *c* values should be specified if a **Function** entry is present.
Function	dictionary or array	*(Optional)* A 1-in, *n*-out function dictionary or an array of *n* 1-in, 1-out function dictionaries (where *n* is the number of color components in the shading dictionary's color space). If this entry is present, the color data for each vertex must be specified by a single parametric variable rather than by *n* separate color components; the designated function(s) will be called with each interpolated value of the parametric variable to determine the actual color at each point. If **DataSource** is a string or a file, each input value will be clipped to the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. If **DataSource** is an array (in which case **Decode** is not relevant), each input value will be clipped to the interval [0.0 1.0]. In either case, each function dictionary's domain must be a superset of the stated interval. If the values returned by the function(s) for a given color component are out of range, they will be adjusted to the nearest valid value. This entry may not be used with an **Indexed** color space.

The dictionary's data source provides a sequence of Bézier control points and color values that define the shape and colors of each patch. All of a patch's control points are given first, followed by the color values for its corners. Note that this differs from a triangle mesh (shading types 4 and 5), in which the coordinates and color of each vertex are given together. All control point coordinates are expressed in the coordinate space in which the shading is painted.

As in triangle meshes, the data source can be either an array of numeric values or a string or stream containing encoded values, whose representation is specified by the **BitsPerFlag**, **BitsPerCoordinate**, **BitsPerComponent**, and **Decode** entries. In the latter case, if the total number of data bits required to define the patch is not divisible by 8, the last byte is padded at the end with extra bits, which are ignored.

As in free-form triangle meshes (type 4), each patch has an *edge flag* that tells which edge, if any, it shares with the previous patch. An edge flag of 0 begins a new patch, unconnected to any other. This must be followed by 12 pairs of coordinates, $x_1\ y_1\ x_2\ y_2\ \ldots\ x_{12}\ y_{12}$, which specify the Bézier control points that define the four boundary curves. Figure 4.18 shows how these control points correspond to the Bézier curves C_1, C_2, D_1, and D_2 identified in Figure 4.16 on page 277. Color values are then given for the four corners of the patch, in the same order as the control points corresponding to the corners. Thus, c_1 is the color at coordinates (x_1, y_1), c_2 at (x_4, y_4), c_3 at (x_7, y_7), and c_4 at (x_{10}, y_{10}), as shown in the figure.

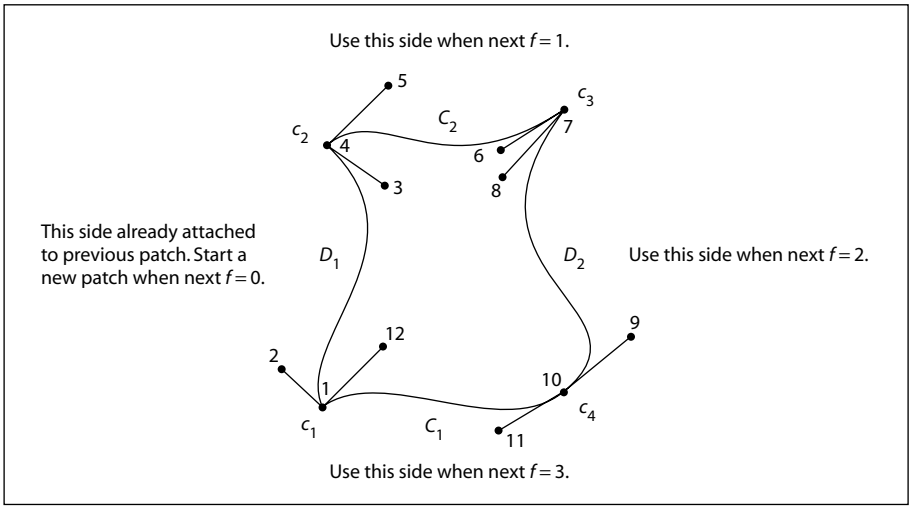


FIGURE 4.18 *Color values and edge flags in Coons patch meshes*

Figure 4.18 also shows how nonzero values of the edge flag ($f = 1, 2,$ or 3) connect a new patch to one of the edges of the previous patch. In this case, some of the previous patch's control points serve implicitly as control points for the new patch as well (see Figure 4.19), and so are not explicitly repeated in the data source. Table 4.18 summarizes the required data values for various values of the edge flag.

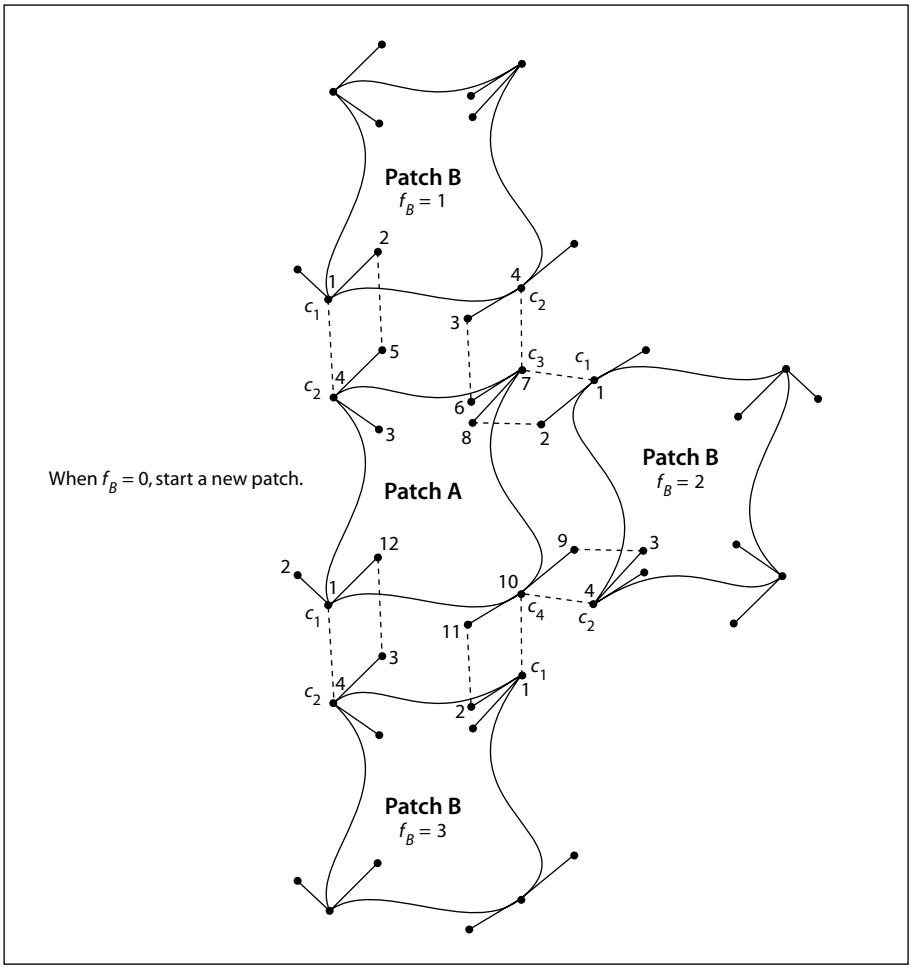


FIGURE 4.19 *Edge connections in a Coons patch mesh*

TABLE 4.18 Data values in a Coons patch mesh

EDGE FLAG	NEXT SET OF DATA VALUES
$f = 0$	$x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ x_4\ y_4\ x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$ $c_1\ c_2\ c_3\ c_4$ New patch; no implicit values

$f = 1$	$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$ $c_3\ c_4$ Implicit values: $(x_1, y_1) = (x_4, y_4)$ previous $(x_2, y_2) = (x_5, y_5)$ previous $(x_3, y_3) = (x_6, y_6)$ previous $(x_4, y_4) = (x_7, y_7)$ previous $c_1 = c_2$ previous $c_2 = c_3$ previous
$f = 2$	$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$ $c_3\ c_4$ Implicit values: $(x_1, y_1) = (x_7, y_7)$ previous $(x_2, y_2) = (x_8, y_8)$ previous $(x_3, y_3) = (x_9, y_9)$ previous $(x_4, y_4) = (x_{10}, y_{10})$ previous $c_1 = c_3$ previous $c_2 = c_4$ previous
$f = 3$	$x_5\ y_5\ x_6\ y_6\ x_7\ y_7\ x_8\ y_8\ x_9\ y_9\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}$ $c_3\ c_4$ Implicit values: $(x_1, y_1) = (x_{10}, y_{10})$ previous $(x_2, y_2) = (x_{11}, y_{11})$ previous $(x_3, y_3) = (x_{12}, y_{12})$ previous $(x_4, y_4) = (x_1, y_1)$ previous $c_1 = c_4$ previous $c_2 = c_1$ previous

If the shading dictionary contains a **Function** entry, the color data for each corner of a patch must be specified by a single parametric value t, rather than by n separate color components $c_1 \ldots c_n$. All linear interpolation within the mesh is done using the t values; after interpolation, the results are passed to the function(s) specified in the **Function** entry to determine the color of each point.

Type 7 Shadings (Tensor-Product Patch Meshes)

Type 7 shadings (tensor-product patch meshes) are identical to type 6, except that they are based on a bicubic tensor-product patch defined by 16 control points, instead of the 12 control points that define a Coons patch. The shading dictionaries representing the two patch types differ only in the value of the **ShadingType** entry and in the number of control points specified for each patch

in the data source. Although the Coons patch is more concise and easier to use, the tensor-product patch affords greater control over color mapping.

Like the Coons patch mapping, the tensor-product patch mapping is controlled by the location and shape of four cubic Bézier curves marking the boundaries of the patch. However, the tensor-product patch has four additional, "internal" control points to adjust the mapping. The 16 control points can be arranged in a 4-by-4 array indexed by row and column, as follows (see Figure 4.20):

$$\begin{array}{cccc} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{array}$$

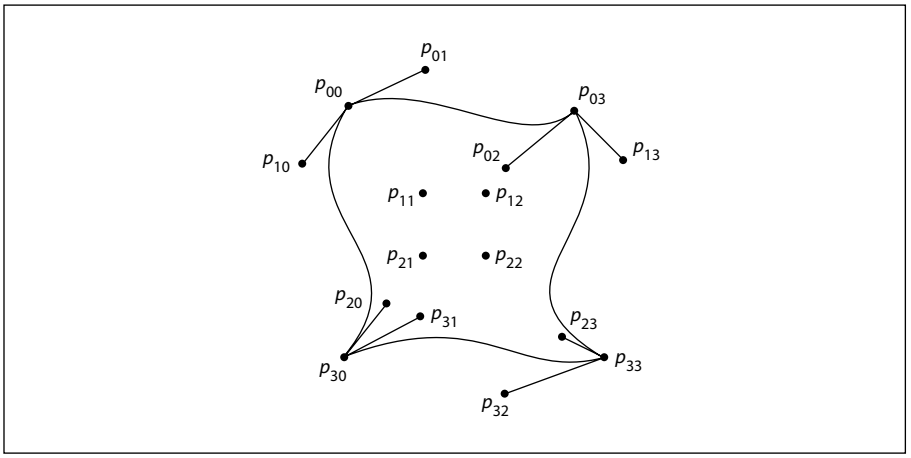


FIGURE 4.20 *Control points in a tensor-product mesh*

As in a Coons patch mesh, the geometry of the tensor-product patch is described by a surface defined over a pair of parametric variables, u and v, which vary horizontally and vertically across the unit square. The surface is defined by the equation

$$S(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} p_{ij} \times B_i(u) \times B_j(v)$$

where p_{ij} is the control point in row i and column j of the tensor, and B_i and B_j are the *Bernstein polynomials*

$$B_0(t) = (1-t)^3$$
$$B_1(t) = 3t \times (1-t)^2$$
$$B_2(t) = 3t^2 \times (1-t)$$
$$B_3(t) = t^3$$

Since each point p_{ij} is actually a pair of coordinates (x_{ij}, y_{ij}), the surface can also be expressed as

$$x(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} x_{ij} \times B_i(u) \times B_j(v)$$

$$y(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} y_{ij} \times B_i(u) \times B_j(v)$$

The geometry of the tensor-product patch can be visualized in terms of a cubic Bézier curve moving from the top boundary of the patch to the bottom. At the top and bottom, the control points of this curve coincide with those of the patch's top ($p_{00} \ldots p_{03}$) and bottom ($p_{30} \ldots p_{33}$) boundary curves, respectively. As the curve moves from the top edge of the patch to the bottom, each of its four control points follows a trajectory that is in turn a cubic Bézier curve defined by the four control points in the corresponding column of the array. That is, the starting point of the moving curve follows the trajectory defined by control points $p_{00} \ldots p_{30}$, the trajectory of the ending point is defined by points $p_{03} \ldots p_{33}$, and those of the two intermediate control points by $p_{01} \ldots p_{31}$ and $p_{02} \ldots p_{32}$. Equivalently, the patch can be considered to be traced by a Bézier curve moving from the left edge to the right, with its control points following the trajectories defined by the rows of the coordinate array instead of the columns.

The Coons patch (type 6) is actually a special case of the tensor-product patch (type 7) in which the four internal control points ($p_{11}, p_{12}, p_{21}, p_{22}$) are implicit-

ly defined by the boundary curves. The values of the internal control points are given by the equations

$$p_{11} = S(1/3, 2/3)$$
$$p_{12} = S(2/3, 2/3)$$
$$p_{21} = S(1/3, 1/3)$$
$$p_{22} = S(2/3, 1/3)$$

where S is the Coons surface equation

$$S = S_C + S_D - S_B$$

discussed above under "Type 6 Shadings (Coons Patch Meshes)" on page 277. In the more general tensor-product patch, the values of these four points are unrestricted.

The coordinates of the control points in a tensor-product patch are actually stored in the shading dictionary's data source in the following order:

1	12	11	10
2	13	16	9
3	14	15	8
4	5	6	7

All control point coordinates are expressed in the coordinate space in which the shading is painted. These are followed by the color values for the four corners of the patch, in the same order as the corners themselves. If the patch's edge flag $f = 0$, all 16 control points and four corner colors must be explicitly specified in the data source; if $f = 1$, 2, or 3, the control points and colors for the patch's shared edge are implicitly understood to be the same as those along the specified edge of the previous patch, and are not repeated in the data source. Table 4.19 summarizes the data values for various values of the edge flag f, expressed in terms of the row and column indices used in Figure 4.20.

TABLE 4.19 Data values in a tensor-product patch mesh

EDGE FLAG	NEXT SET OF DATA VALUES
$f = 0$	$x_{00}\ y_{00}\ x_{10}\ y_{10}\ x_{20}\ y_{20}\ x_{30}\ y_{30}\ x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}$ $x_{13}\ y_{13}\ x_{03}\ y_{03}\ x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$ $c_{00}\ c_{30}\ c_{33}\ c_{03}$ New patch; no implicit values
$f = 1$	$x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$ $x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$ $c_{33}\ c_{03}$ Implicit values: $(x_{00}, y_{00}) = (x_{30}, y_{30})$ previous; $c_{00} = c_{30}$ previous $(x_{10}, y_{10}) = (x_{31}, y_{31})$ previous; $c_{30} = c_{33}$ previous $(x_{20}, y_{20}) = (x_{32}, y_{32})$ previous $(x_{30}, y_{30}) = (x_{33}, y_{33})$ previous
$f = 2$	$x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$ $x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$ $c_{33}\ c_{03}$ Implicit values: $(x_{00}, y_{00}) = (x_{33}, y_{33})$ previous; $c_{00} = c_{33}$ previous $(x_{10}, y_{10}) = (x_{23}, y_{23})$ previous; $c_{30} = c_{03}$ previous $(x_{20}, y_{20}) = (x_{13}, y_{13})$ previous $(x_{30}, y_{30}) = (x_{03}, y_{03})$ previous
$f = 3$	$x_{31}\ y_{31}\ x_{32}\ y_{32}\ x_{33}\ y_{33}\ x_{23}\ y_{23}\ x_{13}\ y_{13}\ x_{03}\ y_{03}$ $x_{02}\ y_{02}\ x_{01}\ y_{01}\ x_{11}\ y_{11}\ x_{21}\ y_{21}\ x_{22}\ y_{22}\ x_{12}\ y_{12}$ $c_{33}\ c_{03}$ Implicit values: $(x_{00}, y_{00}) = (x_{03}, y_{03})$ previous; $c_{00} = c_{03}$ previous $(x_{10}, y_{10}) = (x_{02}, y_{02})$ previous; $c_{30} = c_{00}$ previous $(x_{20}, y_{20}) = (x_{01}, y_{01})$ previous $(x_{30}, y_{30}) = (x_{00}, y_{00})$ previous

4.10 Images

The PostScript language's painting operators include general facilities for dealing with sampled images. A *sampled image* (or just "image" for short) is a rectangular array of *sample values*, each representing a color. The image may approximate the appearance of some natural scene obtained through an input scanner or a video camera, or it may be generated synthetically.

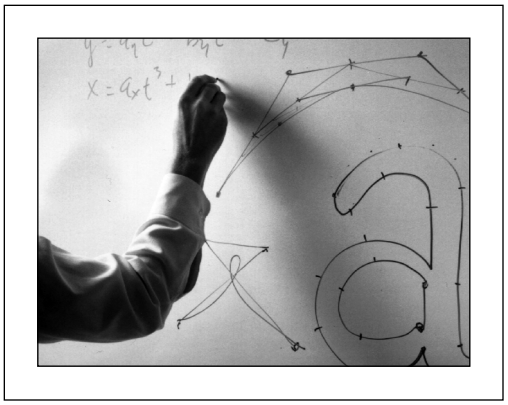


FIGURE 4.21 *Typical sampled image*

An image is defined by a sequence of samples obtained by scanning the image array in row or column order. Each sample in the array consists of as many color components as are needed for the color space in which they are specified—for example, one component for **DeviceGray**, three for **DeviceRGB**, four for **DeviceCMYK**, or whatever number is required by a particular **DeviceN** space. Each component consists of a 1-, 2-, 4-, 8-, or 12-bit integer, permitting the representation of 2, 4, 16, 256, or 4096 different values for each component.

Depending on LanguageLevel, PostScript implementations differ in the facilities they offer for images:

- Most LanguageLevel 1 implementations support only grayscale images—that is, ones whose image samples consist of a single gray component. These can be painted by means of the five-operand form of the **image** operator. Image samples must consist of 1, 2, 4, or 8 bits per component; 12-bit components are not supported. The image's source data must be provided by a procedure and not directly by a file or string.

- A few LanguageLevel 1 implementations have been extended to support color images containing three or four components per sample, interpreted as RGB or CMYK. These can be painted by means of the **colorimage** operator. LanguageLevel 1 products containing this feature are primarily color printers, and also support the **setcmykcolor** operator and 4-color rendering features.
- LanguageLevel 2 includes all features of LanguageLevel 1. Additionally, it supports a one-operand form of the **image** operator in which the operand is an *image dictionary*, providing a more general means for specifying the image's characteristics. Other LanguageLevel 2 features include 12-bit component values, direct use of files or strings as data sources, interpretation of sample values in arbitrary color spaces (such as CIE-based), and additional decoding and rendering options.
- All implementations support the **imagemask** operator, which paints the current color through a *stencil mask* specified as a bitmap (see "Stencil Masking" on page 302). However, specification of the operands using an image dictionary is a LanguageLevel 2 feature.
- LanguageLevel 3 supports two additional forms of masking, using *explicit masks* and *color key masks.* These features are described in the sections "Explicit Masking" on page 303 and "Color Key Masking" on page 307.

There are often several ways to paint a given image, depending on the level of language features to be used. Fortunately, most properties of images do not depend on how painting is invoked or how operands are represented. The sections that follow frequently refer to specific features, such as **colorimage** or image dictionaries; see the summary above to determine which features are available in a particular PostScript implementation.

4.10.1 Image Parameters

The properties of an image—resolution, orientation, scanning order, and so forth—are entirely independent of the characteristics of the raster output device on which the image is to be rendered. The PostScript interpreter usually renders images by a sampling and halftoning technique that attempts to approximate the color values of the source as accurately as possible. The actual accuracy achieved depends on the resolution and other properties of the output device.

To paint an image, a PostScript program must specify four interrelated items:

- The format of the source image: number of columns (width), number of rows (height), number of color components per sample, and number of bits per color component
- A data source capable of providing the image's sample data, which consists of *height* × *width* × *components* × *bits*/*component* bits of information
- The correspondence between coordinates in user space and coordinates in the source image space, defining the region of user space that will receive the image
- The mapping from component values in the source image to component values in the current color space

The PostScript program entirely controls these four aspects of image specification.

4.10.2 Sample Representation

The source format for an image can be described by four parameters: *width*, *height*, *components*, and *bits*/*component*. A PostScript program specifies *width*, *height*, and *bits*/*component* explicitly. The interpreter infers the *components* parameter as follows:

- With the five-operand form of the **image** operator and with **imagemask**, *components* is always 1.
- With the one-operand (image dictionary) form of the **image** operator, *components* is the number of components in the current color space (see Section 4.8, "Color Spaces").
- With the **colorimage** operator, *components* is specified explicitly as the *ncomp* operand.

Sample data is represented as a stream of bytes, interpreted as 8-bit integers in the range 0 to 255, obtained from some data source (either returned from a procedure or read from a file or string). The bytes constitute a continuous bit stream, with the high-order bit of each byte first. This bit stream is in turn divided into units of *bits*/*component* bits each, ignoring byte boundaries. Sample values of 12 bits straddle byte boundaries; other sizes never do. Each unit encodes a color component value, given with the high-order bit first.

Each row of the source image begins on a byte boundary. If the number of data bits per row is not a multiple of 8, the end of the row must be padded with extra bits to fill out the last byte. The PostScript interpreter ignores these padding bits.

Each source sample component is interpreted as an integer in the range 0 to $2^n - 1$, where n is the number of bits per component. The PostScript interpreter maps this to a color component value (equivalent to what could be used with operators such as **setgray** or **setcolor**) by one of two methods:

- With the five-operand form of **image**, and with all forms of **colorimage**, the integer 0 maps to the number 0.0, the integer $2^n - 1$ maps to the number 1.0, and intermediate values map linearly to numbers between 0.0 to 1.0.
- With the one-operand (dictionary) form of **image**, the mapping is specified explicitly by the **Decode** entry in the image dictionary.
- With **imagemask**, image samples do not represent color values, so mapping is not relevant (see Section 4.10.6, "Masked Images").

The imaging operators (**image**, **colorimage**, and **imagemask**) can obtain sample data from any of three types of object:

- *Procedure.* Whenever the interpreter requires additional sample data, it calls the procedure, which is expected to return a string containing some more data. The amount of data returned by each call is arbitrary. However, returning one or more complete rows at a time simplifies programming, especially when reading image data that appears in-line in a PostScript program. This is the only type of data source permitted in LanguageLevel 1.
- *File.* The interpreter simply reads data from the file as necessary. Note that the file can be filtered to perform some kind of decoding or decompression (see Section 3.8.4, "Filters"). This type of data source is a LanguageLevel 2 feature.
- *String.* The interpreter simply reads data from the string, reusing the string as many times as necessary to satisfy the needs of the imaging operation. This type of data source is a LanguageLevel 2 feature, though equivalent behavior can be obtained in LanguageLevel 1 by providing a procedure that simply returns the same string each time it is called.

Data sources for images are much the same as those for filters; for further elaboration on their semantics, see Section 3.13.1, "Data Sources and Targets." When reading from a data source causes a PostScript procedure to be invoked, that pro-

cedure must not do anything to disturb the ongoing imaging operation, such as alter the graphics state or image dictionary or initiate a new imaging operation.

A data source can end prematurely. This occurs if a procedure returns a zero-length string or a file encounters end-of-file. If a data source ends before all samples have been read, the remainder of the image that would have been painted by the missing samples is left unpainted. If the last source row is incomplete—that is, if the data source ends in the middle of a row—the partial source row may be discarded and not painted.

When there are multiple color components per sample (the value of *components* is greater than 1), the source data can be organized in either of two ways:

- *Single data source.* All color components are obtained from the same source, interleaved sample by sample. For example, in a three-component RGB image, the red, green, and blue components for one sample are followed by the red, green, and blue components for the next sample.
- *Multiple data sources.* Each color component is obtained from a separate source—for example, all red components from one source, all green components from a second, and all blue components from a third. If the data sources are strings, they must all be of the same length. If they are procedures, they must all return strings of the same length on any given call. The interpreter calls each procedure in turn, in the order in which they are specified as operands to the **colorimage** operator or in which they appear in the image dictionary's **DataSource** array. The procedures may read data from the same file, but they must return their results in separate strings, since the interpreter does not copy this data elsewhere in the course of assembling the values for a single sequence of samples. If the data sources are files, the interpreter may read unpredictable amounts of data from them in unpredictable orders, so they must be completely independent of each other—that is, they must not be the same file, nor filters that ultimately read from the same file.

A PostScript program specifies which organization to use by means of the *multi* operand of the **colorimage** operator or the **MultipleDataSources** entry in the image dictionary. Figure 4.22 illustrates some typical organizations for data sources. It also shows the image sample decoding operation.

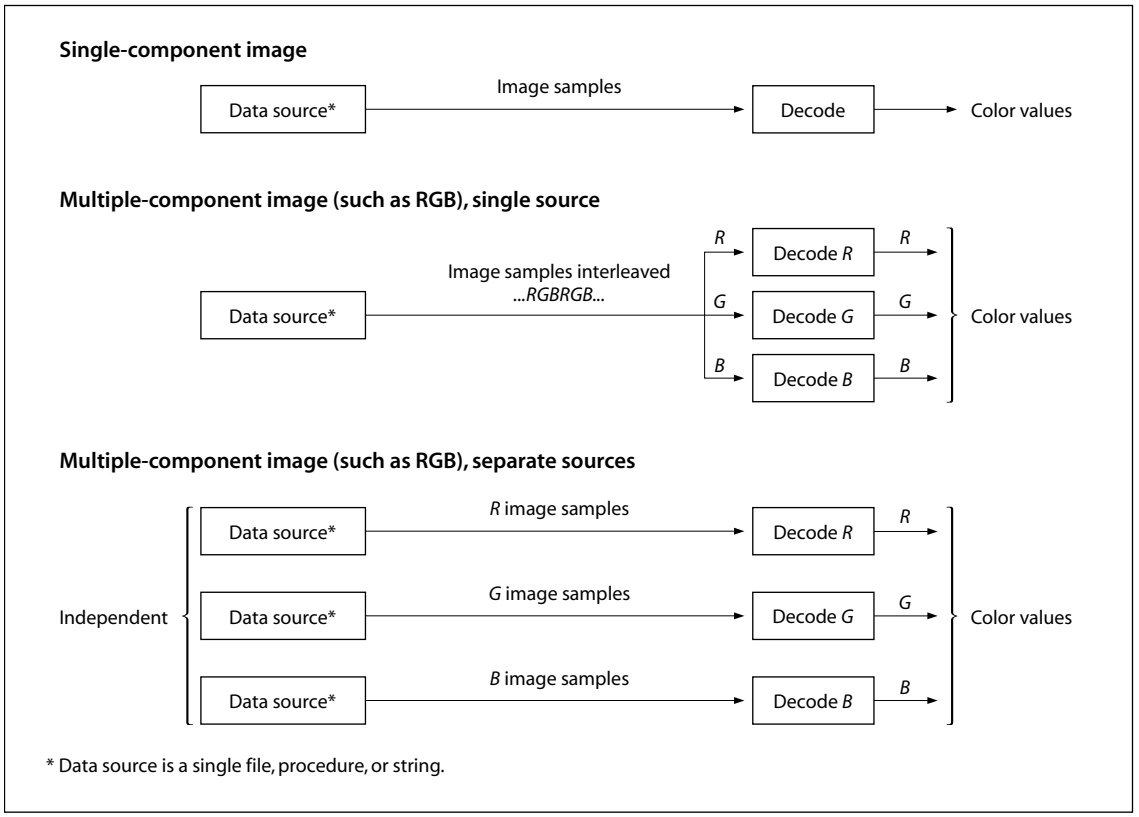


FIGURE 4.22 *Image data organization and processing*

4.10.3 Source Coordinate System

The image operators impose a coordinate system on the source image. They consider the source image to be a rectangle h units high and w units wide. Each sample occupies one square unit. The origin $(0, 0)$ is in the lower-left corner, with coordinates ranging from 0 to w horizontally and 0 to h vertically.

The image operators assume that the sample data they receive from their data source is ordered by row, with the horizontal coordinate varying most rapidly. The lower-left corner of the first sample is at coordinates $(0, 0)$, the second at $(1, 0)$, and so on through the last sample of the first row, whose lower-left corner is at $(w - 1, 0)$ and whose lower-right corner is at $(w, 0)$. The next samples after that are at coordinates $(0, 1)$, $(1, 1)$, and so on, until the final sample of the image,

whose lower-left corner is at $(w-1, h-1)$ and whose upper-right corner is at (w, h). Figure 4.23 illustrates the organization of the source coordinate system. The numbers inside the squares indicate the order of the samples, counting from 0.

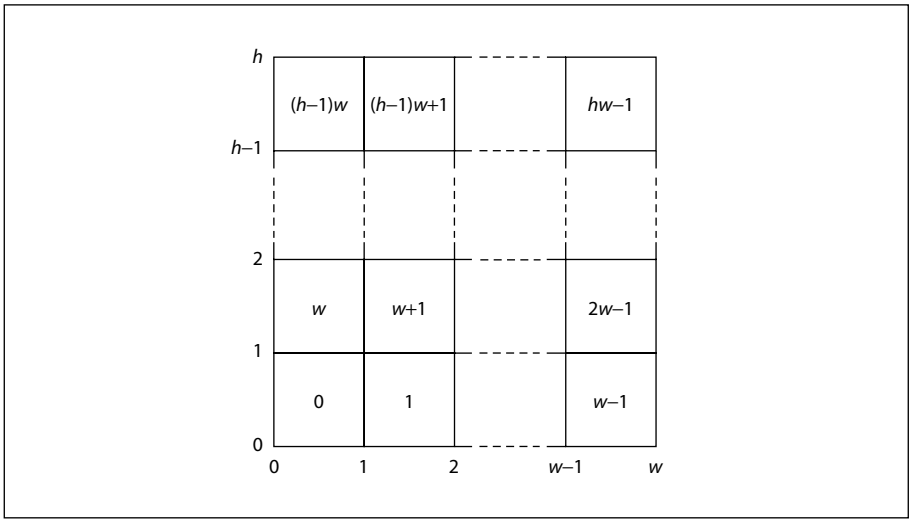


FIGURE 4.23 *Source image coordinate system*

The source coordinate system and scanning order imposed by the image operators do not preclude using different conventions in the actual source image. Coordinate transformation can be used to map other conventions into the PostScript convention.

The correspondence between this source image coordinate system (or *image space*) and user space is specified by a special matrix. This is a general linear transformation that defines a mapping from user space to image space; that is, it transforms user space coordinates to the corresponding coordinates in image space. It can include translation, rotation, reflection, and shearing (see Section 4.3, "Coordinate Systems and Transformations"). The matrix is provided in one of two ways:

- In the five-operand forms of **image** and **imagemask** and in all forms of **colorimage**, there is a separate *matrix* operand.
- In image dictionaries, there is a required **ImageMatrix** entry.

Although it is possible to map directly from current user space to image space by defining the image matrix appropriately, it is easier to think about the transformation by dividing it into two stages (see Figure 4.24):

1. The image matrix maps the unit square of user space, bounded by user coordinates (0, 0) and (1, 1), to the boundary of the source image in image space.
2. The current transformation matrix (CTM) maps the unit square of user space to the rectangle or parallelogram on the page that is to receive the image.

This is just a convention, but it is a useful one that is recommended. Under this convention, the image matrix is used solely to describe the image itself, independently of how it is to be positioned, oriented, and scaled on a particular page. It defines an idealized image space consisting of a unit square that conforms to PostScript conventions for coordinate system and scanning order. A program can then map this idealized image space into current user space by altering the CTM in straightforward ways.

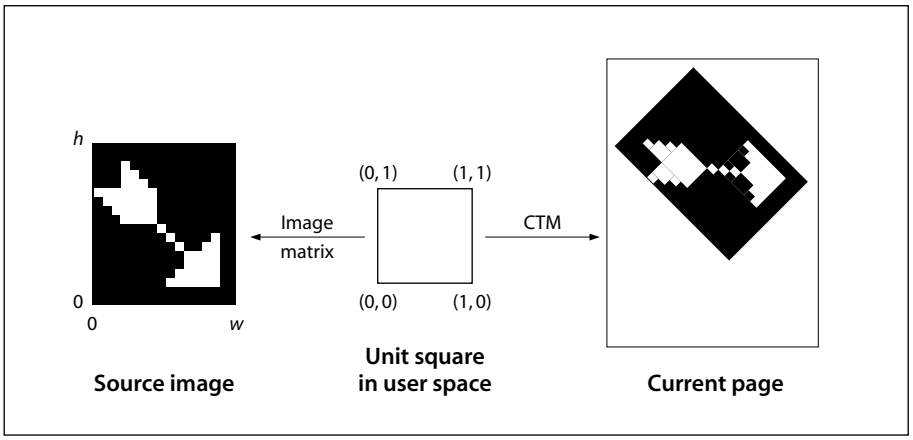


FIGURE 4.24 *Mapping the source image*

An image that happens to use the PostScript conventions (scanning from left to right and bottom to top) can be described by the image matrix

[*w* 0 0 *h* 0 0]

(where *w* and *h* are the width and height of the image, respectively). An image that is scanned from left to right and top to bottom (a commonly used order) is described by the image matrix

[*w* 0 0 –*h* 0 *h*]

Images scanned in other common orders can be described in similar ways.

An image that has been mapped into the unit square in this way can then be placed on the output page in the desired position, orientation, and size by invoking PostScript operators that transform user space: **translate**, **rotate**, and **scale**. For example, to map such an image into a rectangle whose lower-left corner is at coordinates (100, 200), that is rotated 45 degrees counterclockwise, and that is 150 units wide and 80 high, a program can execute

```
100 200 translate
45 rotate
150 80 scale
```

before invoking the **image**, **colorimage**, or **imagemask** operator. This works for any image that has been mapped into the unit square by an appropriate image matrix. Of course, if the aspect ratio (width to height) of the source image in this example were different from 150:80, the result would be distorted.

4.10.4 Images and Color Spaces

The color samples in an image are interpreted according to some color space (see Section 4.8, "Color Spaces"). The color space to be used depends on how imaging is invoked:

- The five-operand form of the **image** operator ignores the current color space and always interprets color samples according to the **DeviceGray** color space. It does not alter the current color space parameter in the graphics state.
- The **colorimage** operator always interprets color samples according to the **DeviceGray**, **DeviceRGB**, or **DeviceCMYK** color space, depending on whether its *ncomp* operand is 1, 3, or 4. It neither reads nor alters the current color space parameter in the graphics state.
- The one-operand (dictionary) form of the **image** operator interprets color samples according to the current color space. The number of components per

sample and the interpretation of the component values depend on the color space. This form of **image** can be used with any color space except **Pattern**.

- The **imagemask** operator always interprets its source data as a stencil mask for applying the current color in the current color space (see "Stencil Masking" on page 302). This works for any color space.

4.10.5 Image Dictionaries

The **image** and **imagemask** operators (but not **colorimage**) have a one-operand form *(LanguageLevel 2)* in which all imaging parameters are bundled together in an *image dictionary.* This arrangement provides more flexibility than the five-operand form of **image** or any form of **colorimage**. The following features are available only by means of image dictionaries:

- Use of arbitrary color spaces, such as **CIEBasedABC** or **Separation** spaces
- User-defined decoding of image sample values
- Interpolation between samples
- Explicit masking and color key masking *(LanguageLevel 3)*

Every image dictionary contains an entry named **ImageType**, which indicates the general organization of the dictionary and the type of image it represents. The most straightforward are type 1 dictionaries *(LanguageLevel 2)*. When used with the **image** operator, type 1 defines an opaque rectangular image to be painted in its entirety, completely overlaying any previous marks already existing on the page. Two more image types, 3 and 4 *(LanguageLevel 3)*, represent *masked images.* Images of these types may include transparent areas, allowing previously existing marks to show through. Type 3 and 4 image dictionaries are discussed in Section 4.10.6, "Masked Images."

Note: *Type 1 dictionaries can be used with the* ***imagemask*** *operator to achieve a form of transparent masking known as* stencil masking. *They are subject to certain restrictions when used in this way, however, as detailed below.*

Type 1 Image Dictionaries

Table 4.20 lists the entries in a type 1 image dictionary, some of which are required and some optional. There are many relationships among these entries,

and the current color space may limit the choices for some of them. Attempting to use an image dictionary whose entries are inconsistent with each other or with the current color space will cause a **rangecheck** error; if any of the required entries are missing or of the wrong type, an **undefined** or **typecheck** error will result.

TABLE 4.20 Entries in a type 1 image dictionary

KEY	TYPE	VALUE
ImageType	integer	*(Required)* A code identifying the image type that this dictionary describes; must be 1 for an opaque image.
Width	integer	*(Required)* The width of the source image, in samples.
Height	integer	*(Required)* The height of the source image, in samples.
ImageMatrix	array	*(Required)* An array of six numbers defining a transformation from user space to image space.
MultipleDataSources	boolean	*(Optional)* A flag indicating whether the image samples are provided through a separate data source for each color component (*true*) or packed into one data stream, interleaved sample by sample (*false*). In an image dictionary used with **imagemask**, this entry must be *false* or absent. Default value: *false*.
DataSource	(various)	*(Required)* The source from which image samples are to be taken. If **MultipleDataSources** is *false* or absent, **DataSource** must be a single file, procedure, or string. If **MultipleDataSources** is *true*, **DataSource** must be an array of *n* such data sources, where *n* is the number of color components in the current color space.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Only a single number may be specified; the number of bits is the same for all color components. Allowed values are 1, 2, 4, 8, and 12. In an image dictionary used with **imagemask**, the value of this entry must be 1.
Decode	array	*(Required)* An array of numbers describing how to map image samples into the range of values appropriate for the current color space; see "Sample Decoding," below. The length of the array must be twice the number of color components in the current color space. In an image dictionary used with **imagemask**, the value of this entry must be either `[0 1]` or `[1 0]`.
Interpolate	boolean	*(Optional)* A flag indicating whether image interpolation is to be performed; see "Image Interpolation," below. Default value: *false*.

The following sections describe the meaning and use of some of these entries in more detail. All of this information applies to image dictionaries used with the **image** operator, and most of it also applies to those used with the **imagemask** operator (see "Stencil Masking" on page 302).

Sample Decoding

The sample data in an image dictionary's data source is initially decomposed into integers in the range 0 to $2^n - 1$, where n is the value of the dictionary's **BitsPerComponent** entry. There is one such integer for each component of a given color sample; the number of components depends on the current color space.

The **Decode** array in the image dictionary specifies a linear mapping of an integer component value to a number that would be appropriate as an operand to the **setcolor** operator in the current color space. For each color component, **Decode** specifies a minimum and maximum output value for the mapping. The linear mapping is defined by

$$c = D_{\min} + \left(i \times \frac{D_{\max} - D_{\min}}{2^n - 1} \right)$$

where

n is the value of **BitsPerComponent**
i is the input value, in the range 0 to $2^n - 1$
$D_{\min}$ and $D_{\max}$ are the parameters in the **Decode** array
c is the output value, to be interpreted as a color component

In other words, an input value of 0 will be mapped to $D_{\min}$, an input value of $2^n - 1$ will be mapped to $D_{\max}$, and intermediate input values will be mapped linearly to values between $D_{\min}$ and $D_{\max}$.

The numbers in the **Decode** array are interpreted in pairs, with successive pairs applying to successive components of the current color space in their standard order. Table 4.21 lists recommended **Decode** arrays for use with the various color spaces.

TABLE 4.21 Typical Decode arrays

COLOR SPACE	Decode ARRAY
DeviceGray	[0 1]
DeviceRGB	[0 1 0 1 0 1]
DeviceCMYK	[0 1 0 1 0 1 0 1]
CIEBasedABC	[0 1 0 1 0 1]
CIEBasedA	[0 1]
CIEBasedDEF	[0 1 0 1 0 1]
CIEBasedDEFG	[0 1 0 1 0 1 0 1]
DeviceN	[0 1 0 1 … 0 1] (one pair for each color component)
Separation	[0 1]
Indexed	[0 N], where $N = 2^n - 1$
Pattern	(**image** not permitted)

For most color spaces, the **Decode** arrays listed in the table map into the full range of allowed component values. For all CIE-based color spaces, the suggested arrays map to component values in the range 0.0 to 1.0. This is typical for the class of calibrated gray or RGB color spaces, but the appropriate values actually depend on how the color spaces have been parameterized. For an **Indexed** color space, the suggested **Decode** array ensures that component values that index a color table are passed through unchanged.

It is possible to specify a mapping that *inverts* sample color intensities, by specifying a $D_{\min}$ value greater than $D_{\max}$. For example, if the current color space is **DeviceGray** and the **Decode** array is [1 0], an input value of 0 will be mapped to 1.0 (white), while an input value of $2^n - 1$ will be mapped to 0.0 (black).

The $D_{\min}$ and $D_{\max}$ parameters for a color component are not required to fall within the range of values allowed for that component. For instance, if an application uses 6-bit numbers as its native image sample format, it can send those samples to the PostScript interpreter in 8-bit form, setting the two unused high-order bits of each sample to 0. The image dictionary should then specify a **Decode** array of [0.0 4.04762], which maps input values from 0 to 63 into the

range 0.0 to 1.0. If an output value falls outside the range allowed for a component, it will automatically be adjusted to the nearest allowed value.

Image Interpolation

When the resolution of a source image is significantly lower than that of the output device, each source sample covers many device pixels. This can cause images to appear "jaggy" or "blocky." These visual artifacts can be reduced by applying an *image interpolation* algorithm during rendering. Instead of painting all pixels covered by a source sample with the same color, image interpolation attempts to produce a smooth transition between adjacent sample values. Because it may increase the time required to render the image, image interpolation is disabled by default; setting the **Interpolate** entry in the image dictionary to *true* for either **image** or **imagemask** enables it.

Note: *The interpolation algorithm is implementation-dependent and not under PostScript program control. Image interpolation may not always be performed for some classes of image or on some output devices.*

4.10.6 Masked Images

In LanguageLevel 1 or 2, images painted with the **image** or **colorimage** operator mark all affected areas of the page as if with opaque paint (see Section 4.1, "Imaging Model"). All portions of the image, whether black, white, gray, or color, completely obscure any marks that may previously have existed in the same place on the page.

In the graphic arts industry and page layout applications, however, it is common to crop or "mask out" the background of an image and then place the masked image on a different background, allowing the existing background to show through the masked areas. A number of PostScript features are available for achieving such masking effects:

- The **imagemask** operator, available in all LanguageLevels, uses a monochrome (black-and-white) image as a *stencil mask* for painting in the current color.
- Type 3 image dictionaries *(LanguageLevel 3)* include an *explicit mask* specifying which areas of the image to paint and which to mask out.

- Type 4 image dictionaries *(LanguageLevel 3)* specify a color or range of colors to be masked out wherever they occur within the image; this technique is known as *color key masking.*

Note: *Although type 3 and 4 image dictionaries are a LanguageLevel 3 feature, their effects are commonly simulated in lower LanguageLevels by defining a clipping path enclosing only those of an image's samples that are to be painted. However, implementation limits can cause* ***limitcheck*** *errors if the clipping path is very complex (or if there is more than one clipping path). An alternative way to achieve the effect of a type 3 image dictionary in LanguageLevel 2 is to define the image being clipped as a pattern, make it the current color, and then paint it with the* ***imagemask*** *operator. In any case, the LanguageLevel 3 features allow masked images to be placed on the page without regard to the complexity of the clipping path.*

Stencil Masking

The **imagemask** operator operates on a monochrome image, in which each sample is specified by a single bit. However, instead of painting the image in opaque black and white, **imagemask** treats it as a *stencil mask* that is partly opaque and partly transparent. Sample values in the image do not represent black and white pixels; rather, they designate places on the page that should either be marked with the current color or masked out (not marked at all). Areas that are masked out retain their former contents. The effect is like applying paint in the current color through a cut-out stencil: a sample value of 1 in the image permits the paint to reach the page and a 0 masks it out, or vice versa.

A program invokes **imagemask** in much the same way as **image**; most of the parameters have equivalent meanings. As with **image**, there is a five-operand form available in all LanguageLevels and a one-operand image dictionary form that is a LanguageLevel 2 feature. **imagemask** differs from **image** in the following significant ways:

- The number of components per sample is always 1, regardless of the current color space, because sample values represent masking properties rather than colors.
- The number of bits per component is always 1. In its five-operand form, **imagemask** has no *bits/sample* operand; in the one-operand (image dictionary) form, the dictionary's **BitsPerComponent** value must be 1.

- The five-operand form of **imagemask** includes a *polarity* operand that determines how the source samples are to be interpreted. If *polarity* is *true*, a sample value of 1 designates a painted sample and 0 designates a masked (unpainted) sample; if *false*, these meanings are reversed. The one-operand form of **imagemask** uses the **Decode** entry in the image dictionary for the same purpose: **Decode** arrays of `[1 0]` and `[0 1]` correspond to *polarity* values of *true* and *false*, respectively.

One of the most important uses of stencil masking is for painting character glyphs represented as bitmaps. Using such a glyph as a stencil mask transfers only its "black" bits to the page, while leaving the "white" bits (which are really just background) unchanged. For reasons discussed in Section 5.5, "Font Cache," **imagemask** rather than **image** should almost always be used to paint glyph bitmaps.

Note: *If image interpolation is requested during stencil masking, the effect is to smooth the edges of the mask, not to interpolate the painted color values (see "Image Interpolation" on page 301). This can minimize the "jaggy" appearance of a low-resolution stencil mask.*

Explicit Masking

Type 3 image dictionaries combine a sampled image with an *explicit mask*. The image and mask need not have the same resolution, but their positions on the page must coincide; that is, they must overlay each other. The mask is treated essentially the same as the stencil mask supplied to the **imagemask** operator: it indicates which places on the page are to be painted and which are to be masked out (left unchanged). Unmasked areas are painted with the corresponding portions of the sampled image; masked areas are not.

Table 4.22 lists the entries in a type 3 image dictionary. The properties of the image and the mask are defined by two subsidiary dictionaries, an *image data dictionary* (**DataDict**) and a *mask dictionary* (**MaskDict**). Both are ordinary type 1 image dictionaries with a few additional restrictions on their contents, as detailed in Tables 4.23 and 4.24. The only other significant entry is **InterleaveType**, which specifies the format in which the image and mask data are organized: the mask samples may be included in the same data source with the image samples or provided separately, depending on the interleave type. In general, any inconsistency among the three dictionaries or violation of the stated restrictions will result in a **typecheck** error.

TABLE 4.22 Entries in a type 3 image dictionary

KEY	TYPE	VALUE
ImageType	integer	*(Required)* A code identifying the image type that this dictionary describes; must be 3 for explicit masking.
DataDict	dictionary	*(Required)* A modified type 1 image dictionary defining the contents of the image (see Table 4.23).
MaskDict	dictionary	*(Required)* A modified type 1 image dictionary defining the image's mask (see Table 4.24).
InterleaveType	integer	*(Required)* A code indicating how the image and mask samples are organized: 1 *Interleaved by sample.* Image and mask samples are combined into a single data source, identified by the **DataSource** entry in the image data dictionary; the mask dictionary must contain no **DataSource** entry. Components are interleaved sample by sample, with the mask component preceding all color components; for example, in the **DeviceRGB** color space, each sample would consist of a mask component followed by three color components (red, green, and blue). The mask sample must have the same number of bits as each color component of the image sample, with all bits set to the same value (that is, either all 0 or all 1); any other value will be treated as if the bits were all 1. 2 *Interleaved by row.* Image and mask samples are combined into a single data source, identified by the **DataSource** entry in the image data dictionary; the mask dictionary must contain no **DataSource** entry. Mask and image data are organized into *interleave blocks* whose format is determined by the **Height** entries in the image data and mask dictionaries. The heights given in the two dictionaries may differ, with the restriction that one must be an integral multiple of the other. Each interleave block thus consists of either one row of mask data followed by one or more rows of image data, or one or more rows of mask data followed by one row of image data, according to the ratio of the heights specified in the two dictionaries. All interleave blocks have the same format. Within each block, all of the mask data precedes all of the image data. Mask data is always one bit per sample, regardless of the number of bits per sample in the image data. Within the image data, color compo-

nents are interleaved on a sample-by-sample basis. Each row of mask and image samples is padded separately to byte boundaries.

3 *Separate data sources*. Image and mask samples are provided through separate data sources, identified by the **DataSource** entries in the image data dictionary and the mask dictionary, respectively. The color components of the image samples themselves may in turn be interleaved or separate, depending on the value of the **MultipleDataSources** entry in the image data dictionary. The width and height of the mask are independent of those of the image, but the image and mask must have the same orientation and placement.

TABLE 4.23 Entries in an image data dictionary

KEY	TYPE	VALUE
ImageType	integer	*(Required)* A code identifying the image type that this dictionary describes; must be 1 for an image data dictionary.
Width	integer	*(Required)* The width of the source image, in samples. In interleave type 1, this value must equal that of the **Width** entry in the mask dictionary.
Height	integer	*(Required)* The height of the source image, in samples. In interleave type 1, this value must equal that of the **Height** entry in the mask dictionary. In interleave type 2, the image and mask heights may differ, with the restriction that one must be an integral multiple of the other. In interleave type 3, the heights of the image and mask are independent.
ImageMatrix	array	*(Required)* An array of six numbers defining a transformation from user space to image space.
MultipleDataSources	boolean	*(Optional)* A flag indicating whether the image samples are provided through a separate data source for each color component (*true*) or packed into one data stream, interleaved sample by sample (*false*). If this entry is *true*, the interleave type in the main image dictionary (Table 4.22) must be 3; that is, the image's mask data must also be provided through a separate data source, designated by the **DataSource** entry in the mask dictionary (Table 4.24). For interleave types 1 and 2, **MultipleDataSources** must be *false*. Default value: *false*.
DataSource	(various)	*(Required)* The source from which image samples are to be taken. If **MultipleDataSources** is *false* or absent, **DataSource** must be a single file, procedure, or string. If **MultipleDataSources** is *true*, **DataSource** must be an array of *n* such data sources, where *n* is the number of color components in the current color space. For interleave types 1 and 2, the designated data source will also include mask samples interleaved with the

		source samples in the manner implied by the interleave type (see Table 4.22); in this case, the mask dictionary (Table 4.24) must contain no **DataSource** entry.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Only a single number may be specified; the number of bits is the same for all color components. Allowed values are 1, 2, 4, 8, and 12.
Decode	array	*(Required)* An array of numbers describing how to map image samples into the range of values appropriate for the current color space; see "Sample Decoding" on page 299. The length of the array must be twice the number of color components in the current color space.
Interpolate	boolean	*(Optional)* A flag indicating whether image interpolation is to be performed on the source data; see "Image Interpolation" on page 301. Default value: *false*.

TABLE 4.24 Entries in a mask dictionary

KEY	TYPE	VALUE
ImageType	integer	*(Required)* A code identifying the image type that this dictionary describes; must be 1 for a mask dictionary.
Width	integer	*(Required)* The width of the mask, in samples. In interleave type 1, this value must equal that of the **Width** entry in the image data dictionary.
Height	integer	*(Required)* The height of the mask, in samples. In interleave type 1, this value must equal that of the **Height** entry in the image data dictionary. In interleave type 2, the image and mask heights may differ, with the restriction that one must be an integral multiple of the other. In interleave type 3, the heights of the image and mask are independent.
ImageMatrix	array	*(Required)* An array of six numbers defining a transformation from user space to image space. This matrix must align the corners of the mask with the corresponding corners of the image, so that they coincide in user space, and must perform any scaling needed to compensate for differences in the dimensions of the mask and the image.
MultipleDataSources	boolean	*(Optional)* If present, must be *false.* Default value: *false*.
DataSource	(various)	*(Required for interleave type 3)* The source (a single file, procedure, or string) from which mask samples are to be taken. This entry must be absent for interleave type 1 or 2.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. In interleave type 1, this value must equal that of the **BitsPerComponent**

		entry in the image data dictionary. In interleave type 2 or 3, the value of this entry must be 1.
Decode	array	*(Required)* An array of two numbers describing how to map mask samples into the appropriate range of values; see "Sample Decoding" on page 299. A decoded value of 0 designates a sample to be painted; a decoded value of 1 designates a sample that is to be masked out (not painted).
Interpolate	boolean	*(Optional)* A flag indicating whether image interpolation is to be performed on the mask; see "Image Interpolation" on page 301. Default value: *false*.

Color Key Masking

Type 4 image dictionaries (Table 4.25) are identical to type 1 with one additional entry, **MaskColor**, specifying a color or range of colors to be masked out. Samples in the image that match this color or fall within this range are not painted, allowing the existing background to show through. The effect is similar to that of the video technique known as *chroma-key.*

TABLE 4.25 Entries in a type 4 image dictionary

KEY	TYPE	VALUE
ImageType	integer	*(Required)* A code identifying the image type that this dictionary describes; must be 4 for color key masking.
MaskColor	array	*(Required)* An array of integers specifying the color to be masked. The array may contain either n or $2 \times n$ integers, where n is the number of components required to specify a color in the current color space. If n integers are given, they specify the masked color exactly; any image sample whose color components match the values in the array will be masked out (not painted). If the array contains $2 \times n$ integers, each pair of consecutive integers specify a range of values for the corresponding color component; an image sample will be masked if each of its component values falls within the specified range. Image samples are compared with the mask color as they are read from the data source. This comparison occurs before the application of the decode mapping specified by the **Decode** array.
Width	integer	*(Required)* The width of the source image, in samples.
Height	integer	*(Required)* The height of the source image, in samples.

ImageMatrix	array	*(Required)* An array of six numbers defining a transformation from user space to image space.
MultipleDataSources	boolean	*(Optional)* A flag indicating whether the image samples are provided through a separate data source for each color component (*true*) or packed into one data stream, interleaved sample by sample (*false*). Default value: *false*.
DataSource	(various)	*(Required)* The source from which image samples are to be taken. If **MultipleDataSources** is *false* or absent, **DataSource** must be a single file, procedure, or string. If **MultipleDataSources** is *true*, **DataSource** must be an array of *n* such data sources, where *n* is the number of color components in the current color space. The use of a **DCTDecode** filter as the data source of a type 4 image dictionary is not recommended. **DCTDecode** is a "lossy" filter, meaning that its output is only an approximation of the original input data. This can lead to slight changes in the color values of image samples, possibly causing samples that were intended to be masked to be unexpectedly painted instead, in colors slightly different from the mask color.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Only a single number may be specified; the number of bits is the same for all color components. Allowed values are 1, 2, 4, 8, and 12.
Decode	array	*(Required)* An array of numbers describing how to map image samples into the range of values appropriate for the current color space; see "Sample Decoding" on page 299. The length of the array must be twice the number of color components in the current color space.
Interpolate	boolean	*(Optional)* A flag indicating whether image interpolation is to be performed; see "Image Interpolation" on page 301. When used in combination with color key masking, image interpolation can cause unexpected or unwanted visual artifacts in the resulting image. To prevent such artifacts, the **Interpolate** flag should normally be set to *false*. Default value: *false*.

4.10.7 Using Images

This section gives some simple examples that demonstrate typical uses of images. The examples are incomplete: they cannot show the image data itself, since it is very bulky. For further information about the imaging operators, see the operator descriptions in Chapter 8.

Monochrome Image

Example 4.24 uses the **image** operator to paint a grayscale image, using facilities available in all LanguageLevels. This program paints an image 256 samples wide by 256 high, at 8 bits per sample. It positions the image with its lower-left corner at coordinates (45, 140) in current user space and scales it to a width and height of 132 user space units. The image data is stored with the first sample in the upper-left corner, so the program uses the image matrix to match its coordinate system with the normal PostScript convention (see Section 4.10.3, "Source Coordinate System").

Example 4.24

```
/picstr 256 string def                  % String to hold image data

45  140  translate                      % Locate lower-left corner of image
132  132  scale                         % Map image to 132-unit square

256  256  8                            % Dimensions of source image
[256  0  0  –256  0  256]               % Map unit square to source
{    currentfile                        % Read image data from program file
     picstr readhexstring pop
}
image

4c47494b4d4c524c4d50535051554c5152…
… Total of 131,072 hexadecimal digits of image data, representing 65,536 samples …
```

The image data appears in-line in the PostScript program. This is the most common way to access image data in a document. Only occasionally will a program refer to image data stored elsewhere—in a file, for instance. The image data is represented in hexadecimal rather than 8-bit binary, to maximize the document's portability.

The program specifies a procedure as the data source. Each time the procedure is called by the **image** operator, it invokes **readhexstring** to read one row of sample data into a string, which it then returns to **image**. It reuses this same string during every call. Reading one row at a time is not necessary, but it simplifies the programming. If the procedure reads multiple rows at a time, or an amount of data that is not a multiple of the image's width, it must take special care not to read past the end of the image data the last time it is called by **image**. Doing so would cause some program text following the image data to be lost.

With most images, it is very important to read the image data incrementally, as shown in this example. Attempting to read the entire image into a single string or to represent it as a PostScript string literal would risk exceeding implementation limits or exhausting available virtual memory.

Color Image with Single Source

As indicated earlier, color images with multiple components per sample can be organized in two ways: with all components interleaved in a single source or obtained from separate sources. The first organization is the only one that is useful for images whose data is provided in-line in a PostScript program. The second organization is limited to situations in which the separate components are stored elsewhere, such as in separate files that can be read in parallel.

Example 4.25 illustrates the use of the **colorimage** operator to paint an image consisting of interleaved RGB data from a single source. This example works in LanguageLevel 2, and also in those LanguageLevel 1 implementations that have the CMYK color extensions.

Example 4.25

```
/picstr 768 string def                  % String to hold 256 RGB samples

45  140  translate                      % Locate lower-left corner of image
132  132  scale                         % Map image to 132-unit square

256  256  8                            % Dimensions of source image
[256  0  0  –256  0  256]               % Map unit square to source
{   currentfile                        % Read image data from program file
    picstr readhexstring pop
}
false 3                                 % Single data source, 3 colors
colorimage

94a1bec8c0b371a3a5c4d281 …
… Total of 393,216 hexadecimal digits of image data, representing 65,536 samples …
```

This **colorimage** example is superficially similar to the **image** example given earlier. The main change is that two additional operands are supplied to **colorimage**, specifying that the image has a single data source and three components. The image data consists of 8-bit red, green, and blue color components for each sample in turn.

Image Dictionary

Example 4.26 produces the same output as Example 4.25, but it uses a type 1 image dictionary and other LanguageLevel 2 features. In this program, the image data source is a file instead of a procedure. The file is a *filtered file* that converts the hexadecimally encoded data from **currentfile** to binary form. For an explanation of this and an example of how to obtain image data that has been compressed, see Section 3.8.4, "Filters."

Example 4.26

```
/DeviceRGB setcolorspace                        % How color values will be interpreted

45 140 translate                                % Locate lower-left corner of image
132 132 scale                                   % Map image to 132-unit square

<<                                              % Start image dictionary
      /ImageType 1
      /Width 256                                % Dimensions of source image
      /Height 256
      /BitsPerComponent 8
      /Decode [0 1 0 1 0 1]                     % Decode color values in normal way
      /ImageMatrix [256 0 0 –256 0 256]         % Map unit square to source
      /DataSource currentfile /ASCIIHexDecode filter  % Obtain in-line data through filter
>>                                              % End image dictionary
image

94a1bec8c0b371a3a5c4d281 …
… Total of 393,216 hexadecimal digits of image data, representing 65,536 samples …
>                                               % End-of-data marker for ASCIIHexDecode
```

CHAPTER 5

Fonts

THIS CHAPTER DESCRIBES the special facilities in the PostScript language for dealing with text—more generally, for representing characters with *glyphs* from *fonts*. A glyph is a graphical shape and is subject to all graphical manipulations, such as coordinate transformation. Because of the importance of text in most page descriptions, the PostScript language provides higher-level facilities that permit a program to describe, select, and render glyphs conveniently and efficiently.

The first section is a general description of how fonts are organized and accessed. This description covers all normal uses of fonts that are already installed.

The information in subsequent sections is somewhat more complex, but is required only by programs with sophisticated needs. These sections discuss the organization of font dictionaries, the encoding scheme that maps character codes to character names and glyph descriptions, the metric information available for fonts, the operation of the font cache, and the construction of new fonts.

Details of the individual PostScript operators are given in Chapter 8. All facilities are supported by LanguageLevel 1 except those specifically documented as LanguageLevel 2 or LanguageLevel 3 features. Some of the LanguageLevel 2 features are also available as part of composite font extensions; see Appendix A for details.

5.1 Organization and Use of Fonts

A *character* is an abstract symbol, whereas a *glyph* is a specific rendering of a character. For example, the glyphs A, **A**, and *A* are renderings of the abstract "A" character. Historically these two terms have often been used interchangeably in

computer typography (as evidenced by the names chosen for some PostScript operators and keys), but advances in this area have made the distinction more meaningful in recent times. Consequently this book distinguishes between characters and glyphs, though with some residual names that are inconsistent.

Glyphs are organized into *fonts.* A font defines glyphs for a particular character set; for example, the Helvetica and Times-Roman fonts define glyphs for a set of standard Latin characters. A font for use with the PostScript interpreter is prepared in the form of a program. When such a *font program* is introduced into a PostScript interpreter, its execution causes a *font dictionary* to come into existence and to be associated with a font name.

In the PostScript language, the term *font* refers to a font dictionary, through which the PostScript interpreter obtains *glyph descriptions* that generate glyphs. For each glyph to be painted, a program specifies a font dictionary and (usually) a character code to select the glyph description that represents the character. The glyph description, which is usually encoded in a special compact representation, consists of a sequence of graphics operators that produce the specific shape for that character in this font. To render a glyph, the PostScript interpreter executes the glyph description.

If you have experience with scan conversion of general shapes, you may be concerned about the amount of computation that this description seems to imply. However, this is only the abstract behavior of glyph descriptions and font programs, not how they are implemented. In fact, the PostScript font machinery works very efficiently.

5.1.1 The Basics of Showing Text

Example 5.1 illustrates the most straightforward use of a font. Suppose you want to place the text ABC 10 inches from the bottom of the page and 4 inches from the left edge, using 12-point Helvetica.

Example 5.1

```
/Helvetica findfont
12 scalefont setfont
288  720  moveto
(ABC) show
```

The four lines of this program perform the following steps:

1. Select the font to use.
2. Scale it to the desired size and install it as the font parameter in the graphics state.
3. Specify a starting position on the page.
4. Paint the glyphs for a string of characters there.

The following paragraphs explain these operations in more detail.

Each PostScript implementation includes a collection of fonts that are either built in or can be obtained automatically from sources such as disks or cartridges. A user can download additional fonts, and a PostScript program can define special fonts for its own use. The interpreter maintains a *font directory* associating the names of fonts, which are name objects, with their definitions, which are font dictionaries. The **findfont** operator takes the name of the font and returns on the operand stack a font dictionary containing all the information the PostScript interpreter needs to render any of that font's glyphs.

A font defines the glyphs for one standard size. This standard is so arranged that the nominal height of tightly spaced lines of text is 1 unit. In the default user coordinate system, this means the standard glyph size is 1 unit in user space, or 1/72 inch. The standard-size font must then be scaled to be usable.

The **scalefont** operator scales the glyphs in a font without affecting the user coordinate system. **scalefont** takes two operands: the original font dictionary and the desired scale factor. It returns a new font dictionary that defines glyphs in the desired size. It is possible to scale the user coordinate system with the coordinate system operators, but it is usually more convenient to encapsulate the desired size in the font dictionary. Another operator, **makefont**, applies more complicated linear transformations to a font.

In Example 5.1, the **scalefont** operator scales the Helvetica font left on the stack by **findfont** to a 12-unit size and returns the scaled font on the operand stack. The **setfont** operator establishes the resulting font dictionary as the font parameter in the graphics state.

Once the font has been selected, scaled, and set, it can be used to paint glyphs. The **moveto** operator (described in Section 4.4, "Path Construction") sets the

current position to the specified *x* and *y* coordinates—in units of 1/72 inch—in the default user coordinate system. This determines the position on the page at which to begin painting glyphs.

The **show** operator takes a string from the operand stack and paints the corresponding glyphs using the *current font* (usually the font parameter most recently established in the graphics state by **setfont**). The **show** operator treats each element of the string (an integer in the range 0 to 255) as a character code. Each code selects a glyph description in the font dictionary; the glyph description is executed to paint the desired glyph on the page. This is the behavior of **show** for base fonts, such as ordinary Latin text fonts; interpretation of the **show** string as a sequence of character codes is more complex for composite fonts, described in Section 5.10, "Composite Fonts."

Note: *What these steps produce on the page is not a 12-point glyph, but rather a 12-unit glyph, where the unit size is that of the user space at the time the glyphs are rendered on the page. If the user space is later scaled to make the unit size 1 centimeter, showing glyphs from the same 12-unit font will generate results that are 12 centimeters high.*

5.1.2 Selecting Fonts

Example 5.1 used PostScript operators in a direct way. It is usually desirable to define procedures to help the application that is generating the text. To illustrate this point, assume that an application is setting many independently positioned text strings and requires switching frequently among three fonts: Helvetica, Helvetica-Oblique, and Helvetica-Bold, all in a 10-point size. Example 5.2 shows the programming to do this, and Figure 5.1 shows the results.

Example 5.2

```
/FSD {findfont exch scalefont def} bind def     % In the document prolog: define
/SMS {setfont moveto show} bind def             % some useful procedures
/MS {moveto show} bind def

/F1  10  /Helvetica  FSD                        % At the start of the script: set up
/F2  10  /Helvetica-Oblique  FSD                % commonly used font dictionaries
/F3  10  /Helvetica-Bold  FSD
```

```
(This is in Helvetica.) 10 78 F1 SMS                % In the body of the script: show
(And more in Helvetica.) 10 66 MS                   % some text
(This is in Helvetica-Oblique.) 10 54 F2 SMS
(This is in Helvetica-Bold.) 10 42 F3 SMS
(And more Helvetica-Bold.) 10 30 MS
```

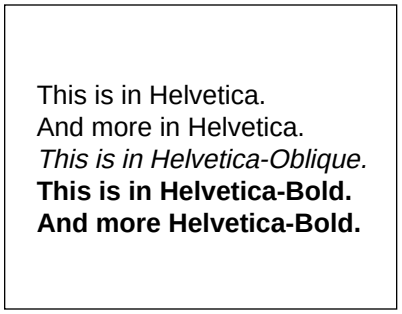


FIGURE 5.1 *Results of Example 5.2*

Several features of Example 5.2 are noteworthy. The document prolog defines three procedures:

- FSD takes a variable name, a scale factor, and a font name. It generates a font dictionary described by the font name and scale factor, then executes **def** to associate the font dictionary with the variable name. This assists in setting up fonts.
- SMS takes a string, a pair of coordinates, and a font dictionary; it shows the glyphs for the string starting at those coordinates, using the specified font.
- MS takes a string and a pair of coordinates; it shows the glyphs for the string at those coordinates, using the current font.

At the beginning of the document script, the program sets up font dictionaries and associates them with the names F1, F2, and F3. The body of the script shows text using the procedures and font dictionaries defined earlier. This example avoids switching fonts when it is unnecessary to do so; taking care in this respect is important for efficient execution.

Many applications must switch frequently among arbitrarily named fonts, where the names and sizes are not known in advance. To facilitate this, the operator **selectfont** *(LanguageLevel 2)* combines the actions of the **findfont**, **scalefont** (or

makefont), and **setfont** operators. **selectfont** saves information from one call to the next to avoid calling **findfont** or performing the **scalefont** or **makefont** computations unnecessarily. In the common case of selecting a font and size combination that has been used recently, **selectfont** works with great efficiency.

The **rootfont** operator returns the font parameter in the graphics state, which is the value most recently specified by **setfont** or **selectfont**. The **currentfont** operator returns the same value, except in certain cases where it may return a descendant of a composite font; see Section 5.10, "Composite Fonts." The *current font* is the value returned by **currentfont** and is the one implicitly referenced by **show**.

5.1.3 Achieving Special Graphical Effects

Normal uses of **show** and other glyph-painting operators cause black-filled glyphs to be painted. Other effects can be obtained by combining font operators with general graphics operators.

The color used for painting glyphs is the current color in the graphics state. The default color is black, but other colors can be obtained by executing **setgray** or some other color-setting operator before painting glyphs. Example 5.3 paints glyphs in 50 percent gray, as shown in Figure 5.2.

Example 5.3

```
/Helvetica-Bold findfont  48 scalefont  setfont
20  40  moveto
.5  setgray
(ABC) show
```



FIGURE 5.2 *Glyphs painted in 50% gray*

More general graphical manipulations can be performed by treating the glyph outline as a path instead of immediately painting it. **charpath** is a path construction operator that appends the outlines of one or more glyphs to the current path in the graphics state. This is useful mainly with glyphs that are defined as outlines (as are most standard fonts). Paths derived from glyphs defined as strokes can be used in limited ways. It is not possible to obtain paths for glyphs defined as images or bitmaps; **charpath** produces an empty path. Also, a path consisting of the outlines of more than a few glyphs is likely to exceed the limit on number of path elements (see Appendix B). If possible, it is best to deal with only one glyph's path at a time.

Example 5.4 treats glyph outlines as a path to be stroked. This program uses **charpath** to obtain the outlines for the string of characters ABC in the current font. (The *false* operand to **charpath** is explained in the description of **charpath** in Chapter 8.) The program then strokes this path with a line 2 points thick, rendering the glyph outlines on the page (see Figure 5.3).

Example 5.4

```
/Helvetica findfont  48 scalefont  setfont
20  38  moveto
(ABC) false charpath
2 setlinewidth stroke
```

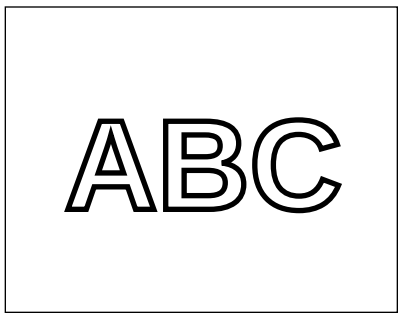


FIGURE 5.3 *Glyph outlines treated as a path*

Example 5.5 obtains the glyphs' path as before, then establishes it as the current clipping path. All subsequent painting operations will mark the page only within this path, as illustrated in Figure 5.4. This state persists until some other clipping path is established—for example, by the **grestore** operator.

Example 5.5

```
/Helvetica findfont  48 scalefont  setfont
newpath
    20  40  moveto
    (ABC) true charpath
clip
… Graphics operators to draw a starburst …
```

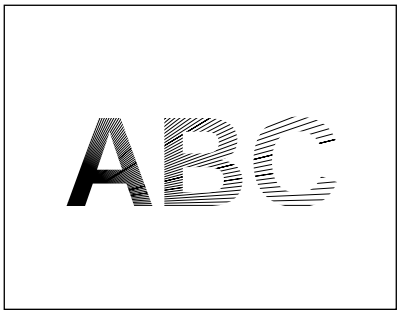


FIGURE 5.4 *Graphics clipped by a glyph path*

5.1.4 Glyph Positioning

A glyph's *width* is the amount of space the glyph occupies along the baseline of a line of text. In other words, it is the distance the current point moves when the glyph is shown. Note that the width is distinct from the dimensions of the glyph outline (see Section 5.4, "Glyph Metric Information").

In some fonts, the width is constant; it does not vary from glyph to glyph. Such fonts are called *fixed-pitch* or *monospaced*. They are used mainly for typewriter-style printing. However, most fonts used for high-quality typography associate a different width with each glyph. Such fonts are called *proportional* fonts or *variable-pitch* fonts. In either case, the **show** operator positions the glyphs for consecutive characters of a string according to their widths.

The width information for each glyph is stored in the font dictionary. A PostScript program can use any of several glyph-painting operators—**show**, **xshow**, **yshow**, **xyshow**, **widthshow**, **ashow**, **awidthshow**—to obtain a variety of width modification effects. If necessary, it can execute **stringwidth** to obtain the width information itself.

The standard operators for showing text (**show** and its variants) are designed on the assumption that glyphs are ordinarily shown with their standard metrics. (See Section 5.4, "Glyph Metric Information".) However, means are provided to vary the metrics in certain limited ways. For example, the **ashow** operator systematically adjusts the widths of all the glyphs it paints. The optional **Metrics** entry of a font dictionary can be added to adjust the widths of all instances of particular glyphs of a font.

Certain applications that show text require very precise control of the positioning of each glyph. There are three LanguageLevel 2 operators to streamline the operation of showing individually positioned glyphs: **xyshow**, **xshow**, and **yshow**. Each operator is given a string of text to be shown, the same as **show**. In addition, it expects a second operand, which is either an array of numbers or a string that can be interpreted as an encoded number string, as described in Section 3.14.5, "Encoded Number Strings." The numbers are used in sequence to control the widths of the glyphs being shown. *They completely override the standard widths of the glyphs.*

The **kshow** and **cshow** operators provide ways for an arbitrary PostScript procedure to intervene in the positioning and painting of each glyph they show. **cshow** is a LanguageLevel 2 operator. These are the most general but least efficient glyph-painting operations.

5.2 Font Dictionaries

Font dictionaries are ordinary dictionary objects, but with certain special entries. The PostScript language has several operators that deal with font dictionaries (see Chapter 8). Some of the contents of a font dictionary are optional and user-definable, while other entries *must* be present and have the correct semantics in order for the PostScript interpreter's font machinery to operate properly.

In addition to fonts, LanguageLevel 3 supports two classes of font-related objects, called *CIDFonts* and *CMaps*, described in Section 5.11, "CID-Keyed Fonts."

There are several kinds of fonts, distinguished by the **FontType** entry in the font dictionary. Each type of font has its own conventions for organizing and representing the information within it. Fonts of type 1, 2, 3, 14, and 42 are called *base fonts*. The standard font types defined at present are listed in Table 5.1.

TABLE 5.1 Font types

TYPE	DESCRIPTION
Type 0	*(LanguageLevel 2)* A *composite* font—a font composed of other fonts, organized hierarchically. See Section 5.10, "Composite Fonts."
Type 1	A font that defines glyph shapes by using a special encoded format. Details on this format are provided in a separate book, *Adobe Type 1 Font Format*.
	The *multiple-master* font format is an extension of the Type 1 font format that allows the generation of a wide variety of typeface styles from a single font. For details of the construction and uses of multiple-master fonts, see Adobe Technical Note #5015, *Type 1 Font Format Supplement*.
Type 2	*(LanguageLevel 3)* A Compact Font Format (CFF) font. See Section 5.8.1, "Type 2 and Type 14 Fonts (CFF and Chameleon)."
Type 3	A font that defines glyphs with ordinary PostScript procedures, which are the values of entries named **BuildGlyph** or **BuildChar** in the font dictionary. See Section 5.7, "Type 3 Fonts."
Types 9, 10, 11, 32	*(LanguageLevel 3)* These **FontType** values identify a class of fontlike objects, called *CIDFonts*, that can be used as descendants in CID-keyed fonts. See Section 5.11, "CID-Keyed Fonts." CIDFonts have their own type numbering, specified by a separate **CIDFontType** entry; a Type 0 font and a Type 0 CIDFont are entirely different kinds of objects. CIDFonts are not considered to be base fonts.
Type 14	*(LanguageLevel 3)* A Chameleon font. See Section 5.8.1, "Type 2 and Type 14 Fonts (CFF and Chameleon)."
Type 42	*(LanguageLevel 3)* A font based on the TrueType font format. See Section 5.8.2, "Type 42 Fonts (TrueType)."

Some products support proprietary font types in addition to the standard ones built into the PostScript language. These product-specific font types are not described in this book, but rather in the manufacturer's documentation for individual products.

As stated earlier, most fonts (and related objects, such as CIDFonts and CMaps) originate as programs contained in files that conform to an external specification, such as *Adobe Type 1 Font Format*. There are two main ways in which such a font program can be introduced into a PostScript interpreter, causing a font dictionary to be created in virtual memory:

- Execution of the **findfont** operator causes the interpreter to locate a font file in external storage and to load its definition into VM on demand.
- The PostScript program defines the font explicitly, by including a copy of the font file embedded directly within the job. (To facilitate document management, a PostScript program containing an embedded font program should use appropriate document structuring comments; see Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*.)

In either case, the PostScript interpreter executes the font program. For some font types, a font program consists entirely of ordinary operators, such as **dict**, **begin**, **end**, and **def**, that construct the font dictionary directly. For other font types, a font program invokes a special-purpose operator, such as **StartData**, that interprets binary-encoded font data appearing in-line and constructs a font dictionary using material extracted from that data.

The font program makes the new font dictionary known to the interpreter by executing the **definefont** operator. **definefont** takes a name and a dictionary, checks that the dictionary is a well-formed font dictionary, makes the dictionary's access read-only, and associates the font name with the dictionary in the font directory. It performs additional modifications that are documented in the **definefont** operator description in Chapter 8. Successful execution of **definefont** makes the font dictionary *valid* for use by other font operators (an **invalidfont** error will occur otherwise).

The operator **undefinefont** *(LanguageLevel 2)* removes a named font from the font directory. A font dictionary that has been removed in this fashion is still a valid font (assuming it is still accessible), but it can no longer be returned by **findfont**.

In LanguageLevels 2 and 3, fonts are actually a special case of *named resources*: a font is an instance of the **Font** resource category. The **findfont**, **definefont**, and **undefinefont** operators are merely conveniences for invoking **findresource**, **defineresource**, and **undefineresource**. Other classes of font-related objects, CIDFonts and CMaps, are ordinarily defined as instances of separate resource

categories, not the **Font** category (however, a CIDFont can be an instance of the **Font** category). Like fonts, these objects become *valid* as a side effect of successful execution of **defineresource**.

A font dictionary can reside in either local or global VM. See Section 3.9, "Named Resources," and the description of the **defineresource** operator for complete information on how resource instances are named and are loaded into VM.

5.2.1 Entries in Font Dictionaries

Table 5.2 lists the entries that have defined meanings in the font dictionaries of *all* types of fonts. Table 5.3 lists additional entries that are meaningful in all *base fonts* (fonts of type 1, 2, 3, 14, or 42). On top of that, Table 5.4 lists additional entries that are meaningful in Type 1 fonts; most of them apply to other base font formats as well. Entries specific to other types of fonts (and to CIDFonts and CMaps) are described later in the sections that discuss those types. Any font dictionary can have additional entries containing information useful to PostScript procedures that are part of the font's definition; the interpreter pays no attention to such entries.

TABLE 5.2 Entries common to all font dictionaries

KEY	TYPE	VALUE
FontType	integer	*(Required)* The font type; see Table 5.1. Indicates where the glyph descriptions are to be found and how they are represented.
FontMatrix	array	*(Required)* An array that transforms the *glyph coordinate system* into the user coordinate system (see Section 5.4, "Glyph Metric Information"). For example, Type 1 font programs from Adobe are usually defined in terms of a 1000-unit glyph coordinate system, and their initial font matrix is [0.001 0 0 0.001 0 0]. When a font is derived by the **scalefont** or **makefont** operator, the new matrix is concatenated with the existing font matrix to yield a new copy of the font with a different font matrix.
FontName	name	*(Optional)* The name of the font. This entry is for information only; it is not used by the PostScript interpreter. Ordinarily, it is the same as the key passed to **definefont**, but it need not be.
FontInfo	dictionary	*(Optional)* A dictionary containing font information that is not accessed by the PostScript interpreter; see Table 5.5 on page 327.

LanguageLevel	integer	*(Optional)* The minimum LanguageLevel required for correct behavior of the font. For example, any font that uses LanguageLevel 2 features for rendering glyphs (such as a glyph description that uses **rectfill** or **glyphshow**) should specify a **LanguageLevel** value of 2. On the other hand, the presence of higher-LanguageLevel information that an interpreter can safely ignore does not require **LanguageLevel** to be set to the higher LanguageLevel. For example, an **XUID** entry in the font dictionary—LanguageLevel 2 information that a LanguageLevel 1 interpreter can ignore—does *not* require setting **LanguageLevel** to 2. Default value: 1.
WMode	integer	*(Optional; LanguageLevel 2)* The writing mode, which determines which of two sets of metrics will be used when glyphs are shown from the font. Mode 0 specifies horizontal writing; mode 1 specifies vertical writing (see Section 5.4, "Glyph Metric Information"). LanguageLevel 1 implementations lacking composite font extensions ignore this entry. Default value: 0.
FID	fontID	*(Inserted by* **definefont***)* A special object of type fontID that serves internal purposes in the font machinery. The **definefont** operator inserts this entry. In LanguageLevel 1, an **FID** entry must not previously exist in the dictionary presented to **definefont**; the dictionary must have sufficient space to insert this entry.

TABLE 5.3 Additional entries common to all base fonts

KEY	TYPE	VALUE
Encoding	array	*(Required)* An array of character names to which character codes are mapped. See Section 5.3, "Character Encoding."
FontBBox	array	*(Required)* An array of four numbers in the glyph coordinate system giving the left, bottom, right, and top coordinates, respectively, of the *font bounding box*. The font bounding box is the smallest rectangle enclosing the shape that would result if all of the glyphs of the font were placed with their origins coincident, and then painted. This information is used in making decisions about glyph caching and clipping. If all four values are 0, the PostScript interpreter makes no assumptions based on the font bounding box. If any value is nonzero, it is essential that the font bounding box be accurate; if any glyph's marks fall outside this bounding box, incorrect behavior may result. In many Type 1 fonts, the **FontBBox** array is executable, though there is no good reason for this to be so. Programs that access **FontBBox** should invoke an explicit **get** or **load** operator to avoid unintended execution.

UniqueID	integer	*(Optional)* An integer in the range 0 to 16,777,215 ($2^{24} - 1$) that uniquely identifies this font. See Section 5.6, "Unique ID Generation."
XUID	array	*(Optional; LanguageLevel 2)* An array of integers that uniquely identifies this font or any variant of it. See Section 5.6, "Unique ID Generation."

TABLE 5.4 Additional entries specific to Type 1 fonts

KEY	TYPE	VALUE
PaintType	integer	*(Required)* A code indicating how the glyphs of the font are to be painted: 0 Glyph outlines are filled. 2 Glyph outlines (designed to be filled) are stroked. To get a stroked-outline font, a program can copy the font dictionary, change the **PaintType** from 0 to 2, add a **StrokeWidth** entry, and define a new font using this dictionary. Note that the previously documented **PaintType** values of 1 and 3 are not supported.
StrokeWidth	number	*(Optional)* The stroke width (in units of the glyph coordinate system) for stroked-outline fonts (**PaintType** 2). This entry is not initially present in filled-outline fonts. It should be added (with a nonzero value) when a stroked font is created from an existing filled font. Default value: 0.
Metrics	dictionary	*(Optional)* A dictionary containing metric information (glyph widths and sidebearings) for writing mode 0. This entry is not normally present in the original definition of a font. Adding a **Metrics** entry to a font overrides the metric information encoded in the glyph descriptions. See Sections 5.4, "Glyph Metric Information," and 5.9.2, "Changing Glyph Metrics."
Metrics2	dictionary	*(Optional; LanguageLevel 2)* Similar to **Metrics**, but for writing mode 1.
CDevProc	procedure	*(Optional; LanguageLevel 2)* A procedure that algorithmically derives global changes to the font's metrics. LanguageLevel 1 implementations lacking composite font extensions ignore this entry.
CharStrings	dictionary	*(Required)* A dictionary associating character names (the keys in the dictionary) with glyph descriptions. Each entry's value is ordinarily a string (called a *charstring*) that represents the glyph description for that character in a special encoded format; see *Adobe Type 1 Font Format* for details. The value can also be a PostScript procedure; see Section 5.9.3, "Replacing or Adding Individual Glyphs." This dictionary must have an entry whose key is .notdef.
Private	dictionary	*(Required)* A dictionary containing other internal information about the font. See *Adobe Type 1 Font Format* for details.

WeightVector	array	*(Required; multiple-master fonts only)* An array specifying the contribution of each master design to the current font instance. The array contains one number per master design, each typically in the range 0.0 to 1.0; the sum of the array elements must be 1.0. The values in the array are used for calculating the weighted interpolation. Elements outside the allowed range may produce unexpected results.

Any font dictionary can contain a **FontInfo** entry whose value is a dictionary containing the information listed in Table 5.5. This information is *entirely* for the benefit of PostScript programs using the font, or for documentation; it is not accessed by the PostScript interpreter, and all of its entries are optional.

TABLE 5.5 Entries in a FontInfo dictionary

KEY	TYPE	VALUE
FamilyName	string	A human-readable name for a group of fonts that are stylistic variants of a single design. All fonts that are members of such a group should have exactly the same **FamilyName** value.
FullName	string	A unique, human-readable name for an individual font.
Notice	string	A trademark or copyright notice, if applicable.
Weight	string	A human-readable name for the weight, or "boldness," attribute of a font.
version	string	The version number of the font program.
ItalicAngle	number	The angle, in degrees counterclockwise from the vertical, of the dominant vertical strokes of the font.
isFixedPitch	boolean	A flag indicating whether the font is a fixed-pitch (monospaced) font.
UnderlinePosition	number	The recommended distance from the baseline for positioning underlining strokes. This number is the *y* coordinate (in the glyph coordinate system) of the center of the stroke.
UnderlineThickness	number	The recommended stroke width for underlining, in units of the glyph coordinate system.

The PostScript language does not specify any formal rules for the names of fonts or for the entries in the **FontInfo** dictionary. However, there are various conventions for organizing fonts that facilitate their use by application programs.

- Some applications use **FamilyName** as part of a hierarchical font-selection user interface. This divides a very large set of individual fonts into a smaller, more manageable set of "font families." The **FamilyName** parameter should be suitable for use in a font selection menu.
- Typically, **FullName** begins with **FamilyName** and continues with various style descriptors separated by spaces—for example, Adobe Garamond Bold Italic. In some designs, a numbering system replaces or augments verbal descriptors—for example, Univers 55 Medium.
- **Weight** is derived from the **FullName** parameter by dropping everything from **FullName** that does not explicitly relate to weight. For example, the **FullName** entry ITC Franklin Gothic Condensed Extra Bold Oblique reduces to a **Weight** entry of Extra Bold.
- The font dictionary's **FontName** parameter, which is also usually used as the key passed to **definefont**, is a condensation of **FullName**. It is customary to remove spaces and to limit its length to fewer than 40 characters. The resulting name should be unique.

5.3 Character Encoding

Font definitions use a flexible *encoding* scheme by which character codes select glyph descriptions. The association between character codes and glyph descriptions is not part of the glyph descriptions themselves, but instead is described by a separate *encoding vector*. A PostScript program can change a font's encoding vector to match the requirements of the application generating the description.

This section describes the character encoding scheme used with base fonts. Composite fonts (Type 0) use a more complicated character mapping algorithm, as discussed in Section 5.10, "Composite Fonts."

Note: *Every base font must have an* ***Encoding*** *entry, which the PostScript font machinery accesses automatically as described below. A Type 3 font's* ***BuildChar*** *procedure should use this entry in the standard way; see Section 5.7, "Type 3 Fonts."*

In a font dictionary, the descriptions of the individual glyphs are keyed by character *names*, not by character *codes*. Character names are ordinary PostScript name objects. Descriptions of Latin alphabetic characters are normally associated with names consisting of single letters, such as A or a. Other characters are associated with names composed of words, such as three, ampersand, or parenleft.

The encoding vector is defined by the array object that is the value of the **Encoding** entry in the font dictionary. The array is indexed by character code (an integer in the range 0 to 255). The elements of the array must be character names, and the array should be 256 elements long. (A base font used as a descendant in a composite font may be accessed using a code that is not 8 bits long, in which case the length of the **Encoding** array should be correspondingly different.)

The operand to the **show** operator is a PostScript string object. Each element of the string is treated as a character code. When **show** paints a character:

1. It uses the character code as an index into the current font's **Encoding** array to obtain a character name.
2. It invokes the glyph description by name. For a Type 1 font, it looks up the name in the font's **CharStrings** dictionary to obtain an encoded glyph description (called a *charstring*), which it executes; Figure 5.5 illustrates this common case. For a Type 3 font, it calls the font's **BuildGlyph** procedure (if present) with the name as its operand; see Section 5.7, "Type 3 Fonts." Other types of fonts have their own conventions for associating character names with glyph descriptions.

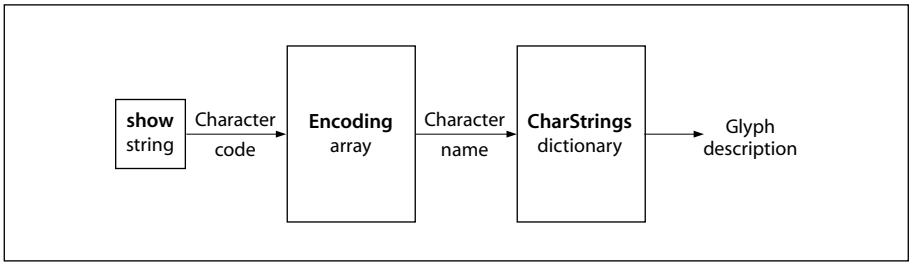


FIGURE 5.5 *Encoding scheme for Type 1 fonts*

For example, in the standard encoding vector used by Type 1 Latin-text fonts such as Helvetica, the element at index 38 is the name object ampersand. When **show** encounters the value 38 (the ASCII character code for &) as an element of a string it is printing, it fetches the encoding vector entry at index 38, obtaining the name object ampersand. It then uses ampersand as a key in the current font dictionary's **CharStrings** subdictionary and executes the associated charstring that renders the & glyph.

Changing an existing font's encoding involves creating a new font dictionary that is a copy of the existing one except for its **Encoding** entry. The subsidiary dictionaries, such as **CharStrings** and **FontInfo**, continue to be shared with the original font. Of course, a new font may be created with any desired encoding vector. This flexibility in character encoding is valuable for two reasons:

- It permits showing text that is encoded according to any of the various existing conventions. For example, the Microsoft Windows and Apple Mac OS operating systems use different standard encodings for Latin text, and many applications use their own special-purpose encodings.
- It allows applications to specify how characters selected from a large character set are to be encoded. Some character sets consist of more than 256 characters, including ligatures, accented characters, and other symbols required for high-quality typography or non-Latin writing systems. Different encodings can select different subsets of the same character set.

Latin-text font programs produced by Adobe Systems use the "Adobe standard" encoding vector, which is associated with the name **StandardEncoding** in **systemdict**. An alternate encoding vector called ISO Latin-1 is associated with the name **ISOLatin1Encoding**. Complete details of these encodings and of the characters present in typical fonts are provided in Appendix E.

All unused positions in an encoding vector must be filled with the character name .notdef. Like any other character name, the name .notdef is defined in the **CharStrings** dictionary. If an encoding maps to a character name that does not exist in the font, the name .notdef is substituted. Every font must contain a glyph description for the .notdef character. The effect produced by showing the .notdef character is at the discretion of the font designer. In Type 1 font programs produced by Adobe, it is the same as the space character.

The **glyphshow** operator *(LanguageLevel 2)* shows the glyph for a single character specified by name instead of by character code. This allows direct access to any character in the font regardless of the font's **Encoding** array. The principal use of **glyphshow** is in defining fonts whose glyph descriptions refer to other characters in the same or a different font. Referring to such characters by name ensures proper behavior if the font is subsequently reencoded.

5.4 Glyph Metric Information

The *glyph coordinate system* is the space in which an individual character's glyph is defined. All path coordinates and metrics are interpreted in glyph space. Figure 5.6 shows a typical glyph outline and its metrics.

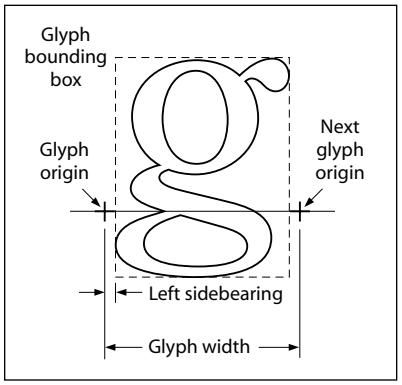


FIGURE 5.6 *Glyph metrics*

The *glyph origin*, or *reference point*, is the point (0, 0) in the glyph coordinate system. **show** and other glyph-painting operators position the origin of the first glyph to be shown at the current point in user space. For example,

```
40 50 moveto
(ABC) show
```

places the origin of the A at coordinate (40, 50) in the user coordinate system.

The *glyph width* is the distance from the glyph's origin to the point at which the origin of the *next* glyph should normally be placed when painting the consecutive glyphs of a word. This distance is a vector in the glyph coordinate system; it has *x* and *y* components. Most Indo-European alphabets, including the Latin alphabet, have a positive *x* width and a zero *y* width. Semitic alphabets have a negative *x* width, and some Asian writing systems have a nonzero *y* width.

The *glyph bounding box* is the smallest rectangle (oriented with the axes of the glyph coordinate system) that will just enclose the entire glyph shape. The

bounding box is expressed in terms of its left, bottom, right, and top coordinates relative to the glyph origin in the glyph coordinate system.

The *left sidebearing* of a glyph is the position of the *left sidebearing point* in glyph space. This is usually the intersection of the left edge of the bounding box with the glyph's baseline; however, the exact interpretation of the left sidebearing depends on details of the font technology (for Type 1, see *Adobe Type 1 Font Format*). Note that the *x* coordinate of the left sidebearing can be negative for glyphs that extend to the left of their origin. The *y* coordinate is almost always 0.

Type 1 fonts are defined in such a way that a glyph's left sidebearing and width can be adjusted; that is, the glyph bounding box and the position of the next glyph can be shifted relative to the origin (see Section 5.9, "Font Derivation and Modification"). Some other types of base fonts work similarly.

In some writing systems, text is frequently aligned in two different directions. For example, it is common to write Japanese and Chinese glyphs either horizontally or vertically. To handle this, a font can optionally contain a second set of metrics for each glyph. This feature is available only in LanguageLevel 2 or 3 or in LanguageLevel 1 implementations with composite font extensions.

The metrics are accessed by **show** and other operators according to a writing mode, given by a **WMode** entry in the font dictionary (or in some parent font dictionary in a composite font). By convention, writing mode 0 (the default) specifies horizontal writing and mode 1 specifies vertical writing. If a font contains only one set of metrics, the **WMode** parameter is ignored.

When a glyph has two sets of metrics, each set specifies a glyph origin and a width vector. Figure 5.7 illustrates the relationship between the two sets of metrics. The left diagram illustrates the glyph metrics associated with writing mode 0. The coordinates *ll* and *ur* specify the bounding box of the glyph relative to origin 0. *w0* is the glyph width vector that specifies how the current point is changed after the glyph is shown in writing mode 0. The center diagram illustrates writing mode 1; *w1* is the glyph width vector for writing mode 1. In the right diagram, *v* is a vector defining the position of origin 1 relative to origin 0.

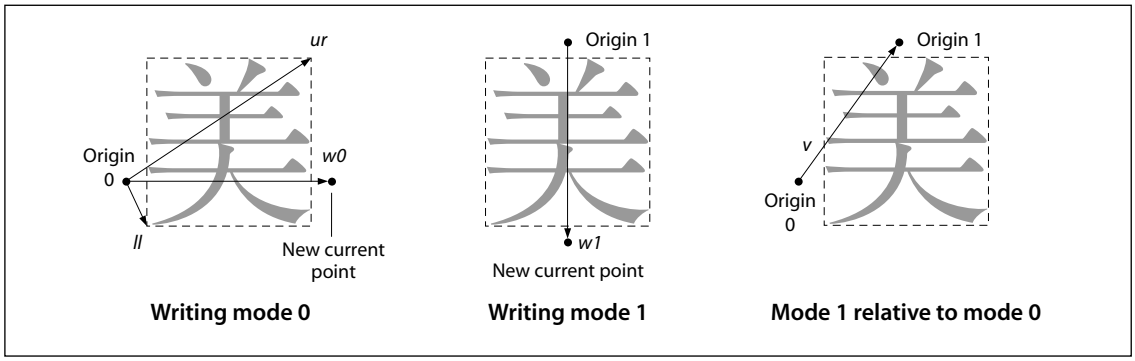


FIGURE 5.7 *Relationship between two sets of metrics*

Glyph metric information can be accessed procedurally by a PostScript program. The **stringwidth** operator obtains glyph widths. The sequence

```
charpath flattenpath pathbbox
```

computes glyph bounding boxes, though this is relatively inefficient. The bounding box for an entire font appears in the font dictionary as an array of four numbers associated with the key **FontBBox**.

Glyph metric information is also available separately in the form of Adobe font metrics (AFM) and Adobe composite font metrics (ACFM) files. These files are for use by application programs that generate PostScript page descriptions and must make formatting decisions based on the widths and other metrics of glyphs. Kerning information is also available in the AFM and ACFM files. When possible, applications should use this information directly instead of generating PostScript instructions to compute it.

Specifications for the AFM and ACFM file formats are available in Adobe Technical Note #5004, *Adobe Font Metrics File Format Specification.*

5.5 Font Cache

The PostScript interpreter includes an internal data structure called the *font cache* whose purpose is to make the process of painting glyphs very efficient. For the most part, font cache operation is automatic. However, fonts whose glyph descriptions are PostScript procedures, such as Type 3 fonts, must adhere to certain

conventions to take advantage of the font cache. Also, there are several operators that control the behavior of the font cache, including ones that directly update the font cache on behalf of a Type 4 CIDFont.

Rendering a glyph from an outline or other high-level description is a relatively costly operation, because it involves performing scan conversion of arbitrary shapes. This presents special problems for printing text, because it is common for several thousand glyphs to appear on a single page. However, a page description that includes large amounts of text normally has many repetitions of the same glyph in a given font, size, and orientation. The number of distinct glyphs thus is very much smaller than the total number of glyphs.

The font cache operates by saving the results of glyph scan conversions (including metric information and device pixel arrays) in temporary storage and using those saved results when the same glyph is requested again. The font cache is usually large enough to accommodate all of the distinct glyphs in a page description. Painting a glyph that is already in the font cache is typically hundreds of times faster than scan-converting it from the glyph description in the font.

The font cache does not retain color information; it remembers only which pixels were painted and which pixels were left unchanged within the glyph's bounding box. For this reason, there are a few restrictions on the set of graphical operators that may be executed as part of glyph descriptions that are to be cached. In particular, the **image** operator is not permitted. However, **imagemask** may be used to define a glyph according to a bitmap representation; see Section 4.10, "Images." Execution of operators that specify colors or other color-related parameters in the graphics state is also not permitted; see Section 4.8, "Color Spaces."

The principal manifestation of the font cache visible to the PostScript program is that showing a glyph does not necessarily result in the execution of the glyph's description. This means that glyph descriptions that are PostScript procedures must interact with the font cache machinery to ensure that the results of their execution are properly saved; this interaction is accomplished by means of the **setcachedevice** or **setcachedevice2** operator, as described in Section 5.7, "Type 3 Fonts."

Each glyph saved in the font cache is identified by the combination of:

- The original base font or CIDFont dictionary from which the glyph description was obtained.
- The *character selector*, which is a character name in a base font or a CID (character identifier) in a CIDFont; see Section 5.11, "CID-Keyed Fonts."
- The current transformation matrix (CTM) at the time the glyph is shown.

To ensure predictable behavior despite font caching, a given combination of font and character selector must always produce the same appearance when rendered. Means exist to provide reliable identification of a font definition; see the next section. Note that a character code is *not* part of the glyph identification in the font cache, since a given glyph may be selected in more than one way.

5.6 Unique ID Generation

A *unique ID* is an optional entry in a font dictionary that helps identify the font to the interpreter. Its primary purpose is to identify cached glyphs built from that font. The PostScript interpreter can retain glyphs in the font cache even for a font that is not permanently in virtual memory. Some implementations can save cached glyphs on disk. This can have a beneficial effect on performance when using fonts that are loaded into VM dynamically, either by explicit downloading or automatically via the resource facility.

If a font has a unique ID, the interpreter can recognize that the cached glyphs belong to that font, even if the font dictionary itself is removed from VM and is later reloaded (by a subsequent job, for instance). If a font does not have a unique ID, the interpreter can recognize cached glyphs for that font only while it remains in VM. When the font is removed, the cached glyphs must be discarded.

Correct management of unique IDs is essential to ensure predictable behavior. If two fonts have the same unique ID but produce glyphs with different appearances when executed, it is unpredictable which glyphs will appear when those fonts are used. Therefore, unique IDs must be assigned systematically from some central registry.

The reason that font caching is based on a special unique ID entry rather than on the font's name or other identifying information is that font names are not necessarily unique. A font with a particular name, such as Garamond-Bold, may be

available from several sources, and there may be successive releases of a font from the same source.

For information about assigning unique IDs, consult the Adobe Developer Relations Web site (see the Bibliography) or send e-mail to the Adobe Unique ID Coordinator at fontdev-person@adobe.com.

As described below, there are two kinds of unique ID entry that can appear in font dictionaries: **UniqueID** and **XUID**. **UniqueID** is supported by all LanguageLevels and applies only to base fonts. **XUID** is a LanguageLevel 2 feature that applies not only to base fonts but also to certain other categories of resources—see Sections 4.7, "Forms"; 4.9, "Patterns"; 5.11.3, "CIDFont Dictionaries"; and 5.11.4, "CMap Dictionaries."

When you create a new font program that will be saved permanently and perhaps distributed widely, you should assign **UniqueID** and **XUID** values for that font and embed those values in the definition of the font dictionary. On the other hand, when an application program constructs a font as part of building a page description, it should not include a **UniqueID** or **XUID** entry in the font dictionary, because there is no opportunity for registering the ID and there is little to be gained from doing so in any event.

When you copy a font dictionary for the purpose of creating a modified font, you should *not* copy the **UniqueID** or **XUID** entry. As an exception to this general rule, it is acceptable (and preferable) to retain the original **UniqueID** or **XUID** entry if the only modified entries are **FontName**, **FontInfo**, **FontMatrix**, or **Encoding**, because those changes do not affect the glyphs' appearance or metrics.

5.6.1 Unique ID Numbers

The **UniqueID** entry in a font dictionary is an integer in the range 0 to 16,777,215 ($2^{24} - 1$). Each font type has its own space of **UniqueID** values. Therefore, a Type 1 font and a Type 3 font could have the same **UniqueID** number and be safely used together without causing conflicts in the font cache.

The **UniqueID** numbers for Type 1 fonts are controlled: Adobe Systems maintains a registry of these numbers. Numbers between 4,000,000 and 4,999,999 are reserved for private interchange in closed environments and cannot be registered.

5.6.2 Extended Unique ID Numbers

The **XUID** entry in a font dictionary is an *extended unique ID*—an array of integers that provides for distributed, hierarchical management of the space of unique ID numbers. A font is uniquely identified by the *entire* sequence of numbers in the array. **XUID** is a LanguageLevel 2 feature; it is ignored by LanguageLevel 1 implementations.

The first element of an **XUID** array is a *unique organization ID*, assigned to an organization by the Adobe registry. The remaining elements—and the allowed length of **XUID** arrays—are controlled by the organization. An organization can establish its own registry for managing the space of numbers in these remaining elements, which are interpreted relative to the organization ID.

The organization ID 1,000,000 is reserved for private interchange in closed environments. **XUID** arrays starting with this number may be of any length, subject to an implementation limit (see Appendix B).

This scheme also makes it possible to derive unique IDs systematically when creating modifications of existing fonts. This is not possible for **UniqueID** values because the space of numbers is too small. A program can replace an **XUID** array with a longer **XUID** array consisting of:

- The organization ID of the originator of the program
- The **XUID** array elements from the original font
- Additional elements indicating exactly what modifications have been performed

PostScript interpreters that recognize the **XUID** array ignore **UniqueID** whenever an **XUID** entry is present. For backward compatibility with the installed base of interpreters, font creator and font modifier software should continue to use and maintain appropriate **UniqueID** numbers for the foreseeable future.

5.7 Type 3 Fonts

A Type 3 font is one whose behavior is determined entirely by PostScript procedures. In contrast, most other types of base fonts originate as font programs that conform to an external specification, such as *Adobe Type 1 Font Format*, that has

no direct connection with the PostScript language. This section describes how to build a Type 3 font from scratch.

In addition to the entries common to all base fonts (Table 5.2 on page 324 and Table 5.3 on page 325), a Type 3 font dictionary includes the entries listed in Table 5.6.

TABLE 5.6 Additional entries specific to Type 3 fonts

KEY	TYPE	VALUE
BuildGlyph	procedure	*(Optional; LanguageLevel 2)* A procedure that constructs the requested glyph. The font dictionary, followed by the character name, is on the stack when the procedure is called. See Section 5.7.1, "BuildGlyph."
BuildChar	procedure	*(Required for LanguageLevel 1 or if **BuildGlyph** is absent)* A procedure that constructs the requested glyph (only if no **BuildGlyph** entry is present, in LanguageLevel 2 or 3). The font dictionary, followed by the character code, is on the stack when the procedure is called. See Section 5.7.2, "BuildChar."

5.7.1 BuildGlyph

When a PostScript program tries to show a glyph from a Type 3 font, and the glyph is not already present in the font cache, the PostScript interpreter:

1. Uses the character code as an index into the current font's **Encoding** array, obtaining the corresponding character name. (This step is omitted during a **glyphshow** operation.)
2. Pushes the current font dictionary and the character name on the operand stack.
3. Executes the font's **BuildGlyph** procedure. **BuildGlyph** must remove these two objects from the operand stack and use this information to construct the requested glyph. This typically involves determining the glyph description needed, supplying glyph metric information, constructing the glyph, and painting it.

All Type 3 fonts must include a character named .notdef. The **BuildGlyph** procedure should be able to accept that character name regardless of whether such a character is encoded in the **Encoding** array. If the **BuildGlyph** procedure is given a

character name it does not recognize, it can handle that condition by painting the glyph for the .notdef character instead.

BuildGlyph is called within the confines of a **gsave** and a **grestore**, so any changes it makes to the graphics state do not persist after it finishes. Each call to **BuildGlyph** is independent of any other call. Because of the effects of font caching, no assumptions can be made about the order in which glyph descriptions will be executed. In particular, **BuildGlyph** should not depend on any nonconstant information in VM, and it should not leave any side effects in VM or on stacks.

When **BuildGlyph** gets control, the current transformation matrix (CTM) is the concatenation of the font matrix (**FontMatrix** in the current font dictionary) and the CTM that was in effect at the time **show** was invoked. This means that shapes described in the glyph coordinate system will be transformed into the user coordinate system and will appear in the appropriate size and orientation on the page. **BuildGlyph** should describe the glyph in terms of absolute coordinates in the glyph coordinate system, placing the glyph origin at (0, 0) in this space. It should make no assumptions about the initial value of the current point parameter.

The results of **BuildGlyph** should depend only on the complete transformation from glyph space to device space, and not on the relative contributions of the font matrix and the CTM prior to **BuildGlyph**. In particular, **BuildGlyph** should not attempt to vary its results depending on the font dictionary's **FontMatrix** entry.

Aside from the CTM, the graphics state is inherited from the environment of the **show** operator (or **show** variant) that caused **BuildGlyph** to be invoked. To ensure predictable results despite font caching, **BuildGlyph** must initialize any graphics state parameters on which it depends. In particular, if it invokes the **stroke** operator, **BuildGlyph** should explicitly set the line width, line join, line cap, and dash pattern to appropriate values. Normally, it is unnecessary and undesirable to initialize the current color parameter, because **show** is defined to paint glyphs with the current color.

BuildGlyph must execute one of the following operators to pass width and bounding box information to the PostScript interpreter. This must precede execution of any path construction or painting operators describing the glyph.

- **setcachedevice** establishes a single set of metrics for both writing modes, and requests that the interpreter save the results in the font cache if possible.
- **setcachedevice2** *(LanguageLevel 2)* establishes separate sets of metrics for writing modes 0 and 1, and requests that the interpreter save the results in the font cache.
- **setcharwidth** passes just the glyph's width (to be used only once), and requests that the glyph *not* be cached. This operator is typically used only if the glyph description includes operators to set the color explicitly.

See the descriptions of **setcachedevice**, **setcachedevice2**, and **setcharwidth** in Chapter 8 for more information.

After executing one of these operators, **BuildGlyph** should execute a sequence of graphics operators to perform path construction and painting. The PostScript interpreter transfers the results into the font cache, if appropriate, and onto the page at the correct position. It also uses the width information to control the spacing between this glyph and the next. The final position of the current point in the glyph coordinate system does not influence glyph spacing.

5.7.2 BuildChar

In LanguageLevel 2 or 3, if there is no **BuildGlyph** procedure for the font, the interpreter calls the **BuildChar** procedure instead. In LanguageLevel 1, **BuildChar** is always called, whether or not a **BuildGlyph** procedure is present.

The semantics of **BuildChar** are essentially the same as for **BuildGlyph**. The only difference is that **BuildChar** is called with the font dictionary and the *character code* on the operand stack, instead of the font dictionary and the *character name*. The **BuildChar** procedure must then perform its own lookup to determine what glyph description corresponds to the given character code.

For backward compatibility with the installed base of LanguageLevel 1 interpreters, all new Type 3 fonts should contain the following **BuildChar** procedure:

```
/BuildChar
    {   1 index /Encoding get exch get
        1 index /BuildGlyph get exec
    } bind def
```

This defines **BuildChar** in terms of the same font's **BuildGlyph** procedure, which contains the actual commands for painting the glyph. This permits the font to be used with higher-LanguageLevel features—such as the LanguageLevel 2 operator **glyphshow**, which requires **BuildGlyph** to be present—yet retains compatibility with LanguageLevel 1.

5.7.3 Example of a Type 3 Font

Example 5.6 shows the definition of a Type 3 font with only two glyphs—a filled square and a filled triangle, selected by the characters a and b. Figure 5.8 shows the output from this example. The glyph coordinate system is on a 1000-unit scale. This is not a realistic example, but it does illustrate all the elements of a Type 3 font, including a **BuildGlyph** procedure, an **Encoding** array, and a subsidiary dictionary for the individual glyph descriptions.

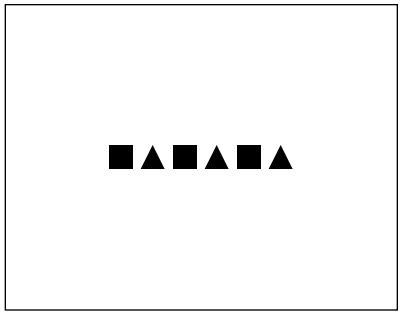


FIGURE 5.8 *Output from Example 5.6*

Example 5.6

```
8 dict begin
    /FontType 3 def                          % Required elements of font
    /FontMatrix [.001  0  0  .001  0  0] def
    /FontBBox [0  0  750  750] def

    /Encoding 256 array def                  % Trivial encoding vector
        0  1  255  {Encoding exch /.notdef put} for
        Encoding 97 /square put              % ASCII a = 97
        Encoding 98 /triangle put            % ASCII b = 98
```

```
/CharProcs 3 dict def                    % Subsidiary dictionary for
    CharProcs begin                      % individual glyph descriptions
        /.notdef { } def
        /square
            {   0 0 moveto
                750 0 lineto
                750 750 lineto
                0 750 lineto
                closepath
                fill
            } bind def
        /triangle
            {   0 0 moveto
                375 750 lineto
                750 0 lineto
                closepath
                fill
            } bind def
    end                                  % End of CharProcs

    /BuildGlyph                          % Stack contains: font charname
        {   1000 0                       % Width
            0 0 750 750                  % Bounding box
            setcachedevice
            exch /CharProcs get exch     % Get CharProcs dictionary
            2 copy known not             % See if charname is known
                {pop /.notdef}
            if
            get exec                     % Execute BuildGlyph procedure
        } bind def

    /BuildChar                           % LanguageLevel 1 compatibility
        {   1 index /Encoding get exch get
            1 index /BuildGlyph get exec
        } bind def

    currentdict
end                                      % End of font dictionary

/ExampleFont exch definefont pop

/ExampleFont findfont 12 scalefont setfont   % Now show some characters
36 52 moveto
(ababab) show
```

5.8 Additional Base Font Types

This section describes additional base font types besides Type 1 and Type 3: Type 2 (CFF), Type 14 (Chameleon), and Type 42 (TrueType). These three additional font formats are LanguageLevel 3 features.

5.8.1 Type 2 and Type 14 Fonts (CFF and Chameleon)

The Type 2 (Compact Font Format, or CFF) and Type 14 (Chameleon) font formats are compact representations that enable multiple fonts to be stored in a unit called a *font set*. A font set is an instance of a resource category named **FontSet**. This category is distinct from the **Font** category; however, the constituent fonts in a font set are also available as instances of the **Font** category and can be accessed by **findfont** or **findresource**.

These formats are not expressed in PostScript language syntax, but as binary-encoded data structures that are decoded by specialized interpreters. The binary font data is enclosed, or "wrapped," in a minimal amount of PostScript syntax, yielding a file that can be treated as an external resource in the **FontSet** category. When a constituent font is accessed by **findfont**, the resulting PostScript font dictionary has a **FontType** value of 2 for CFF or 14 for Chameleon.

CFF is a representation of one or more fonts based on the Type 1 charstring format. It retains full fidelity to the original fonts, while achieving significant space reduction due to a compact binary representation and sharing of data that is common to multiple fonts. For more information on this format, see Adobe Technical Note #5176, *Compact Font Format Specification.*

The Chameleon font format is an implementation of a "shape library" that allows compact representations of Latin-text fonts. This format consists of a master font and its *font descriptor* database: the master font is tailored to address the needs of a particular product, while the font descriptors define how to extract fonts of interest from the master. The details of the Chameleon font format are not documented.

Typically, for a product's built-in fonts, there is one font set for all CFF fonts and one each for a Chameleon master font and its descriptor database. Additional font sets can be installed on disk or embedded in a PostScript page description.

FontSet Resources

The external representation for a **FontSet** resource instance consists of minimal PostScript syntax enclosing the binary-encoded font data. Example 5.7 illustrates the resource file for a CFF font set; the same syntax is used for the Chameleon font format. For details on the document structuring comments used in this example, see Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification.*

Example 5.7

```
%!PS-Adobe-3.0 Resource-FontSet
%%DocumentNeedResources: ProcSet (FontSetInit)
%%Title: (FontSet/CFFRoman27)
%%Version: 1.000
%%EndComments
%%IncludeResource: ProcSet (FontSetInit)
%%BeginResource: FontSet (CFFRoman27)
/FontSetInit /ProcSet findresource begin
%%BeginData: 622532 Binary Bytes
/CFFRoman27 622503 StartData
… 622,503 bytes of binary data …
%%EndData
%%EndResource
%%EOF
```

When the code in Example 5.7 is executed, the **StartData** operator defined in the **FontSetInit** procedure set processes the binary data, builds a **FontSet** resource instance, and invokes the **defineresource** operator, naming the resource instance CFFRoman27. The contents of the **FontSet** resource instance (a dictionary) are undocumented and subject to change.

In this example the **StartData** operator takes two operands; in a **FontSet** resource file containing Chameleon font descriptors, a third operand is present that gives the name of the associated master font. For more information, see the description of **StartData** in Chapter 8.

***Note: StartData** is the only operator in the **FontSetInit** procedure set that is documented as part of the PostScript language. Other entries in this procedure set are private to the implementation of font sets and should not be accessed by a PostScript program.*

Accessing CFF and Chameleon Fonts in a PostScript Program

The fonts contained in all CFF and Chameleon **FontSet** resources also appear as instances of the **Font** resource category, which makes them individually accessible via the **findfont** operator and the LanguageLevel 2 resource operators (such as **findresource** and **resourceforall**). The mechanism by which this is achieved is not specified as part of the PostScript language.

***Note: FontSet** resource instances can have arbitrary keys, which have nothing to do with the names of their constituent fonts. If a given font name appears in multiple **FontSet** instances, it is unpredictable which one will be selected by **findfont**.*

When **findfont** or **findresource** finds a font contained in a **FontSet** resource instance, it constructs a font dictionary containing information extracted from the **FontSet** instance's binary data. The resulting font dictionary has a **FontType** value of 2 (for CFF) or 14 (for Chameleon). It contains the same entries as a Type 1 font dictionary, as defined in Tables 5.2, 5.3, and 5.4 (starting on page 324), except:

- An entry in its **CharStrings** dictionary is ordinarily a *glyph index*—an integer used as an index into the binary data for the font. (The interpretation of a glyph index is internal to the font; it has no external significance and, in particular, has nothing to do with a character code.) A PostScript program can replace this integer with a procedure; see Section 5.9.3, "Replacing or Adding Individual Glyphs."
- It does not contain a **Private** entry.
- It has additional entries that are undocumented and subject to change.

A PostScript program can copy the font dictionary and insert or modify entries as specified in the aforementioned tables. These modifications have the same effects as for Type 1 fonts.

In Adobe PostScript implementations, there exists a fictitious file system named %fontset%, whose "files" are the constituent fonts in all available **FontSet** resource

instances. The purpose of this file system is to provide compatibility with common methods of querying external font instances, used in some existing applications and drivers.

LanguageLevel 1 does not provide any standard means for querying or enumerating external font instances (equivalent to **resourcestatus** or **resourceforall** in LanguageLevel 2). Instead, some applications invoke the **status** and **filenameforall** operators to refer to font files directly. The %fontset% fictitious file system enables such applications to work correctly with fonts defined in **FontSet** resources. This compatibility mechanism is documented in the *PostScript Language Reference Supplement.*

5.8.2 Type 42 Fonts (TrueType)

LanguageLevel 3 includes support for Type 42 fonts. The Type 42 font format enables TrueType fonts to be accessed from within a PostScript program. Like the CFF and Chameleon font formats, the TrueType font format is not expressed in PostScript language syntax, but rather as binary-encoded data structures that are decoded by specialized interpreters. The binary font data defining the TrueType font is wrapped in PostScript language syntax to make it conform to the PostScript language font model: the font data is stored in an entry named **sfnts** in a Type 42 font dictionary.

In addition to the entries common to all base fonts (Table 5.2 on page 324 and Table 5.3 on page 325), a Type 42 font dictionary includes the entries listed in Table 5.7. For further details, see Adobe Technical Note #5012, *The Type 42 Font Format Specification.*

TABLE 5.7 Additional entries specific to Type 42 fonts

KEY	TYPE	VALUE
CharStrings	dictionary	*(Required)* A dictionary associating character names (keys) with glyph descriptions. Each entry's value is ordinarily a glyph index—an integer used as an index into the TrueType "loca" table, which contains the byte offsets of glyph descriptions in the TrueType "glyf" table. If the font dictionary also has a **GlyphDirectory** entry, the integer is instead used as an index or key in **GlyphDirectory**. The value of an entry in **CharStrings** can also be a PostScript procedure; see Section 5.9.3, "Replacing or Adding Individual Glyphs." This dictionary must have an entry whose key is .notdef.

sfnts	array	*(Required)* An array of one or more strings which, when concatenated, are treated as the binary representation of the TrueType font. (Multiple strings may be required because of the implementation limit on the length of a string; see Appendix B.) See also Adobe Technical Note #5012, *The Type 42 Font Format Specification.*
PaintType	integer	*(Required)* A code indicating how the glyphs of the font are to be painted: 0 Glyph outlines are filled. 2 Glyph outlines (designed to be filled) are stroked.
StrokeWidth	number	*(Optional)* The stroke width (in units of the glyph coordinate system) for stroked-outline fonts (**PaintType** 2). This entry is not initially present in filled-outline fonts. It should be added (with a nonzero value) when a stroked font is created from an existing filled font. Default value: 0.
Metrics	dictionary	*(Optional)* A dictionary containing metric information (glyph widths and sidebearings) for writing mode 0. This entry is not normally present in the original definition of a font. Adding a **Metrics** entry to a font overrides the metric information encoded in the glyph descriptions. See Sections 5.4, "Glyph Metric Information," and 5.9.2, "Changing Glyph Metrics."
Metrics2	dictionary	*(Optional)* Similar to **Metrics**, but for writing mode 1.
CDevProc	procedure	*(Optional)* A procedure that algorithmically derives global changes to the font's metrics. LanguageLevel 1 implementations lacking composite font extensions ignore this entry.
GlyphDirectory	array or dictionary	*(Optional)* A mechanism for the subsetting or incremental definition of glyphs in the font; see Section 5.9.4, "Subsetting and Incremental Definition of Glyphs."

Type 42 fonts are usually defined in terms of an identity transform, so the value of **FontMatrix** (which transforms the glyph coordinate system into the user coordinate system) should be `[1 0 0 1 0 0]` in a Type 42 font dictionary. This is in contrast to a Type 1 font, whose glyph coordinate system is typically defined at a 1000-unit scale relative to user space. This difference has implications related to the interpretation of font dictionary entries whose values are defined in glyph space. If a PostScript program adds or changes such entries, it must choose values that are appropriate to the font's glyph coordinate system; values appropriate for

a Type 1 font would be 1000 times too large for a Type 42 font. The font dictionary entries for which this issue arises include:

- The value of **StrokeWidth** (when **PaintType** has been set to 2)
- The contents of the **Metrics** and **Metrics2** dictionaries
- The operands and result of the **CDevProc** procedure
- The values of **UnderlinePosition** and **UnderlineThickness** in the **FontInfo** dictionary

5.9 Font Derivation and Modification

A PostScript program can perform various useful manipulations on existing font dictionaries. These manipulations fall into two categories:

- Derive a new font by copying an existing one and modifying certain things in the copy, such as the encoding vector, character set, glyph metrics, or other entries affecting the font's behavior.
- Alter an existing font in place, without copying it. This is permitted only under certain circumstances; it is useful mainly for incremental definition of glyph descriptions.

When such modifications are performed, considerable care must be taken to ensure predictable behavior despite font caching. When creating a new font derived from an existing one, a PostScript program must manage the unique ID properly to ensure that the interpreter can distinguish between the fonts; see Section 5.6, "Unique ID Generation." Alterations to an existing font must be limited to the cases described below; the font machinery consults the font dictionary at unpredictable times and will not necessarily detect modifications made in violation of these guidelines.

All of the facilities described in this section are PostScript language features, which are intended to be applied to font dictionaries already existing in PostScript VM. They are *not* part of the external font format specifications, such as *Adobe Type 1 Font Format*. It is inappropriate to create external font programs that use these features, since they will not work with font interpreters (such as the Adobe Type Manager® software) that lack a PostScript interpreter.

This section applies to base fonts. Similar techniques apply to some types of CIDFonts, as described in Section 5.11.3, "CIDFont Dictionaries."

5.9.1 Changing the Encoding Vector

The most common font derivation is the installation of a different encoding vector, discussed in Section 5.3, "Character Encoding." Example 5.8 creates a copy of the Helvetica font in which the Adobe standard encoding for the font is replaced by the ISO Latin-1 encoding, described in Appendix E.

Example 5.8

```
/Helvetica findfont
dup length dict begin
    {    1 index /FID ne
                {def}
                {pop pop}
            ifelse
    } forall
    /Encoding ISOLatin1Encoding def
    currentdict
end
/Helvetica-ISOLatin1 exch definefont pop
```

This program performs the following steps:

1. Makes a copy of the font dictionary, including all entries except the one whose key is **FID**. (This exclusion is necessary only in LanguageLevel 1; in LanguageLevel 2, the interpreter ignores any existing **FID** entry in a font being defined.)
2. Installs the desired change: replaces the font's **Encoding** array with the value of **ISOLatin1Encoding**, which is a built-in, 256-element array of character names defined in **systemdict**.
3. Registers this modified font under a new name (Helvetica-ISOLatin1).

In Type 1 fonts, the glyphs for some accented characters are produced by combining the glyphs for two or more other characters (such as a letter and an accent) defined in the same font. In LanguageLevel 1, if an encoding vector includes the name of an accented character, it must also include the names of the components of that character.

Note: *If you create a new encoding for a Type 1 font, Adobe suggests that you place the accents in control character positions, which are typically unused. The built-in* ***ISOLatin1Encoding*** *array uses this technique.*

As stated in Section 5.6, "Unique ID Generation," it is unnecessary to remove or change the font's unique ID (**UniqueID** or **XUID** entry) if the **Encoding** entry is the only one that is modified. This is because glyph caching is based on character names, not character codes. However, changing any entries that affect the glyphs' appearance, such as **PaintType** or **StrokeWidth**, does require proper management of the unique ID.

5.9.2 Changing Glyph Metrics

A PostScript program may create a derived font whose glyph metrics are altered from their native values. This is accomplished by adding **Metrics**, **Metrics2** *(LanguageLevel 2)*, or **CDevProc** *(LanguageLevel 2)* entries to the derived font dictionary. Most standard types of fonts and CIDFonts support this method for overriding metrics; however, Type 3 base fonts and Type 1 and 4 CIDFonts do not.

Note: *Determining a pleasing and correct glyph spacing is a difficult and laborious art that requires considerable skill. A font's glyphs have been designed with certain metrics in mind. Changing those metrics haphazardly will almost certainly produce poor results.*

The **Metrics** entry is a dictionary consisting of entries whose keys are character names, as they appear in the **CharStrings** dictionary and **Encoding** array. (In a CIDFont, the keys are integer CIDs.) Entries in the **Metrics** dictionary override the normal metrics for the corresponding glyphs. The values of these entries take various forms. An entry's value may be:

- A single number, specifying a new *x* width only (the *y* value is 0)
- An array of two numbers, specifying the *x* components of a new left sidebearing and new width (the *y* values are 0)
- An array of four numbers, specifying the *x* and *y* components of the left sidebearing followed by *x* and *y* components of the width

These forms can be intermixed in one **Metrics** dictionary. All of the numeric values are expressed in the glyph coordinate system of the font.

In a font that supports two writing modes (as described in Section 5.4, "Glyph Metric Information"), the **Metrics** dictionary is used during writing mode 0. Another dictionary, **Metrics2**, is used during writing mode 1; the value of an entry in this dictionary must be an array of four numbers, which specify *x* and *y* components of *w1* followed by *x* and *y* components of *v* (see Figure 5.7 on page 333).

Whereas the **Metrics** and **Metrics2** dictionaries allow modifications of individual glyph metrics in a given font, a procedure named **CDevProc** allows global changes to a font's metrics to be algorithmically derived from the native metrics in the glyph descriptions.

If present, the **CDevProc** procedure is called after metric information has been extracted from the glyph description and from the **Metrics** and **Metrics2** dictionaries, but immediately before the interpreter makes an internal call to **setcachedevice2**. Eleven operands are on the stack: the ten values that are to be passed to **setcachedevice2** followed by the character's name (or CID, in the case of a CIDFont). On return, there should be ten values, which are then passed to **setcachedevice2**.

5.9.3 Replacing or Adding Individual Glyphs

As stated earlier, a number of font formats, including Type 1, contain a **CharStrings** dictionary that maps character names to glyph descriptions. Depending on the font type, the glyph description is represented by either a string (a charstring) or an integer (a glyph index).

For any such font, a PostScript program can create a derived font whose **CharStrings** dictionary is copied and altered. The alterations may include adding, removing, or changing keys associated with existing values. Additionally, any value may be replaced with a PostScript procedure. If a **CharStrings** entry is a procedure, the PostScript interpreter executes the procedure to render the glyph. (It is possible to replace .notdef the same as any other character.)

The required behavior of a **CharStrings** procedure is very similar to that of the **BuildGlyph** procedure for Type 3 fonts, described in Section 5.7, "Type 3 Fonts." The **CharStrings** procedure must perform essentially the same functions as a **BuildGlyph** procedure, including executing the **setcachedevice**, **setcachedevice2**, or **setcharwidth** operator. Unlike the situation with **BuildGlyph**, there is potentially a different procedure for each character, although several characters can share one procedure.

The execution environment of a **CharStrings** procedure is slightly different from that of a Type 3 **BuildGlyph** procedure.

- Before executing a **CharStrings** procedure, the PostScript interpreter first pushes **systemdict** and then the font dictionary on the dictionary stack, and pushes either the character code or the character name on the operand stack. The operand is a character code if the interpreter is in the midst of an ordinary **show** operation or any **show** variant that takes a string operand; it is a character name if the interpreter is executing the **glyphshow** operator *(LanguageLevel 2)*.
- After executing the procedure, the PostScript interpreter pops the two dictionaries that it pushed on the dictionary stack. It expects the procedure to have consumed the character code or character name operand.

Because a **CharStrings** procedure must be able to accept either a character code or a character name as an operand, it is strongly recommended that every **CharStrings** procedure begin as follows:

```
dup type /integertype eq
        {/Encoding load exch get}
    if
```

This ensures that the object on the stack is a name object, which the procedure can now use to look up the glyph description. If the glyph description is contained in the **CharStrings** procedure itself, the procedure can simply discard its operand.

Note that when executing a **CharStrings** procedure, the PostScript interpreter does not consult the value of **Metrics**, **Metrics2**, or **CDevProc**.

5.9.4 Subsetting and Incremental Definition of Glyphs

When an application or driver must embed a font definition in a PostScript page description, it may choose not to embed the entire font. Instead, it can create and embed a *subset* font that contains only the glyph descriptions that are actually referenced from **show** strings in the document. This practice reduces communication costs and VM use, but at the expense of considerable effort by the application to analyze the use of glyphs in the document and reconstruct the font program.

A subset font created in this way appears to be an ordinary, well-formed font that happens to have a smaller character set than the font from which it was derived. Of course, this font is useful only in the document for which it was created; it cannot usefully be extracted and used with some other document. Furthermore, the inclusion of a subset font precludes subsequent editing of the **show** strings in the document, since some characters in the font's normal character set are missing and cannot be shown.

The creation and use of subset fonts, as described above, requires no special PostScript language features. However, it does require the application or driver to have intimate knowledge of the font file format in order to disassemble and reassemble it. Furthermore, it requires complete analysis of the document's glyph use before generating the subset fonts. This is not always convenient, particularly for a driver that is called from an application programming interface or that translates from some other document format.

Some types of fonts can be defined to allow glyph descriptions to be added incrementally during the execution of a PostScript program. This practice is called *incremental definition*. With incremental definition, the glyph descriptions need not be present at the time the font is defined by **definefont**. A given glyph description needs to be defined only before the first use of that glyph.

This section presents the general conventions and considerations for incremental glyph definition, followed by details for Types 1, 3, and 42, which are the only base font types that support this capability. Incremental definition of CIDFonts is described in Section 5.11.3, "CIDFont Dictionaries."

General Considerations

When defining glyph descriptions incrementally, a PostScript program must follow some rules to ensure predictable behavior. Once **definefont** has been executed, only certain objects within an existing font can be updated; all others must be treated as if they were read-only (even though they may not actually have read-only access attributes).

Note: *Although none of the rules described in this section are enforced by the PostScript interpreter, violation of the rules may result in inconsistent and unpredictable behavior.*

The method for installing a glyph description in a font varies by font type; it is described in later sections. Regardless of the method used, incremental definition is subject to the following rules:

- A glyph description may only be *added* to a font. It may not replace an existing glyph description with a different value.
- The glyph description for a particular character name must be defined before the first **show** operation that selects this character.
- If a glyph description is added to a font in local VM between **save** and **restore** operations (such as the ones that usually bracket each page of a document), the **restore** operation will remove the glyph description. The application or driver that is embedding incremental glyph descriptions in the document must be aware of this behavior; it must define the glyph description again (with the same value) before the next use of that glyph.

These rules ensure predictable behavior despite font caching. A **show** operation may obtain a previously rendered glyph from the font cache, without consulting the glyph description at all. For consistent results, a given character name must always map to the same glyph description any time a **show** operation might reference it.

In font types that support overriding of glyph metrics, the contents of the **Metrics** and **Metrics2** dictionaries may be defined incrementally, subject to the same rules that apply to the glyph descriptions themselves.

Additionally, the contents of the **Encoding** array in a base font may be defined incrementally. This is a convenience in the situation where the application invents not only the character set but also the character encoding as it goes along. Similar rules apply to incremental definition of the **Encoding** array:

- A character name may replace only the name .notdef, thereby defining a character name for a previously unused character code. It may not replace an existing array element whose value is any name other than .notdef.
- The **Encoding** array entry for a particular character code must be defined before the first **show** operation that selects this code.
- The same considerations about **save** and **restore** operations apply to the **Encoding** array as to glyph descriptions.

Incremental Definition of Type 1 Fonts

In a Type 1 font, the contents of the **CharStrings** dictionary may be defined incrementally. Usually, this entails adding an entry whose key is a character name and whose value is a string containing a Type 1 glyph description (a charstring). Alternatively, the value can be a PostScript procedure, as described in Section 5.9.3, "Replacing or Adding Individual Glyphs."

A charstring can contain references to one or more other glyphs in the same font—for example, to produce accented characters; see *Adobe Type 1 Font Format*. An incrementally downloaded font containing such glyphs must also contain all component glyphs that they reference. Additionally, any subroutines that the charstrings call must be present at the time the font is defined (in the **Subrs** entry in the **Private** dictionary); subroutines cannot be defined incrementally.

Incremental Definition of Type 3 Fonts

Incremental definition is permitted for Type 3 fonts. How this is accomplished depends on how the PostScript program defines the font's **BuildGlyph** or **BuildChar** procedure. Because of the effects of font caching, it is still subject to the general considerations described above.

Incremental Definition of Type 42 Fonts

In a Type 42 font *(LanguageLevel 3)*, the glyph descriptions are normally embedded in the TrueType font data that is contained in the **sfnts** array; see Section 5.8.2, "Type 42 Fonts (TrueType)." To enable incremental definition, the glyph descriptions are omitted from the TrueType font data; instead, they are defined as strings in an array or dictionary named **GlyphDirectory** in the Type 42 font dictionary.

As described in Table 5.7 on page 346, a glyph description for a Type 42 font is represented by an integer glyph index that is the value of an entry in the **CharStrings** dictionary. In the absence of a **GlyphDirectory** entry, the glyph index is used to access the TrueType data directly; otherwise, it is used as an index or a key in **GlyphDirectory**, depending on whether **GlyphDirectory** is an array or a dictionary.

If **GlyphDirectory** is an array, its length must be greater than the highest glyph index used in the font. Each array element can be either *null* (denoting an empty element) or a string containing the TrueType glyph description for the corresponding glyph index. If **GlyphDirectory** is a dictionary, each key is a glyph index and the value is a string containing the TrueType glyph description. As for all fonts, there must be a glyph description for the .notdef character—that is, .notdef must map to a glyph index for which there is a glyph description in **GlyphDirectory**.

If **GlyphDirectory** is an array, any unused entries in the array will be wasted space. An array of a given length consumes about 40 percent of the memory used by a dictionary of the same length. Thus, the dictionary representation is advisable only for a sparsely populated font containing less than 40 percent of its characters.

GlyphDirectory can be used only with TrueType font data (contained in the **sfnts** entry) that meets the following requirements:

- The TrueType "loca" and "glyf" tables must not be present.
- There must be a TrueType "gdir" table whose size and offset are 0. This is simply an indication that the TrueType font was constructed for incremental downloading; the "gdir" table contains no useful information.

A TrueType glyph description can contain references to one or more other glyphs in the same font—for example, to produce accented characters. These references are by glyph index. An incrementally downloaded font containing such glyphs must also contain all component glyphs that they reference.

A TrueType glyph's metrics are not part of the glyph description itself; rather, they come from a parallel table, "hmtx," which is also indexed by glyph index. No provision is made for downloading the contents of the "hmtx" table incrementally; this information must be present at the time the font is defined. The metrics may be overridden, if desired, by entries in the **Metrics** and **Metrics2** dictionaries, which may be defined incrementally.

5.10 Composite Fonts

This section describes how to build *composite fonts*—fonts with a **FontType** of 0. Base fonts (fonts of type 1, 2, 3, 14, or 42) contain individual glyph descriptions; composite fonts are combinations of base fonts, with Type 0 font dictionaries tying them together.

The ability to use composite fonts is supported by LanguageLevels 2 and 3 and by LanguageLevel 1 implementations that have the composite font extensions. Additionally, LanguageLevel 3 introduces support for a special class of composite fonts called *CID-keyed fonts*, described in Section 5.11, "CID-Keyed Fonts"; these fonts can combine CIDFonts as well as base fonts.

A composite font, then, is a collection of base fonts, CIDFonts, and even other composite fonts, organized hierarchically. The Type 0 font at the top level of the hierarchy is the *root font*. Fonts and CIDFonts immediately below a Type 0 font are called its *descendant fonts.* The Type 0 font immediately above a descendant font is called its *parent font*.

When the current font is composite, the **show** operator and its variants behave differently than with base fonts: they use a mapping algorithm that decodes **show** strings to select characters from any of the descendant base fonts or CIDFonts. This facility supports the use of very large character sets, such as those for the Japanese and Chinese languages. It also simplifies the organization of fonts that have complex encoding requirements.

In addition to the entries common to all font dictionaries (Table 5.2 on page 324), a Type 0 font dictionary includes the entries listed in Table 5.8.

TABLE 5.8 Additional entries specific to Type 0 fonts

KEY	TYPE	VALUE
FMapType	integer	*(Required)* A code indicating which mapping algorithm to use when interpreting the sequence of bytes in a string. See Table 5.9 on page 360.
Encoding	array	*(Required)* An array of integers, each used as an index to extract a font dictionary from the **FDepVector** array. Note that this is different from the use of **Encoding** in base fonts.
FDepVector	array	*(Required)* An array of font or CIDFont dictionaries that are the descendants of this Type 0 font. It is recommended that this array contain no more than

		one reference to any given descendant. The **Encoding** array can be used to represent any desired repetitions.
EscChar	integer	*(Optional)* The escape code value, used only when **FMapType** is 3 or 7. If this entry is needed but is not present, **definefont** inserts one with the value 255.
ShiftOut	integer	*(Optional)* The shift code value, used only when **FMapType** is 8. If this entry is needed but is not present, **definefont** inserts one with the value 14.
ShiftIn	integer	*(Optional)* The shift code value, used only when **FMapType** is 8. If this entry is not present but is needed, **definefont** inserts one with the value 15.
SubsVector	string	*(Optional)* A table that specifies the division of codes into ranges, used only when **FMapType** is 6.
CMap	dictionary	*(Optional; LanguageLevel 3)* A CMap dictionary, used only when **FMapType** is 9. See Section 5.11.4, "CMap Dictionaries."
PrefEnc	array	*(Optional)* An array that is usually the same as the **Encoding** array of the majority of the descendant base fonts. If this entry is not initially present, **definefont** may insert one with a null value. See Section 5.10.2, "Other Dictionary Entries for Type 0 Fonts."
CurMID	any	*(Optional)* An object that serves internal purposes of the implementation. **definefont** may insert this entry.
MIDVector	any	*(Optional)* An object that serves internal purposes of the implementation. **definefont** may insert this entry.

5.10.1 Character Mapping

The **FMapType** entry in the Type 0 font dictionary indicates which mapping algorithm will be used to interpret the sequence of bytes in a **show** string. Instead of each byte selecting a character independently, as is done for base fonts, the **show** string encodes a more complex sequence of font and character selections. The mapping algorithm does the following:

1. Decodes bytes from the **show** string to determine a *font number* and a *character selector*. The character selector is always a character code unless the composite font is a CID-keyed font (as described in Section 5.11, "CID-Keyed Fonts").
2. Uses the font number as an index into the **Encoding** array of the Type 0 font, obtaining an integer.

3. Uses that integer in turn as an index into the **FDepVector** array, selecting a descendant font or CIDFont and temporarily establishing it as the current font.
4. Uses the character selector to select a character from the descendant, in whatever way is appropriate for that font or CIDFont.

Figure 5.9 illustrates this mapping algorithm for a composite font with Type 1 fonts as its descendants.

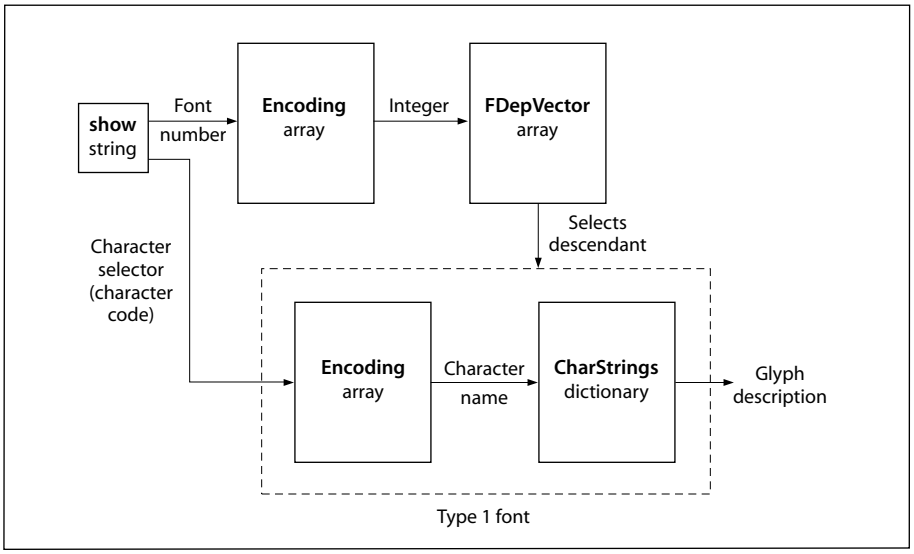


FIGURE 5.9 *Composite font mapping example*

The code length—the number of bytes extracted from the **show** string for each operation of the mapping algorithm—varies depending on the algorithm. Table 5.9 lists the mapping algorithms that the **FMapType** value can select. If the mapping of any string passed to a **show** operator is incomplete or if an attempt is made to index beyond the end of an **Encoding** or **FDepVector** array, a **rangecheck** error results.

TABLE 5.9 FMapType mapping algorithms

ALGORITHM	FMAPTYPE	EXPLANATION
8/8 mapping	2	Two bytes are extracted from the **show** string. The first byte is the font number and the second is the character code.
Escape mapping	3	One byte is extracted from the **show** string. If it is equal to the value of the font's **EscChar** entry, the next byte is the font number, and subsequent bytes (until the next escape code) are character codes for that font. At the beginning of a **show** string, font 0 is selected. A font number equal to the escape code is treated specially; see Section 5.10.3, "Nested Composite Fonts."
1/7 mapping	4	One byte is extracted from the **show** string. The most significant bit is the font number, and the remaining 7 bits are the character code.
9/7 mapping	5	Two bytes are extracted from the **show** string and combined to form a 16-bit number, high-order byte first. The most significant 9 bits are the font number, and the remaining 7 bits are the character code.
SubsVector mapping	6	One or more bytes are extracted from the **show** string and decoded according to information in the **SubsVector** entry of the font. The format of **SubsVector** is described below.
Double escape mapping	7	This mapping is very similar to **FMapType** 3. However, when an escape code is immediately followed by another escape code, a third byte is extracted from the **show** string. The font number is the value of this byte plus 256.
Shift mapping	8	This mapping provides exactly two descendant fonts. A byte is extracted from the **show** string. If it is equal to the value of the font's **ShiftIn** entry, subsequent bytes are character codes for font 0; if equal to the **ShiftOut** entry, subsequent bytes are character codes for font 1. At the beginning of a **show** string, font 0 is selected.
CMap mapping	9	*(LanguageLevel 3)* One or more bytes are extracted from the **show** string and decoded according to information in the font's **CMap** entry. See "CMap Mapping" on page 388.

SubsVector Mapping

SubsVector is a string that controls the mapping algorithm for a Type 0 font with an **FMapType** value of 6. This mapping algorithm allows the space of character

codes to be divided into ranges, where each range corresponds to one descendant font. The ranges can be of irregular sizes that are not necessarily powers of 2.

The first byte of a **SubsVector** string specifies 1 less than the code length. A value of 0 specifies a code length of 1 byte, 1 specifies 2 bytes, and so on. When a character code is longer than 1 byte, the bytes are interpreted high-order byte first. The code length cannot exceed the number of bytes representable in an integer (see Appendix B).

The remainder of the **SubsVector** string defines a sequence of ranges of consecutive code values. The first range is the one for font 0, the second range is the one for font 1, and so on. Each range is described by one or more bytes; the number of bytes is the same as the code length. The value contained in those bytes (interpreted high-order byte first) gives the *size* of the code range. There is an implicit code range at the end of the sequence that contains all remaining codes; this range should not be specified explicitly.

When using a **SubsVector** mapping, the **show** operator interprets a character code extracted from the **show** string as follows:

1. Determines the code range that contains the character code. The position of the code range in the **SubsVector** sequence (counting from 0) is used as the index into the font's **Encoding** array, selecting a descendant font.
2. Subtracts the base of the code range from the character code. The result is treated as a character code to select a character from the descendant font.

The following examples show how some of the other mapping algorithms could be described in terms of the **SubsVector** mapping. This is for illustrative purposes only; the other mapping algorithms should be used rather than the **SubsVector** mapping if they achieve the desired effect. The **SubsVector** strings are shown as hexadecimal string literals.

- 1/7 mapping: <00 80>

  The code length is 1 byte. There are two code ranges. The first is explicitly of length 80 hexadecimal; it contains character codes 0 to 127 decimal. The second code range implicitly contains all remaining characters that can be coded in 1 byte—that is, character codes in the range 128 to 255.

- 9/7 mapping: <01 0080 0080 … 0080>

  The code length is 2 bytes. There are up to 512 code ranges, each 80 hexadecimal (128 decimal) in size. The **SubsVector** string that describes all 512 code ranges would be 1023 bytes long. Remember that the last code range is specified implicitly.

- 8/8 mapping: <01 0100 0100 … 0100>

  The code length is 2 bytes. There are up to 256 code ranges, each 100 hexadecimal (256 decimal) in size. The **SubsVector** string that describes all 256 code ranges would be 511 bytes long. The last code range is specified implicitly.

5.10.2 Other Dictionary Entries for Type 0 Fonts

FontMatrix plays the same role in a Type 0 font as it does in a base font. When a glyph is shown, both the font matrix of the Type 0 font and the font matrix of the descendant base font or CIDFont are concatenated to the CTM. (Special considerations apply if the descendant of a Type 0 font is itself a Type 0 font; see Section 5.10.3, "Nested Composite Fonts.")

WMode is an integer with value 0 or 1, indicating which of two sets of glyph metrics will be used when glyphs from the base fonts are shown (see Section 5.4, "Glyph Metric Information"). If it is omitted, writing mode 0 will be used. The writing mode of the root font overrides the writing modes of all its descendants. This allows a given base font to be used as part of many composite fonts, some of which use writing mode 0 while some use writing mode 1.

PrefEnc (preferred encoding) is an array that should be the same as the **Encoding** array of one or more of the descendant base fonts. Characters from descendant fonts whose **Encoding** array is the same as the **PrefEnc** array of the Type 0 font will be processed more efficiently than characters from other descendant fonts.

The **definefont** operator may insert one or more additional entries in a Type 0 font dictionary if they are needed but are not present. **FID** is always inserted; see Table 5.2 on page 324. **EscChar**, **ShiftIn**, and **ShiftOut** are inserted if they are required by the mapping algorithm indicated by the **FMapType** entry. **PrefEnc**, **MIDVector**, and **CurMID** are inserted by some implementations. In a Language-Level 1 implementation that supports the composite font extensions, the font dictionary must be sufficiently large to allow these entries to be inserted.

5.10.3 Nested Composite Fonts

The descendant fonts in a composite font may themselves be composite fonts, nested to a maximum depth of five levels. The mapping algorithms nest according to two sets of rules, depending on whether the constituent Type 0 fonts are modal or nonmodal.

Type 0 fonts with **FMapType** 3, 7, or 8 are *modal* fonts: some byte codes select a descendant font, and then successive bytes of the **show** string are interpreted with respect to the selected font until a new descendant font is selected. Modal fonts follow these rules:

- The parent of an **FMapType** 3 font must be of **FMapType** 3 or 7. The **EscChar** entry of the root font overrides the **EscChar** entry of descendant escape-mapped fonts.
- Fonts with **FMapType** 7 and 8 may not be used as descendant fonts.
- The occurrence of an escape or shift code in the **show** string causes the mapping algorithm to ascend the font hierarchy from the currently selected descendant font to the nearest parent modal font. If that font's **FMapType** is 8, the algorithm selects the new descendant according to the shift code. If **FMapType** is 3 or 7, the algorithm extracts another byte from the **show** string. If the byte is not an escape code, the algorithm uses it as a font number to select a descendant of that font. But if the byte is an escape code and **FMapType** is 3, the algorithm ascends to the parent of that font, extracts yet another byte from the **show** string, and repeats the selection process.
- When a modal font is first encountered, if the next byte of the **show** string is not an escape code, descendant font 0 of the modal font is chosen and the byte is passed down to that font. This also occurs if an escape code is followed by another escape code but the currently selected font has no parent.

Type 0 fonts with the other **FMapType** values (2, 4, 5, 6, 9) are *nonmodal,* in that their mapping algorithm restarts for each new character. Nonmodal fonts follow these rules:

- The parent of a nonmodal font may be any Type 0 font, including a modal font.
- If the parent of a nonmodal font is a modal font, the modal font's escape or shift code is recognized only when it appears as the *first* byte of a multiple-byte mapping sequence for the nonmodal font.

- If the descendant of a nonmodal Type 0 font is itself a nonmodal Type 0 font, the second part (character code) of the value extracted from the **show** string is used in place of the first byte that would be extracted by the descendant font's mapping algorithm. This rule is independent of the number of bits actually contained in the code contributed by the parent font.

The **FontMatrix** entries of nested composite fonts are treated in a nonobvious way. When a glyph is shown, the interpreter consults the **FontMatrix** entries of only the selected base font and the immediate parent of the base font. The immediate parent's **FontMatrix** entry contains the concatenation of the **FontMatrix** entries of all ancestor fonts. To achieve this, the **definefont**, **makefont**, **scalefont**, and **selectfont** operators give special treatment to any Type 0 font that has at least one descendant Type 0 font:

- If the **FontMatrix** value is not the identity matrix, **definefont** constructs a new **FDepVector** array in which each descendant Type 0 font is replaced by the result of performing **makefont** on it using this font matrix. It does not perform **makefont** on descendant base fonts or CIDFonts.
- **makefont**, **scalefont**, and **selectfont** apply their transformations recursively to all descendant Type 0 fonts but not to base fonts or CIDFonts.

5.11 CID-Keyed Fonts

CID-keyed fonts provide a convenient and efficient method for defining multiple-byte character encodings, base fonts with a large number of glyphs, and composite fonts that use these base fonts and character encodings. Additionally, they provide straightforward methods for creating a *rearranged font*, which selects glyphs from one or more existing fonts by means of a revised encoding. These capabilities provide great flexibility for representing text in writing systems for languages with large character sets, such as Chinese, Japanese, and Korean.

The CID-keyed font architecture specifies the external representation of certain font programs, called *CMap* and *CIDFont* files, along with some conventions for combining and using those files. This architecture is independent of the PostScript language; CID-keyed fonts can be used in environments where no PostScript interpreter is present. For complete documentation on the architecture and the file formats, see Adobe Technical Notes #5092, *CID-Keyed Font Technology Overview*, and #5014, *Adobe CMap and CIDFont Files Specification*.

This section describes the PostScript language support for CID-keyed fonts—that is, their representation and behavior as objects in PostScript VM, as distinct from their external file representation. Support for CID-keyed fonts is provided in LanguageLevel 3 through the **CIDFont** and **CMap** resource categories and the **composefont** operator. If a PostScript program accesses CID-keyed fonts using these facilities, the CMap and CIDFont files materialize directly as dictionary objects in VM, as documented in this section.

An alternative method for using CID-keyed fonts makes use of the CID Support Library (CSL). The CSL is a separate package of software, implemented in the PostScript language, that accompanies CID-keyed font products from Adobe Systems. The CSL serves two purposes:

- It enables CID-keyed fonts to be used with PostScript interpreters that do not have built-in support for them.
- It provides compatibility with applications that access Chinese, Japanese, and Korean fonts according to older conventions for identifying and organizing them. Under this *compatibility mode*, accessing a CID-keyed font sometimes results in a composite font hierarchy in VM that bears little resemblance to the structure of the CMap and CIDFont programs.

The CSL is not further documented in this book. For information, see Adobe Technical Note #5092, *CID-Keyed Font Technology Overview*. The CSL, certain CMap files, and other related software are available from the Adobe Developers Association.

5.11.1 The Basics of CID-Keyed Fonts

The term *CID-keyed font* reflects the fact that *CID* (character identifier) numbers are used to index and access the glyph descriptions in the font. This method is more efficient for large fonts than the method of accessing by character name, as is used for base fonts. CIDs range from 0 to a maximum value that is subject to an implementation limit (see Appendix B).

A *character collection* is an ordered set of all characters needed to support one or more popular character sets for a particular language. The order of the characters in the character collection determines the CID number for each character. Each CID-keyed font must explicitly reference the character collection on which its CID numbers are based; see Section 5.11.2, "CIDSystemInfo Dictionaries."

A *CMap* (character map) file specifies the correspondence between character codes and the CID numbers used to identify characters. It is equivalent to the concept of an encoding vector as used in base fonts. Whereas a base font allows a maximum of 256 characters to be encoded and accessible at one time, a CMap can describe a mapping from multiple-byte codes to thousands of characters in a large CID-keyed font. For example, it can describe JIS, one of several widely used encodings for Japanese, or ISO 10646 (Unicode®), an international standard encoding that covers many languages.

A CMap can reference an entire character collection, a subset, or multiple character collections. It can also reference characters in base fonts (by character code or character name) or composite fonts (by character code). The CMap mapping yields a *font number* and a *character selector* that can be a CID, a character code, or a character name. Furthermore, a CMap can incorporate another CMap by reference, without having to duplicate it. These features enable character collections to be combined or supplemented, and make all the constituent characters accessible to **show** operations through a single encoding.

A *CIDFont* file contains the glyph descriptions for a character collection. The glyph descriptions themselves are typically in a format similar to those used in base fonts, such as Type 1. However, they are identified by CIDs rather than by names, and they are organized differently.

In the PostScript language, CMap and CIDFont files are treated as instances of the **CMap** and **CIDFont** resource categories, respectively. When loaded into VM, they are represented as dictionaries, whose contents are documented in Sections 5.11.4, "CMap Dictionaries," and 5.11.3, "CIDFont Dictionaries." As stated earlier, the external file formats are not documented here, but in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*.

Finally, a *CID-keyed font* is the combination of a CMap with one or more CIDFonts, base fonts, or composite fonts containing glyph descriptions. In the PostScript language, a CID-keyed font is represented as a Type 0 font whose **FMapType** value is 9. It contains a **CMap** entry whose value is a CMap dictionary, and its **FDepVector** array references the CIDFont, base font, or composite font dictionaries with which the CMap has been combined. The **composefont** operator provides a convenient means for creating a CID-keyed font dictionary.

Figure 5.10 illustrates the concepts introduced above, including the mapping from a CMap into a descendant of a CID-keyed font (with a Type 1 font as the example descendant base font). Details are provided in the subsections that follow.

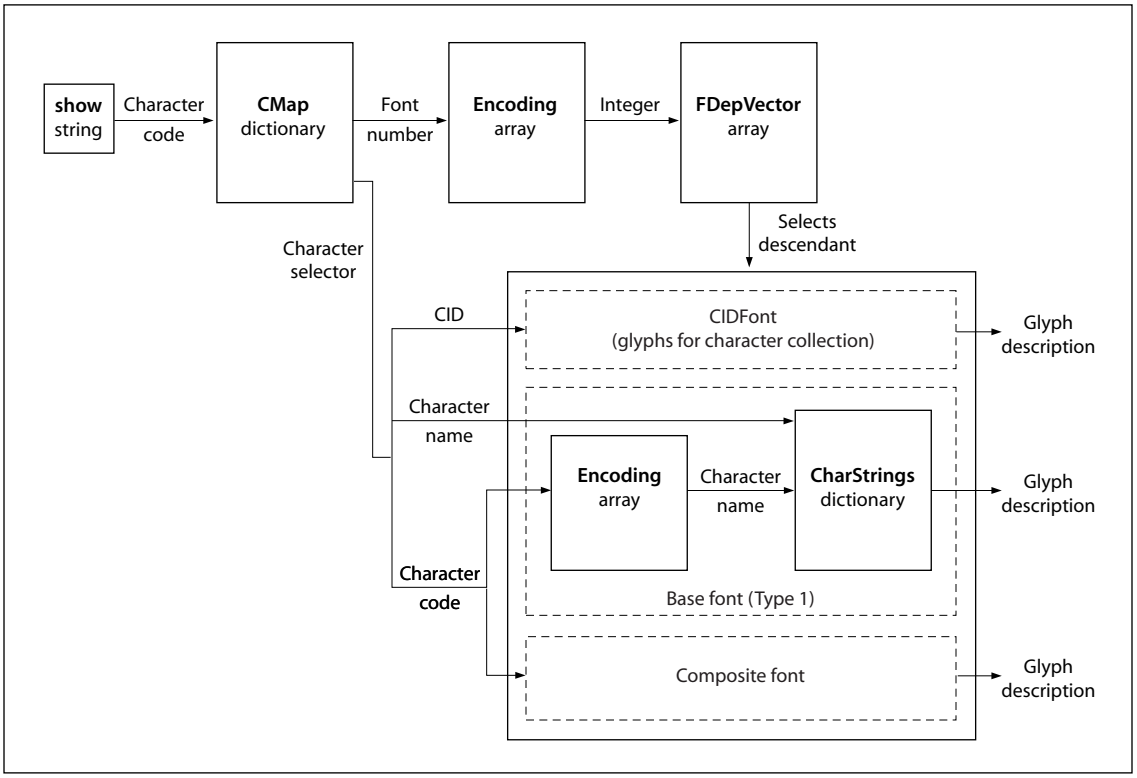


FIGURE 5.10 *CID-keyed font basics*

5.11.2 CIDSystemInfo Dictionaries

CIDFont and CMap dictionaries contain a **CIDSystemInfo** entry that specifies the character collection that the resource assumes—that is, the interpretation of the CID numbers it uses. A character collection is uniquely identified by the **Registry**, **Ordering**, and **Supplement** entries in the **CIDSystemInfo** dictionary, as described in Table 5.10. Character collections whose **Registry** and **Ordering** values are the same are compatible.

TABLE 5.10 Entries in a CIDSystemInfo dictionary

KEY	TYPE	VALUE
Registry	string	*(Required)* A string identifying an issuer of character collections—for example, Adobe. For information about assigning a registry identifier, consult the Adobe Developer Relations Web site (see the Bibliography) or send e-mail to the Adobe Unique ID Coordinator at fontdev-person@adobe.com.
Ordering	string	*(Required)* A string that uniquely names a character collection issued by a specific registry—for example, Japan1.
Supplement	integer	*(Required)* The *supplement number* of the character collection. An original character collection has a supplement number of 0. Whenever additional CIDs are assigned in a character collection, the supplement number is increased. Supplements do not alter the ordering of existing CIDs in the character collection. This value is not used in determining compatibility between character collections.

In a CIDFont, the **CIDSystemInfo** entry is a dictionary that specifies the CIDFont's character collection. Note that the CIDFont need not contain glyph descriptions for all the CIDs in a collection; it can contain a subset. In a CMap, the **CIDSystemInfo** entry is either a single dictionary or an array of dictionaries, depending on whether it associates codes with a single character collection or with multiple character collections; see Section 5.11.4, "CMap Dictionaries."

For proper behavior, the **CIDSystemInfo** entry of a CMap should be compatible with that of the CIDFont or CIDFonts with which it is used. The PostScript interpreter (specifically, the **composefont** operator) does not enforce compatibility. However, the CID Support Library (CSL) and other application and support software do depend on it.

5.11.3 CIDFont Dictionaries

A CIDFont dictionary contains glyph descriptions that are accessed using a CID as the character selector. The glyph descriptions may take the form of Type 1 charstrings, **BuildGlyph** procedures, TrueType glyph descriptions, or bitmaps installed in the font cache. The **CIDFontType** entry in the CIDFont dictionary indicates which kind it is.

Every CIDFont must contain a glyph description for CID 0, which is analogous to the .notdef character name in base fonts. See "Handling Undefined Characters" on page 389.

As explained earlier, the primary use of CIDFont dictionaries is as descendants in CID-keyed fonts, which are composite fonts whose root Type 0 font has an **FMapType** value of 9. However, a CIDFont dictionary may be treated as if it were a font dictionary by the **makefont**, **scalefont**, **selectfont**, and **setfont** operators. The following limitations apply when a CIDFont is the current font:

- The **glyphshow** operator can be used, but it accepts only an integer operand specifying a CID as the character selector.
- The **show** operator, any **show** variant except **glyphshow**, and the **stringwidth** and **charpath** operators cannot be used. (Some exceptions to this rule are described in the section "CMap Mapping" on page 388 and in the **cshow** operator description in Chapter 8.)

Having a CIDFont as a descendant in a composite font raises certain compatibility issues. Applications that manipulate composite font hierarchies expect all descendants to have a **FontType** entry. Furthermore, applications sometimes produce modified descendants (changing **PaintType**, for example) and invoke **definefont** on those fonts. For these reasons, a CIDFont is treated as if it were a font in the following respects:

- Every CIDFont dictionary also includes a **FontType** entry (inserted by **definefont** or **defineresource**).
- A CIDFont dictionary can be treated as an instance of either the **CIDFont** resource category or the **Font** resource category, with no difference in structure or behavior except the resource category in which the instance gets defined.
- All font operators are prepared to recognize either a CIDFont dictionary or a font dictionary. These operators distinguish between a CIDFont and a font by the presence or absence of a **CIDFontType** entry in the dictionary.

Table 5.11 lists the **CIDFontType** and **FontType** values corresponding to the different types of CIDFont. Note that references to a CIDFont by type, as in "Type 0 CIDFont," indicate the font's **CIDFontType**, not its **FontType**.

TABLE 5.11 CIDFontType and FontType values

CIDFONTTYPE	FONTTYPE	GLYPH DESCRIPTIONS
0	9	Charstrings, as in Type 1 base fonts
1	10	**BuildGlyph** procedures, similar to Type 3 base fonts
2	11	TrueType glyph descriptions, similar to Type 42 base fonts
4	32	Bitmaps installed in the font cache

Table 5.12 lists the entries that have defined meanings in all CIDFont dictionaries, regardless of the **CIDFontType** value.

TABLE 5.12 Entries common to all CIDFont dictionaries

KEY	TYPE	VALUE
CIDFontType	integer	*(Required)* The CIDFont type; see Table 5.11. Indicates where the glyph descriptions are to be found and how they are represented.
CIDFontName	name	*(Required, except for **CIDFontType** 4)* The name of the CIDFont. Ordinarily, this is the same as the key given to **defineresource** when defining the dictionary as an instance of the **CIDFont** resource category. The **StartData** operator, used during the creation of a Type 0 CIDFont, requires **CIDFontName** to be interpreted in this way.
CIDSystemInfo	dictionary	*(Required, except for **CIDFontType** 4)* The character collection used by the CIDFont. The entries contained in the **CIDSystemInfo** dictionary are listed in Table 5.10 on page 368.
FontBBox	array	*(Required)* An array of four numbers in the glyph coordinate system giving the left, bottom, right, and top coordinates, respectively, of the font bounding box. See Table 5.3 on page 325.
FontMatrix	array	*(Required, except for **CIDFontType** 0)* An array that transforms the glyph coordinate system into the user coordinate system (see Section 5.4, "Glyph Metric Information"). See the next section for special considerations that apply only to Type 0 CIDFonts.
FontType	integer	*(Inserted by **definefont** or **defineresource**)* The font type; see Table 5.11. In a Type 2 CIDFont, this entry must be present with value 42 at the time **definefont** or **defineresource** is invoked; its value is replaced with 11.
FontInfo	dictionary	*(Optional)* A dictionary containing font information that is not accessed by the PostScript interpreter; see Table 5.5 on page 327.

UIDBase	integer	*(Optional)* An integer used in combination with the **UIDOffset** entry in a CMap to form **UniqueID** entries for the base fonts within a composite font. This entry is used only in PostScript interpreters lacking built-in CID-keyed font support but having other means, such as the CID Support Library (CSL), to achieve similar capabilities.
WMode	integer	*(Optional)* The writing mode, which indicates which of two sets of metrics will be used when glyphs are shown from the font; see Section 5.4, "Glyph Metric Information." This value applies only when characters are shown directly from the CIDFont. When the CIDFont is used as a descendant in a composite font, this value is overridden by the composite font's writing mode. Default value: 0.
XUID	array	*(Optional)* An array of integers that uniquely identifies this font or any variant of it; see Section 5.6, "Unique ID Generation."
FID	fontID	*(Inserted by **definefont** or **defineresource**)* A special object of type fontID that serves internal purposes in the font machinery.

Type 0 CIDFonts

A Type 0 CIDFont is the analog of a Type 1 base font whose charstrings (Type 1 glyph descriptions) are identified by CIDs rather than by character names. A CIDFont file is partitioned into two contiguous sections:

- A *PostScript section* that defines the CIDFont dictionary, as described below
- A *binary data section* that contains all the charstrings and any subroutines that they call

The two sections are separated by an invocation of the **StartData** operator defined in the **CIDInit** procedure set. The binary data section begins immediately following the white-space character that delimits the **StartData** token. The length of the binary data section is given as an operand to **StartData**. **StartData** processes the binary data, completes the construction of the CIDFont dictionary, and automatically invokes **defineresource** to define it as an instance of the **CIDFont** category, using the value of the **CIDFontName** entry as its key. For further information on the construction of a Type 0 CIDFont, see Adobe Technical Note #5014, *Adobe CMap and CID Font Files Specification*.

The PostScript section defines the CIDFont dictionary, which contains many of the entries found in Type 1 font dictionaries, as well as various other data struc-

tures that contain information required to interpret the binary data section. As shown in Figure 5.11, the **FDArray** entry in the **CIDFont** dictionary is an array of subsidiary dictionaries, each of which contains a few of the entries found in Type 1 fonts (including **FontMatrix** and **Private**) and is used by a subset of the charstrings in the CIDFont. For example, the charstrings access subroutines through information found in the **Private** dictionary.

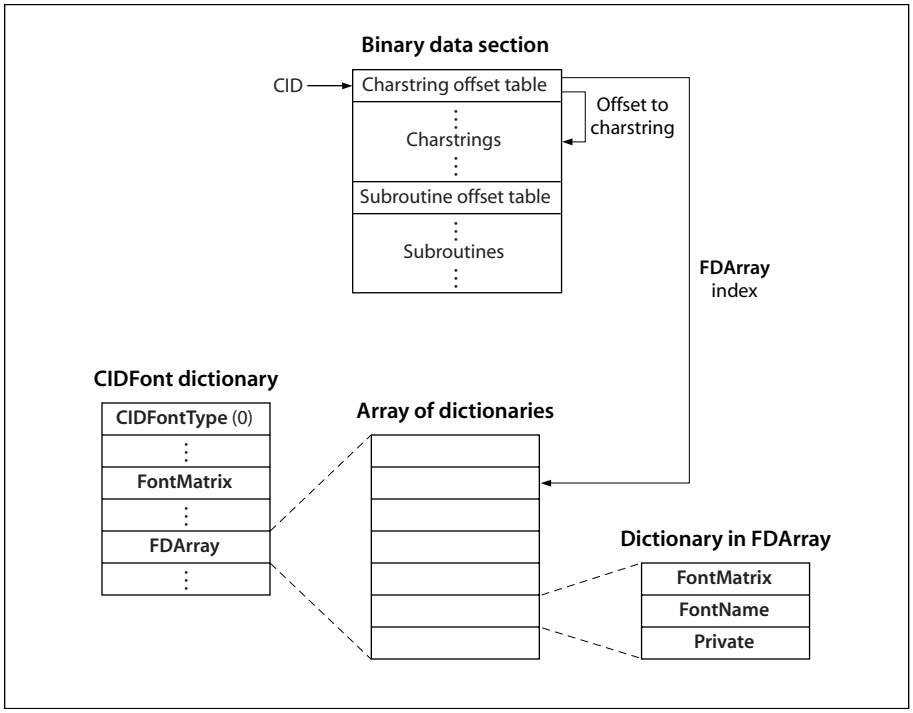


FIGURE 5.11 *Type 0 CIDFont character processing*

The binary data section contains charstrings that conform to the same format as those in the **CharStrings** dictionary in a Type 1 font. An offset table in this section is indexed by CID; for each CID, it contains an **FDArray** index (**FDBytes** long) followed by an offset (**GDBytes** long). The **FDArray** index specifies which **FDArray** dictionary to use for **Private** information; the offset gives the location of the charstring for that CID.

In addition to the entries common to all CIDFont dictionaries (Table 5.12 on page 370), Type 0 CIDFont dictionaries includes the entries listed in Table 5.13.

TABLE 5.13 Additional entries specific to Type 0 CIDFont dictionaries

KEY	TYPE	VALUE
CIDCount	integer	*(Required)* The number of valid CIDs in the CIDFont. Valid CIDs range from 0 to (**CIDCount** − 1); CIDs outside this range are treated as undefined glyphs.
GDBytes	integer	*(Required)* The length in bytes of the offset to the charstring for each CID in the CIDFont. If the length is greater than 1, the bytes within an offset are interpreted high-order byte first.
CIDMapOffset	integer	*(Required)* The offset in bytes to the charstring offset table in the binary data section of the CIDFont. The offset is relative to the beginning of the binary data section and is typically 0. The table is indexed by CID, and each entry consists of an **FDArray** index (**FDBytes** long) followed by an offset (**GDBytes** bytes long) to the charstring relative to the beginning of the binary data section.
FDArray	array	*(Required)* An array of dictionaries containing private information used with charstrings to render glyphs. There must be at least one dictionary in this array; typically there are more. A dictionary in **FDArray** contains the entries listed in Table 5.14. Glyphs having common private information reference the same dictionary in **FDArray**.
FDBytes	integer	*(Required)* The length in bytes of the **FDArray** index for each CID. If **FDBytes** is 0, the charstring offset table contains no **FDArray** indices, and an **FDArray** index of 0 is assumed for all CIDs.
PaintType	integer	*(Optional)* A code indicating how the glyphs of the CIDFont are to be painted: 0 Glyph outlines are filled. 2 Glyph outlines (designed to be filled) are stroked. Default value: 0.
StrokeWidth	integer	*(Optional)* The stroke width (in units of the glyph coordinate system) for stroked-outline fonts (**PaintType** 2). This entry is not initially present in filled-outline CIDFonts. It should be added (with a nonzero value) when a stroked font is created from an existing filled font. Default value: 0.
Metrics	dictionary	*(Optional)* A dictionary containing the metric information (glyph widths and sidebearings) for writing mode 0. This entry is not normally present in the original definition of a CIDFont. Adding a **Metrics** dictionary to a CIDFont overrides the metric information encoded in the glyph descriptions. See Sections 5.4, "Glyph Metric Information," and 5.9.2, "Changing Glyph Metrics." In this dictionary, the keys are CIDs and the values are specified in the units defined by the **FontMatrix** entry in the top-level CIDFont dictionary.

Metrics2	dictionary	*(Optional)* Similar to **Metrics**, but for writing mode 1.
CDevProc	procedure	*(Optional)* A procedure that algorithmically derives global changes to the font's metrics. When this procedure is executed, 11 elements are put on the stack; the eleventh operand is the CID. See Section 5.9.2, "Changing Glyph Metrics."
GlyphDirectory	array or dictionary	*(Optional)* A mechanism for the subsetting or incremental definition of glyphs in the font; see "GlyphDirectory for Type 0 CIDFonts" on page 375.
GlyphData	string, array, or integer	*(Inserted by **StartData**)* If the CIDFont has been defined by direct execution of a CIDFont file embedded in the PostScript program, **GlyphData** is either a string or an array of strings containing the binary data section of the CIDFont. (An array of strings may be required because of the implementation limit on the length of a string; see Appendix B.) However, if the font has been loaded from a random-access file system by **findresource**, **GlyphData** is an integer whose meaning is implementation-dependent. In the latter case, the binary data section is not loaded into VM; instead, portions of it are accessed dynamically as needed during glyph rendering.

TABLE 5.14 Entries in a dictionary in FDArray

KEY	TYPE	VALUE
FontMatrix	array	*(Required)* An array defining the transformation from the glyph space for glyphs associated with this dictionary to the glyph space for the CIDFont overall.
Private	dictionary	*(Required)* A dictionary containing internal information that is shared among the glyphs associated with this dictionary; see below.
FontName	string	*(Optional)* A name for the glyph subset associated with this dictionary. This entry is for information only; it is not used by the PostScript interpreter.

At glyph rendering time, the glyph coordinate system is defined by the concatenation of the **FontMatrix** entry in the selected **FDArray** dictionary with the **FontMatrix** entry in the CIDFont dictionary. Since the **FDArray** dictionaries can contain different **FontMatrix** values, this allows the definition of glyphs that use different glyph space units in the same Type 0 CIDFont.

As noted in Table 5.12 on page 370, the **FontMatrix** entry is optional in the Type 0 CIDFont dictionary. Typically, a Type 0 CIDFont file defines **FontMatrix** only in the subsidiary dictionaries in the **FDArray**. However, the PostScript font machin-

ery requires **FontMatrix** to be present in the main CIDFont dictionary for interpreting coordinate information that applies to the CIDFont as a whole. If it is not present:

- **defineresource** inserts a **FontMatrix** entry in the CIDFont dictionary with a value of [.001 0 0 .001 0 0] (since Type 0 CIDFonts, like Type 1 fonts, are usually defined in terms of a 1000-unit glyph coordinate system).
- In each **FDArray** dictionary, **defineresource** replaces its **FontMatrix** value with one scaled by a factor of 1000.

The **Private** dictionary in the **FDArray** subdictionary of a Type 0 CIDFont serves the same purpose as a **Private** dictionary in a Type 1 font (see *Adobe Type 1 Font Format*); however, in a Type 0 CIDFont it applies only to those glyphs in the CIDFont that specify its **FDArray** index. Normally an array named **Subrs** in the **Private** dictionary contains the subroutines, but there is an alternative that is useful with CIDFonts: the subroutines can instead be contained in the binary data section of the CIDFont. In this case, the **Subrs** array is replaced by the **Private** dictionary entries listed in Table 5.15.

TABLE 5.15 Entries replacing Subrs in the Private dictionary of an FDArray dictionary

KEY	TYPE	VALUE
SubrCount	integer	*(Required if **Subrs** is absent)* The number of subroutines, which are numbered from 0 to (**SubrCount** – 1). If there are no subroutines, **SubrCount** is 0, but the other two entries must be present nonetheless.
SDBytes	integer	*(Required if **Subrs** is absent)* The length in bytes of each offset in the subroutine offset table for glyphs referencing this **FDArray** dictionary. If the length is greater than 1, the bytes within an offset are interpreted high-order byte first.
SubrMapOffset	integer	*(Required if **Subrs** is absent)* The offset in bytes to the subroutine offset table in the binary data section of the CIDFont. The offset is relative to the beginning of the binary data section. The table is indexed by subroutine number. Each entry in the table is **SDBytes** long and is interpreted as the offset to the subroutine relative to the beginning of the binary data section.

GlyphDirectory for Type 0 CIDFonts

GlyphDirectory is a mechanism for the subsetting or incremental definition of glyphs in the CIDFont. Section 5.9.4, "Subsetting and Incremental Definition of

Glyphs," explains the general motivations for incremental definition and considerations for using it. The following information is specific to Type 0 CIDFonts.

In the absence of a **GlyphDirectory** entry, the CID is used as an index into the charstring offset table as described previously, selecting an **FDArray** index and a charstring in the binary data section of the CIDFont. However, if **GlyphDirectory** is present, the CID is used as an index or a key in **GlyphDirectory**, depending on whether **GlyphDirectory** is an array or a dictionary.

If **GlyphDirectory** is an array, its length must be at least as great as the value of the **CIDCount** entry. Each array element can be either *null* (indicating an empty element) or a string. If **GlyphDirectory** is a dictionary, the keys are integers interpreted as CIDs and the values are strings. In either case, each string consists of an optional **FDArray** index followed by a charstring. The **FDArray** index is **FDBytes** long, or absent if **FDBytes** is 0. As for all CIDFonts, there must be a glyph description for CID 0.

If **GlyphDirectory** is an array, any unused entries in the array will be wasted space. An array of a given length consumes about 40 percent of the memory used by a dictionary of the same length. Thus, the dictionary representation is advisable only for a sparsely populated font containing less than 40 percent of its characters.

The binary data section of a CIDFont with a **GlyphDirectory** entry does not need a charstring offset table or charstrings; if present, they will be ignored. However, the information contained in **FDArray**, including **Private** dictionaries and subroutines used by any charstrings that will ever be defined in **GlyphDirectory**, must be supplied when the CIDFont is created. No provision is made for downloading any of this data incrementally.

Type 1 CIDFonts

Type 1 CIDFonts use PostScript procedures to construct glyphs; they are the CIDFont analog to Type 3 base fonts. However, instead of a character code or character name, they use a CID to select the glyph to build. They do not have a binary data section; they consist of PostScript code only.

In addition to the entries common to all CIDFont dictionaries (Table 5.12 on page 370), Type 1 CIDFont dictionaries contain a **BuildGlyph** entry, as shown in Table 5.16.

TABLE 5.16 Additional entry specific to Type 1 CIDFont dictionaries

KEY	TYPE	VALUE
BuildGlyph	procedure	*(Required)* A procedure that constructs the requested glyph. The CIDFont dictionary, followed by the CID for the character, is on the stack when the procedure is called.

When a PostScript program tries to show a glyph from a Type 1 CIDFont and the glyph is not already present in the font cache, the PostScript interpreter pushes the current CIDFont dictionary and the CID on the operand stack and executes the font's **BuildGlyph** procedure. The **BuildGlyph** procedure must remove these two objects from the operand stack and use this information to construct the requested glyph, following the guidelines described in Section 5.7, "Type 3 Fonts." In particular, the procedure must supply the metrics for the glyph by executing the **setcachedevice**, **setcachedevice2**, or **setcharwidth** operator.

Incremental definition is also permitted for Type 1 CIDFonts. How this is accomplished depends on how the PostScript program defines the font's **BuildGlyph** procedure. See Section 5.9.4, "Subsetting and Incremental Definition of Glyphs."

Type 2 CIDFonts

A Type 2 CIDFont is the analog of a Type 42 (TrueType) base font whose glyph descriptions are identified by CIDs rather than by character names. As in a Type 42 font, the binary font data defining the TrueType font is wrapped in PostScript language syntax to make it conform to the PostScript language font model: the font data is stored in an entry named **sfnts** in a Type 2 CIDFont dictionary.

The contents of a Type 2 CIDFont dictionary are an upward-compatible extension of a Type 42 font. This is intended to enable Type 2 CIDFonts to be used with PostScript interpreters that do not have built-in support for them. (Such use requires the assistance of a separate software package, analogous to the CID Support Library; at the time of publication, no such package has been developed by Adobe.) Two entries in a Type 42 font dictionary, **CharStrings** and **Encoding**, are required to be present in a Type 2 CIDFont dictionary, even though they are not meaningful and have no effect. Also, the **FontType** value must be 42 at the time a Type 2 CIDFont is defined; **definefont** or **defineresource** replaces its value with 11.

In addition to the entries common to all CIDFont dictionaries (Table 5.12 on page 370), Type 2 CIDFont dictionaries include the entries listed in Table 5.17.

TABLE 5.17 Additional entries specific to Type 2 CIDFont dictionaries

KEY	TYPE	VALUE
CIDCount	integer	*(Required)* The number of valid CIDs in the CIDFont. Valid CIDs range from 0 to (**CIDCount** − 1); CIDs outside this range are treated as undefined glyphs.
GDBytes	integer	*(Required)* The length in bytes of the TrueType glyph index in the **CIDMap** table. If the length is greater than 1, the bytes comprising a glyph index are interpreted high-order byte first. In most cases **GDBytes** will be 2, allowing glyph indices in the range 0 to 65,535.
CIDMap	string or array	*(Required)* A table containing the glyph index for each CID. The table must be **CIDCount** × **GDBytes** long. It may be represented as a single string or as an array of strings in which each element is a multiple of **GDBytes** long. (An array may be required because of the implementation limit on the length of a string; see Appendix B.)
sfnts	array	*(Required)* An array of one or more strings that, when concatenated, are treated as the binary representation of the TrueType font. (Multiple strings may be required because of the implementation limit on the length of a string; see Appendix B.) See also Adobe Technical Note #5012, *The Type 42 Font Format Specification.*
Encoding	array	*(Required)* Required for compatibility with Type 42 fonts. The contents of this array are ignored.
CharStrings	dictionary	*(Required)* Required for compatibility with Type 42 fonts. At a minimum, the **CharStrings** dictionary must have a .notdef entry that maps to a valid glyph index. The contents of this dictionary are otherwise ignored.
PaintType	integer	*(Optional)* A code indicating how the glyphs of the CIDFont are to be painted: 0 Glyph outlines are filled. 2 Glyph outlines (designed to be filled) are stroked. Default value: 0.
StrokeWidth	integer	*(Optional)* The stroke width (in units of the glyph coordinate system) for stroked-outline fonts (**PaintType** 2). This entry is not initially present in filled-outline CIDFonts. It should be added (with a nonzero value) when a stroked font is created from an existing filled font. Default value: 0.

Metrics	dictionary	*(Optional)* A dictionary containing the metric information (glyph widths and sidebearings) for writing mode 0. This entry is not normally present in the original definition of a CIDFont. Adding a **Metrics** dictionary to a CIDFont overrides the metric information encoded in the glyph descriptions. See Sections 5.4, "Glyph Metric Information," and 5.9.2, "Changing Glyph Metrics." In this dictionary, the keys are CIDs and the values are specified in the units defined by the **FontMatrix** entry in the top-level CIDFont dictionary.
Metrics2	dictionary	*(Optional)* Similar to **Metrics**, but for writing mode 1.
CDevProc	procedure	*(Optional)* A procedure that algorithmically derives global changes to the font's metrics. When this procedure is executed, 11 elements are put on the stack; the eleventh operand is the CID. See Section 5.9.2, "Changing Glyph Metrics."
GlyphDirectory	array or dictionary	*(Optional)* A mechanism for the subsetting or incremental addition of glyphs in the font; see "Incremental Definition of Type 42 Fonts" on page 355. **GlyphDirectory** works exactly the same in a Type 2 CIDFont as it does in a Type 42 font, except that the glyph index used to access **GlyphDirectory** is obtained from the **CIDMap** table instead of the **CharStrings** dictionary.

The **FontMatrix** value of a Type 2 CIDFont is usually defined as an identity transformation, just as it is in a Type 42 base font. See Section 5.8.2, "Type 42 Fonts (TrueType)" for a discussion of the implications of this.

Type 4 CIDFonts

A Type 4 CIDFont consists entirely of glyphs that have been prerendered as device pixel arrays (bitmaps). Unlike all other types of fonts and CIDFonts, the Type 4 CIDFont dictionary does not contain any glyph descriptions; instead, glyph bitmaps for the CIDFont are incrementally loaded directly into the font cache by explicit execution of special operators.

Note: *For correct results, the application or driver generating the PostScript page description must know certain device-dependent details of the target device, including the resolution and orientation of device space and the capacity of the font cache. Therefore, Type 4 CIDFonts should be used only for attached printer systems that are under the direct control of host software. They should not be used in a document that is intended to be portable.*

The results produced by a Type 4 CIDFont are essentially similar to those produced by a Type 3 base font or a Type 1 CIDFont whose **BuildGlyph** or **BuildChar** procedure invokes the **imagemask** operator to paint a prerendered glyph bitmap as a stencil mask. The latter technique is sometimes used by applications or drivers that choose to render glyphs on the host, either because the glyph descriptions cannot be embedded in the document or because doing the rendering on the host is faster than doing it in the PostScript interpreter. Using a Type 4 CIDFont for this purpose has the following advantages:

- A Type 4 CIDFont requires only a small amount of VM for the font dictionary, in addition to the font cache memory for the glyph bitmaps. In contrast, the equivalent Type 3 font or Type 1 CIDFont requires VM for strings containing all the glyph bitmaps; when the font's **BuildGlyph** or **BuildChar** procedure is invoked, those bitmaps are then replicated in the font cache, effectively doubling the storage required.
- Installing a glyph directly into the font cache is much more efficient than first defining it in VM and then invoking **BuildGlyph** and **imagemask** to cause the glyph to be installed in the font cache.

A Type 4 CIDFont is compatible with any PostScript interpreter that supports this feature. Nevertheless, ideal results are obtained only when the font is used with the device for which the prerendered glyphs were intended—in particular, one having the proper resolution and orientation of device space. Some usage restrictions apply to Type 4 CIDFonts:

- They should not be used within encapsulated PostScript (EPS) files or in application-generated PostScript code that drivers pass through, because there is no guarantee that the interpreter that eventually processes this code is capable of supporting these fonts or that the prerendered bitmaps will be appropriate for the device.
- Since they do not contain glyph descriptions, they cannot be used in any operation that requires obtaining the glyph outline as a path, such as clipping. The **charpath** operator has no effect when used with a Type 4 CIDFont.
- They do not support any of the effects that can be produced in most types of fonts and CIDFonts by modifying entries such as **PaintType**, **StrokeWidth**, **Metrics**, **Metrics2**, and **CDevProc**.

A Type 4 CIDFont dictionary can contain the entries listed in Table 5.12 on page 370. Note the following:

- The **CIDFontName** and **CIDSystemInfo** entries, which are required in other types of CIDFonts, are optional in a Type 4 CIDFont. (Typically, a Type 4 CIDFont does not contain a standard character collection but is used with a special-purpose, custom CMap.)
- **FontMatrix** must be the inverse of the transformation from default user space to device space of the device for which the bitmap is designed to be used. Initially, the translation components of the font matrix should be 0 (although a subsequent **makefont** operation may impose a translation). Defining the font matrix this way ensures that when glyphs are shown at their intended size and orientation, the glyph coordinate system will be the same as the device coordinate system except for translation.

Operators for Type 4 CIDFonts

Once a Type 4 CIDFont has been defined by invoking **defineresource**, the following three operators may be used to manage the CIDFont's glyph bitmaps in the font cache. Because of their specialized use, these operators are defined in the **BitmapFontInit** procedure set rather than in **systemdict**.

- **addglyph** loads the glyph bitmap (and metrics) associated with a CID in a Type 4 CIDFont.
- **removeglyphs** removes the bitmaps for specified CIDs associated with a Type 4 CIDFont.
- **removeall** removes all bitmaps associated with a Type 4 CIDFont.

Unlike most glyphs in the font cache, glyphs loaded by **addglyph** cannot be removed automatically to make room for other glyphs (except in the case that the Type 4 CIDFont itself is removed from VM). The PostScript program must manage the font cache properly to avoid exhausting it. As indicated earlier, this requires it to know device-dependent details about the memory consumed by cached glyphs and about the total capacity of the font cache.

The **removeglyphs** and **removeall** operators logically remove glyphs from the font cache at the moment the operators are executed; the glyphs are no longer accessible during subsequent **show** operations. However, glyphs that have been re-

moved may continue to occupy font cache memory until all pages on which the glyphs were used have been produced. Such glyphs compete with new glyphs for font cache memory; their consumption is reflected in the value of the **CurFontCache** system parameter. As a general rule, if glyphs from any Type 4 CIDFonts are used on a page, all the glyphs for the page must fit in the font cache; otherwise, a **limitcheck** error may occur.

When characters are shown from a Type 4 CIDFont, the font machinery first performs its usual action of checking the font cache to see if the glyphs are already present. If a glyph has previously been loaded by **addglyph** and the concatenation of the **FontMatrix** entry and the CTM is the identity matrix (except for translation), the cached glyph is transferred directly to the current page. This is the normal behavior that occurs when a Type 4 CIDFont is used as intended.

If a character being shown is not present in the font cache, one of two exceptional conditions has arisen:

- No glyph for the requested CID has been loaded into the font cache by **addglyph**. This case is handled by performing a glyph substitution, just as in any other type of CIDFont; see "Handling Undefined Characters" on page 389.
- The glyph exists in the font cache, but the concatenation of the **FontMatrix** entry and the CTM is not the identity matrix (because one or the other has been scaled or otherwise transformed). In this case, the glyph bitmap is treated as if it were image source data and is painted as if by the **imagemask** operator, using the transformed coordinate system.

Note: *Such a transformation degrades the quality of prerendered glyphs for Type 4 CIDFonts in the same way as for Type 3 base fonts or Type 1 CIDFonts that describe glyphs as bitmaps.*

5.11.4 CMap Dictionaries

As stated earlier, a CMap dictionary specifies the mapping from character codes to character selectors (CIDs, character names, or character codes) in one or more associated fonts or CIDFonts. The CMap does not refer directly to specific fonts or CIDFonts; instead, it is combined with them as part of a Type 0 font whose **FMapType** value is 9.

Within the CMap, the character mappings refer to the associated fonts or CIDFonts by *font number*, indexing from 0. All of the mappings for a particular font number must specify the same kind of character selector. If the character selectors are CIDs, the associated dictionary is expected to be a CIDFont. If the character selectors are names or codes, the associated dictionary is expected to be a font.

A CMap dictionary is created with the assistance of operators defined in the **CIDInit** procedure set, described below. It includes the entries listed in Table 5.18.

TABLE 5.18 Entries in a CMap dictionary

KEY	TYPE	VALUE
CMapType	integer	*(Required)* The CMap type, indicating how the CMap is organized. The only defined values are 0 and 1, which are equivalent; 1 is recommended.
CMapName	name	*(Required)* The name of the CMap. Ordinarily, it is the same as the key given to **defineresource** when defining the dictionary as an instance of the **CMap** resource category.
CIDSystemInfo	dictionary or array	*(Required)* A dictionary or array identifying the character collection for each associated font or CIDFont. If the CMap selects only font 0 and specifies character selectors that are CIDs, this entry can be a dictionary identifying the character collection for the associated CIDFont. Otherwise, it is an array indexed by the font number. If the character selectors for a given font number are CIDs, the corresponding entry is a dictionary identifying the character collection for the associated CIDFont. If the character selectors are names or codes (to be used with an associated font, not a CIDFont), the entry should be *null*. For details of the **CIDSystemInfo** dictionaries, see Section 5.11.2, "CIDSystemInfo Dictionaries."
CodeMap	varies	*(Inserted by the CMap construction operators)* The information used to map from character codes to a font number and character selector. This is an internal representation of the information presented to the CMap construction operators (defined in the **CIDInit** procedure set) executed during the definition of the CMap. The contents and format are implementation-dependent.
CMapVersion	number	*(Optional)* The version number of this CMap. This entry is for information only; it is not used by the PostScript interpreter.
XUID	array	*(Optional)* An array of integers that uniquely identifies the CMap or any variant of it. See Section 5.6.2, "Extended Unique ID Numbers."

UIDOffset	integer	*(Optional)* An integer used in combination with the **UIDBase** entry in a CIDFont to form **UniqueID** entries for the base fonts within a composite font. This entry is used only in PostScript interpreters lacking built-in CID-keyed font support but having other means, such as the CID Support Library (CSL), to achieve similar capabilities.
WMode	integer	*(Optional)* The value to be used for the **WMode** entry in any font constructed by the **composefont** operator using this CMap; see Section 5.4, "Glyph Metric Information." Default value: 0.

CMap Operators in the CIDInit Procedure Set

The operators needed to construct a CMap dictionary are contained in the **CIDInit** procedure set (an instance of the **ProcSet** resource category); this includes the operators needed in a rearranged font's CMap to derive the rearranged font from an existing CID-keyed font. The following is a summary of the CMap construction operators. The use of these operators to construct a CMap for a CID-keyed font is discussed in Section 5.11.5, "FMapType 9 Composite Fonts." Chapter 8 gives the syntax and brief descriptions of all the operators in **CIDInit**; for complete documentation, see Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*.

- **begincmap** and **endcmap** enclose the CMap definition.
- **usecmap** incorporates the code mappings from another **CMap** resource instance.
- **beginrearrangedfont** and **endrearrangedfont**, in the CMap for a rearranged CID-keyed font, specify an array of component fonts for the rearranged font.
- **begincodespacerange** and **endcodespacerange** define *codespace ranges*—the valid input character code ranges—by specifying a pair of codes of some particular length giving the lower and upper bounds of each range; see "CMap Mapping" on page 388.
- **usefont** specifies a font number that is an implicit operand of all the character code mapping operations that follow.
- **beginbfchar** and **endbfchar** define mappings of individual input character codes to character codes or character names in the associated font. **beginbfrange** and **endbfrange** do the same, but for ranges of input codes.

- **begincidchar** and **endcidchar** define mappings of individual input character codes to CIDs in the associated CIDFont. **begincidrange** and **endcidrange** do the same, but for ranges of input codes.
- **beginnotdefchar, endnotdefchar**, **beginnotdefrange,** and **endnotdefrange** define notdef mappings from character codes to CIDs. As described in the section "Handling Undefined Characters" on page 389, a notdef mapping is used if the normal mapping produces a CID for which no glyph is present in the associated CIDFont.
- **beginusematrix** and **endusematrix** define a transformation matrix to be applied to an associated font or CIDFont.

In addition to the operators listed above, the **CIDInit** procedure set also includes the **StartData** operator, which is used in defining Type 0 CIDFonts (as described in the section "Type 0 CIDFonts" on page 371). Other entries in the **CIDInit** procedure set are private and should not be accessed by a PostScript program.

Note: *Although the contents of the* ***CIDInit*** *procedure set are documented as "operators," they may actually be implemented as procedures rather than as operator objects. Since they are always directly executed by name, there is no difference in behavior between procedures and operators.*

CMap Example

The following example (with paragraphs of comments interspersed) creates a simple CMap dictionary. It demonstrates the use of several of the operators in the **CIDInit** procedure set to define mappings from character codes to a character selector and font number. The example is contrived and is not entirely explained by the comments; its purpose is to provide background for subsequent descriptions.

```
/CIDInit /ProcSet findresource begin
    8 dict begin
        begincmap
```

The CMap definition begins here. It maps character codes to a CIDFont and a base font; therefore, the **CIDSystemInfo** entry must be an array with two elements. Font 0 is a CIDFont that references the Adobe-Japan1-1 character collection, and font 1 is a base font (represented by the *null* element of the array).

```
/CIDSystemInfo
   [   3 dict dup begin
          /Registry (Adobe) def
          /Ordering (Japan1) def
          /Supplement 1 def
       end
       null
   ] def
```

The following operators fill in some other CMap entries.

```
/CMapName /ExampleCMap def
/CMapVersion 1 def
/CMapType 1 def
/XUID [1000000  10] def
/WMode 0 def
```

The mapping information from here on gets stored in the **CodeMap** entry by the CMap construction operators. First, four valid codespace ranges are defined for a mixed single-byte and double-byte encoding.

```
4 begincodespacerange
   <00>     <80>
   <8140>   <9FFC>
   <A0>     <DF>
   <E040>   <FCFC>
endcodespacerange
```

Next come the mappings for font 0 (as specified by the **usefont** operator below). The **…cidrange** operators define a mapping from four character code ranges to CIDs, with each range mapping to consecutive CIDs starting at a specified number.

```
0 usefont
4 begincidrange
   <20>     <7e>     231
   <8140>   <817e>   633
   <8180>   <81ac>   696
   <8940>   <897e>   1219
endcidrange
```

The following operators apply a transformation to font 1 that rotates the glyphs counterclockwise by 90 degrees.

```
1 usefont
1 beginusematrix
   [0 1 –1 0 0 0]
endusematrix
```

The mappings for font 1 follow. Here each character code range is mapped to another range of consecutive codes or to a list of character names in a font.

```
2 beginbfrange
   <C1>    <C3>    <63>
   <A1>    <A3>    [/A /B /C]
endbfrange
```

Codes can also be mapped one code at a time.

```
3 beginbfchar
   <CA>    /j
   <CB>    <6B>
   <CC>    <6C>
endbfchar
```

Finally, the CMap definition is ended and the CMap resource instance is defined.

```
      endcmap
      currentdict CMapName exch /CMap defineresource pop
   end        % CMap dictionary
end           % CIDInit ProcSet
```

5.11.5 FMapType 9 Composite Fonts

This section explains how to combine a CMap with one or more fonts or CIDFonts to produce a CID-keyed font, and it describes the behavior of such fonts during **show** operations.

As indicated earlier, a CID-keyed font is represented as a Type 0 font whose **FMapType** value is 9. It contains a **CMap** entry whose value is a CMap dictionary. Its **FDepVector** array references the CIDFont, base font, or composite font dictionaries with which the CMap has been combined.

A PostScript program can directly create a Type 0 font having this structure. As a convenience, the **composefont** operator performs most of the required work. The following example combines the CMap created in the preceding section with a CIDFont, Ryumin-Light, and a base font, Times-Roman (which must previously exist as instances of the **CIDFont** and **Font** resource categories, respectively). It defines a composite font named ExampleFont in the **Font** category.

```
/ExampleFont /ExampleCMap [/Ryumin-Light /Times-Roman] composefont pop
```

CMap Mapping

Type 0 fonts with **FMapType** 9 require a **CMap** entry in the font dictionary. The mapping algorithm for these fonts starts out by using the information in the **CodeMap** entry of the **CMap** dictionary to decode bytes from the **show** string. This information is derived from the operands of the CMap construction operators that were invoked while the CMap was being defined; see Section 5.11.4, "CMap Dictionaries."

The number of bytes extracted from the **show** string for each successive character is determined exclusively by the codespace ranges in the CMap (delimited by **begincodespacerange** and **endcodespacerange**). A codespace range is specified by a pair of codes of some particular length giving the lower and upper bounds of that range. A code is considered to match the range if it is the same length as the bounding codes and the value of each of its bytes lies between the corresponding bytes of the lower and upper bounds. The code length cannot exceed the number of bytes representable in an integer (see Appendix B).

A sequence of one or more bytes is extracted from the **show** string and matched against the codespace ranges in the CMap. That is, the first byte is matched against one-byte codespace ranges; if no match is found, a second byte is extracted, and the two-byte code is matched against two-byte codespace ranges. This continues for successively longer codes until a match is found or all codespace ranges have been tested. There will be at most one match, since codespace ranges do not overlap.

The code extracted from the **show** string is then looked up in the character code mappings for codes of that length. (These are the mappings defined by **beginbfchar**, **endbfchar**, **begincidchar**, **endcidchar**, and corresponding operators for ranges.) Failing that, it is looked up in the notdef mappings, as described in the next section.

The results of the CMap mapping algorithm are a font number and a character selector. Character selection from the descendant font depends on whether the descendant is a base font, composite font, or CIDFont and whether the character selector is a character name, a character code, or a CID, as described below (and illustrated in Figure 5.10 on page 367).

- If the character selector is a character name, the descendant must be a base font. The equivalent of a **glyphshow** operation is performed on the name, using the descendant font as the current font. If the descendant font contains no glyph for that name, the .notdef character is used instead.
- If the character selector is a character code, the descendant may be either a base font or another composite font, but not a CIDFont. In the case of a base font, the code must be only one byte long; the **Encoding** entry in the base font is consulted in the usual manner to determine which glyph to render. In the case of a composite font, the code may be multiple bytes long; it is interpreted according to the mapping algorithm specified by the **FMapType** value for that font. During this interpretation, no additional bytes will be consumed from the **show** string.
- If the character selector is a CID, the descendant must be a CIDFont (except in a special case described below). The equivalent of a **glyphshow** operation is performed on the CID, using the descendant CIDFont as the current font. If the CIDFont contains no glyph for that CID, the notdef mappings are consulted, as described in the next section.

  Under special conditions, a CID can be used when the descendant is a Type 3 base font. The font's **BuildGlyph** or **BuildChar** procedure is invoked to render a character whose code is the last byte originally extracted from the **show** string. If this procedure executes **setfont** to establish a CIDFont as the current font and then executes a **show** operation on a string consisting of just that character code, the code is ignored; instead, the CID determined by the earlier CMap mapping is used to look up the glyph in the CIDFont. This special case exists for compatibility with applications that substitute Type 3 fonts for base fonts in a composite font hierarchy to achieve certain special effects.

Handling Undefined Characters

A CMap mapping operation can fail to select a glyph for any of a variety of reasons. This section describes what happens when that occurs.

If a code maps to a character selector that is a CID, but there is no such glyph in the descendant CIDFont, the *notdef mappings* in the CMap are consulted to obtain a substitute character selector. These mappings (so called by analogy with the .notdef character mechanism in base fonts) are delimited by **beginnotdefchar**, **endnotdefchar**, **beginnotdefrange**, and **endnotdefrange**; they always map to a CID. If a matching notdef mapping is found, the CID selects a glyph in the associated descendant, which must be a CIDFont. (Note that the font number in the notdef mapping can be different from the font number in the original character mapping.) If there is no glyph for that CID, the glyph for CID 0 (which is required to be present) is substituted.

Note: *For Type 1 CIDFonts, the* ***BuildGlyph*** *procedure must handle a missing glyph by rendering the glyph for CID 0. The* ***BuildGlyph*** *procedure cannot consult the* ***CMap*** *dictionary again.*

If the CMap does not contain either a character mapping or a notdef mapping for the code, font 0 is selected and a glyph is substituted from the associated font or CIDFont. If it is a base font, the character name .notdef is used; if it is a CIDFont, CID 0 is used. If it is a composite font, the behavior is implementation-dependent.

If the code is invalid—that is, the bytes extracted from the **show** string do not match any codespace range in the CMap—a substitute glyph is chosen as just described. The character mapping algorithm is reset to its original position in the **show** string, and a modified mapping algorithm chooses the best partially matching codespace range, as follows:

1. If the first byte extracted from the **show** string does not match the first byte of any codespace range, the range having the shortest codes is chosen.
2. Otherwise (that is, if there is a partial match), then for each additional byte extracted, the code accumulated so far is matched against the beginnings of all longer codespace ranges, until the longest such partial match has been found. If multiple codespace ranges have partial matches of the same length, the one having the shortest codes is chosen.

The length of the codes in the chosen codespace range determines the total number of bytes to consume from the **show** string for the current mapping operation.

CHAPTER 6

Device Control

THIS CHAPTER DESCRIBES the PostScript language's facilities for configuring a *page device*: a raster output device capable of realizing PostScript page descriptions on a physical medium. The term *medium* (plural *media*) refers to the physical material on which the device generates its results, such as paper, film, transparency material, or a virtual page on a display. Most of the processing options discussed in this chapter are oriented toward printers that produce output on paper, so paper is a good universal material to envision when you read the term *medium*.

Note: *A page device is only one of several kinds of raster output device. Other kinds include the* cache device *to put characters into the font cache and the* null device *to discard output entirely. These are set, usually temporarily, by the* ***setcachedevice*** *and* ***nulldevice*** *operators.*

The state of any device, including a page device, is represented as an internal object that is an element of the graphics state. Each execution of a device setup operator, such as **setpagedevice** (described below), **setcachedevice**, or **nulldevice**, creates a new instance of an internal device object (referred to hereafter simply as a "device"). Multiple devices can refer to the same physical resource, such as a printing engine, perhaps with different values of configuration options such as page sizes or feature settings.

Only one device—the current device in the graphics state—is active at any given time. However, there can be multiple inactive devices belonging to copies of the graphics state that have been saved by **save**, **gsave**, **gstate**, or **currentgstate**. An inactive device can be reactivated when a saved graphics state is reinstated with **restore**, **grestore**, **grestoreall**, or **setgstate**. When a device is reactivated, it brings its device parameters with it. (In an interpreter that supports multiple execution contexts, each context can have an independent active device.)

Configuring a page device includes:

- Selecting the proper medium
- Establishing a default transformation matrix from user space to device space, along with other device-dependent rendering parameters for producing output on the medium
- Selecting processing options such as multiple copies, or special features of the output device such as duplex (two-sided) printing

Once a device has been configured, a PostScript program can proceed to describe a sequence of pages. The program paints the contents of each page in turn in raster memory, with everything that is to appear on that page: text, graphics, and sampled images. It then invokes the **showpage** operator to cause the page to be rendered on the physical output device. **showpage** transmits the contents of raster memory to the device, then erases the page and partially resets the graphics state in preparation for the next page. (Another operator, **copypage**, is similar to **showpage**, but its behavior varies by LanguageLevel and its use is not recommended.)

The facilities for configuring a page device differ by LanguageLevel:

- LanguageLevel 1 provides a collection of device control operators, defined in a special dictionary named **statusdict**. The contents of **statusdict** are device-dependent (although an attempt has been made to maintain a consistent specification for features common to multiple devices). They are not described in this book, but rather in the *PostScript Language Reference Supplement*. Application programs wishing to use **statusdict** features can extract information from *PostScript printer description* (PPD) files; for the format of these files, see Adobe Technical Note #5003, *PostScript Printer Description File Format Specification.*
- LanguageLevels 2 and 3 support a page device setup operator named **setpagedevice**. This operator provides a standard framework for specifying the requirements of a page description and for controlling both standard and optional features of a page device.
- LanguageLevel 3 includes an optional feature, *in-RIP trapping*, whose purpose is to compensate for misregistration between colorants. This feature is supported primarily in devices (such as imagesetters) that are used in the production of plates for printing presses. It is controlled partially by **setpagedevice**

and partially by two special-purpose operators, **settrapzone** and **settrapparams**, defined in the **Trapping** procedure set.

The remainder of this chapter describes the page device configuration facilities of LanguageLevels 2 and 3. Features that are specifically part of LanguageLevel 3 are so indicated, though many such features have historically originated as extensions to LanguageLevel 2 and are available in some LanguageLevel 2 implementations (see Appendix A). Furthermore, as discussed below, not all page device features are supported on all devices.

6.1 Using Page Devices

Many output devices have special hardware features, such as multiple paper trays with different sizes of paper, duplex (two-sided) printing, collation, finishing options, and so forth. Not all such features are available on all devices, however. The PostScript interpreter knows what features a device supports and which are currently available and ready for use.

A document's device configuration requirements may limit the set of devices on which the document can be printed. Specifying such configuration information within a PostScript page description allows the interpreter to match the document's requirements with the features and options available on a given output device. The **setpagedevice** operator provides a uniform framework for specifying configuration requirements and options. It uses a standard format to request features supported by all devices (such as selecting a page size) and those supported only by some devices (such as duplex printing). In addition, **setpagedevice** allows a user or system administrator to establish default device configuration parameters to be used when a page description does not specify them explicitly. It also provides a standard mechanism for determining what to do when a page description makes feature requests that the device cannot fulfill.

It is useful, at least in concept, to envision two separate tasks when printing from an application:

1. Generate a device-independent page description.
2. Request that the page description be rendered on a particular device. At this point, the user should have an opportunity to add processing options, including device-dependent ones, to the page description.

Even if a single application performs both of these functions, it is best to maintain the distinction. Many applications have an option to store the generated page description in a file for later use. Such a file should not contain unnecessary device-dependent processing options. The distinction between document generation and document rendering is essential when using PostScript programs for document storage and interchange.

While there is no clear division between device-independent processing requests and device-dependent ones, you should keep in mind the important goal of producing device-independent page descriptions. One important criterion to apply is whether a particular feature is inherently part of the document specification itself or merely a processing option. For example, the page size—in particular, the aspect ratio between width and height—is an important part of the document specification, because the application generating the document must make formatting decisions based on it. On the other hand, the number of copies to be printed or the color of the paper to be used is not an inherent part of the document description, but rather a processing option.

6.1.1 Page Device Dictionary

The current internal state of a page device is modeled by a *page device dictionary* containing entries called *page device parameters*. The keys in this dictionary represent device features or processing options; the associated values represent the current settings of those features or options. The page device dictionary is not directly accessible to a PostScript program, but its contents can be read and altered by the **currentpagedevice** and **setpagedevice** operators.

***Note: setpagedevice** is a page-oriented operator used to control the output processing of one or more pages of a page description. Any call to **setpagedevice** implicitly invokes **erasepage** and **initgraphics**, and thus must precede the descriptions of the pages to be affected. If the current device is not a page device, the effect of invoking **setpagedevice** is device-dependent.*

The operand to **setpagedevice** is a *request dictionary* whose entries specify desired settings or values for one or more page device parameters. The request dictionary is simply a container that can hold multiple parameter requests to be issued in a single call to **setpagedevice**. The interpreter uses the contents of the request dictionary to alter the state of the page device parameters, but it does not retain the request dictionary itself. Note that the entries in the request dictionary are merely *requests* for certain parameter settings; depending on the capabilities

of a given page device, these requests may or may not be honored. The **currentpagedevice** operator returns a dictionary whose entries reflect the actual current settings of the page device parameters, not necessarily those that have been requested via **setpagedevice**.

Because **setpagedevice** merges new parameter settings into the existing state of the device, its effects are cumulative over multiple executions. That is, the effect of setting a particular page device parameter persists through subsequent invocations of **setpagedevice** until explicitly overridden or until the device is restored to some previous state by a **restore**, **grestore**, **grestoreall**, or **setgstate** operation. This cumulative behavior also applies recursively (to one level) to the contents of subsidiary dictionaries that are the values of the page device parameters **Policies**, **InputAttributes**, and **OutputAttributes** (all of which are described in detail later in this chapter). It does not apply to the contents of other entries whose values happen to be dictionaries (except possibly to the contents of *details dictionaries* passed to **setpagedevice**; see Section 6.1.2, "Details Dictionaries").

Because the effects of **setpagedevice** are cumulative, a PostScript program can make multiple calls to **setpagedevice**, each requesting particular parameter settings but leaving the settings of other parameters undisturbed. This allows different features or options to be specified independently. In particular:

- When an application generates a page description, it can include a call to **setpagedevice** specifying parameters that reflect assumptions the application has made in formatting the document, such as the page size and bounding box.
- When a user requests printing, an additional call to **setpagedevice** can be prepended to the page description to specify print-time options such as two-sided printing or the number of copies.
- The person operating the device can invoke **setpagedevice**, as part of an unencapsulated job, to specify the available media, establish recovery policies for dealing with unsatisfied requests, and establish default values for other device options. (Jobs and encapsulation are discussed in Section 3.7.7, "Job Execution Environment.")

For certain parameters, there is a null value that indicates the absence of any specific request or preference. In all cases, the null object (that is, the value of **null** in **systemdict**) is used for this purpose. For example, a null value for the **MediaColor** parameter indicates that no specific paper color has been requested. Null values

are permitted only for certain features, as indicated in the relevant table entries in Section 6.2, "Page Device Parameters."

Omitting a parameter key from the request dictionary has a different meaning than including the key with a null value. Omitting the key leaves the parameter's previous value unchanged; specifying a null value sets it to the null object, canceling any previous value it may have had. The dictionary returned by **currentpagedevice** always contains an entry for every parameter supported by the device, though the value for some parameters may be null (indicating that the feature is supported but no setting has yet been requested for it).

Note: *The PostScript language does not prescribe a default value for any page device parameter. The usual default value for optional features is either* false *or* null, *but this is not invariably the case for all devices. A PostScript program can change the default values by invoking* ***setpagedevice*** *as part of an unencapsulated job.*

If the request dictionary requests parameter settings that the device cannot satisfy (for example, if a program requests duplex printing on a device that does not support it), the PostScript interpreter invokes a uniform *recovery policy* for determining what to do. This policy may vary depending on the specific page device parameter involved. For most parameters, there are three possibilities:

- Ignore the request (for example, print simplex on a device that cannot honor a request for duplex printing).
- Generate a **configurationerror** (reject the job).
- Notify the human operator or the print management software to determine what to do.

The choice is based on information in the **Policies** subdictionary of the page device dictionary, which in turn can be altered by **setpagedevice**; see Section 6.2.7, "Unsatisfied Parameter Requests," for further details.

Note: *In the descriptions of individual page device parameters in this chapter, the statement "a configuration error will occur" actually means that the PostScript interpreter will take one of the three possible actions listed above, according to specified policy. Thus a "configuration error" is not necessarily the same thing as a PostScript* ***configurationerror****.*

If a device does not support a particular feature, **setpagedevice** will not recognize any request to specify a value for the corresponding parameter. For example, if a device does not have a duplexing mechanism, **setpagedevice** will not recognize the parameter key **Duplex**—even if the request is to set the value of **Duplex** to *false*, indicating no duplexing. Instead, **setpagedevice** will consult policy to determine what to do. (Note that this means a print job cannot expand the set of keys in a device's page device dictionary, though it *can* add new keys to the **Policies** dictionary.) This behavior may seem surprising, but it is necessitated by the fact that the set of device features is open-ended.

6.1.2 Details Dictionaries

The operation of some page device features depends on the values of variables (which may be quite different for different devices) that determine precisely how the feature functions. Features of this type are generally controlled by two separate page device parameters: a boolean parameter that enables or disables the feature as a whole and a *details dictionary* containing the variables that determine its precise behavior. This allows an application that is not knowledgeable about the feature to use it in a straightforward way, while more sophisticated applications can exercise greater control over the details of its operation.

Details dictionaries follow a consistent naming convention: the name of the dictionary is taken from the name of the page device parameter that enables or disables it, with the word **Details** appended. For example, the feature enabled by the **Trapping** parameter is controlled by a details dictionary named **TrappingDetails**.

Since the details of a particular feature may not be the same on all devices, every details dictionary must contain a **Type** entry indicating the overall composition of the dictionary. The value of this entry is an integer that determines how the dictionary's entries are organized and interpreted (as well as whether changes to its contents are cumulative, as discussed in Section 6.1.1, "Page Device Dictionary"). When a request dictionary passed to **setpagedevice** includes a details dictionary as the value of one of its entries, a configuration error will occur if the device does not support the **Type** value specified in the details dictionary.

6.2 Page Device Parameters

This section describes specific page device parameters that have been defined at the time of publication of this book. Table 6.1 classifies them into general categories for easier understanding. (This classification is not rigid, however; parameters in different categories can sometimes interact with each other.) In the future, other parameters may be defined as needed to cover new processing options or device features. Once defined for any device, a given parameter name will always be used for the same feature in any subsequent devices that support it.

Note: *Excluded from the table (and from this chapter) are parameters that are device-specific, controlling features unique to a particular device or limited to only a few devices; for information on these, see the PostScript Language Reference Supplement and individual product documentation. Note also that not all parameters described here are supported by all devices; consult the product documentation for each device to see exactly which parameters it supports.*

6.2.1 Media Selection

A given output device may support one or more physical sources for the media on which PostScript page descriptions are rendered. These *media sources* (often called *trays* or *positions*) are arbitrarily numbered by small integer *position numbers*. A position number usually designates a specific physical location in the hardware, though it may refer instead to some logical capability such as a pair of trays that contain the same medium and are used alternately. The correspondence between position numbers and media sources is device-dependent; it is not described in this book, but rather in individual product documentation.

The page device parameters shown in Table 6.2 control the selection of an input media source. Some of these parameters (**PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, **InsertSheet**) are used by a page description to specify its media requirements. The **InputAttributes** parameter describes the properties of the physical media currently available on the device. The **setpagedevice** operator uses this information to match a page's requirements with the available media and decide which physical media source to use for that page. The remaining parameters in Table 6.2 specify additional requirements for input media handling that may also influence the media selection process.

TABLE 6.1 Categories of page device parameters

CATEGORY	DESCRIPTION	PARAMETERS	
Media selection	Select the appropriate type of paper or other physical medium.	**DeferredMediaSelection** **InputAttributes** **InsertSheet** **LeadingEdge** **ManualFeed** **MediaClass**	**MediaColor** **MediaPosition** **MediaType** **MediaWeight** **PageSize** **TraySwitch**
Roll-fed media	Provide additional information pertaining to media that are fed from a continuous roll, such as film in an imagesetter.	**AdvanceDistance** **AdvanceMedia** **CutMedia**	**Orientation** **RollFedMedia**
Page image placement	Specify how page images are to be rendered onto the physical medium.	**Duplex** **HWResolution** **ImageShift** **ImagingBBox** **Margins**	**MirrorPrint** **NegativePrint** **PageOffset** **Tumble**
Page delivery	Control the disposition of physical output.	**Collate** **Jog** **NumCopies** **OutputAttributes**	**OutputDevice** **OutputFaceUp** **OutputType**
Color support	Specify various aspects of color output on devices capable of producing it.	**MaxSeparations** **PageDeviceName** **ProcessColorModel** **SeparationColorNames** **SeparationOrder**	**Separations** **Trapping** **TrappingDetails** **UseCIEColor**
Device initialization and page setup	Define special actions to be performed when the device is installed and before and after each page is printed.	**BeginPage** **EndPage**	**Install**
Unsatisfied parameter requests	Specifies recovery policies for handling requests for unsupported features.	**Policies**	

Media selection is performed according to one of two models:

- *Immediate media selection*. All information about the available media is present in the **InputAttributes** and other page device parameters at the time **setpagedevice** is invoked. **setpagedevice** uses this information, along with the policies established for handling unsatisfied parameter requests, to select a media source. The outcome of this decision is immediately visible to the PostScript program.
- *Deferred media selection*. Information about the available media is not known at the time **setpagedevice** is invoked. Instead of selecting the media source immediately, **setpagedevice** merely collects the selection requests and saves them for use at some later time when the page image is actually applied to the medium.

Most of this section is devoted to describing the immediate media selection process in detail. Deferred media selection is discussed in "Deferred Media Selection" on page 411.

TABLE 6.2 Page device parameters related to media selection

KEY	TYPE	VALUE
InputAttributes	dictionary or null	A dictionary specifying the attributes of all input media currently available for use by this output device. The dictionary contains an entry for each available media source on the device—for example, each input paper tray on a printer. The key for each entry is an arbitrary integer position number; the value is a subdictionary describing the medium currently available from that source. Entries in these subdictionaries include **PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, and **InsertSheet**, with the same meanings as the corresponding page device parameters described in this table. Two other optional entries, **Priority** and **MatchAll**, control details of the matching algorithm; see "Matching Requests with Attributes" on page 403. Changes to the contents of the **InputAttributes** dictionary are cumulative; that is, the **setpagedevice** operator merges the contents of **InputAttributes** from the request dictionary with those of the existing **InputAttributes** dictionary for the current device. However, this cumulative merging is not recursive: the contents of subdictionaries representing individual media sources within the **InputAttributes** dictionary are replaced outright rather than merged.

On devices that perform deferred media selection (see "Deferred Media Selection" on page 411), the PostScript interpreter has no prior knowledge of the available input media. In such cases, **InputAttributes** is either an empty dictionary or *null*. Instead of attempting to fulfill the PostScript program's input media requests, the **setpagedevice** operator will then simply save them and pass them to the device's printing subsystem along with the contents of the rendered page image.

PageSize array

An array of two numbers, [*width height*], specifying the overall dimensions of the physical medium that were assumed during the generation of this page description. The dimensions are expressed in units of the default user space (72nds of an inch) and include any unprinted borders along the edges of the page; thus the lower-left and upper-right corner of the assumed physical page are at user space coordinates (0, 0) and (*width*, *height*), respectively.

The specified page dimensions are considered to match those of an available input medium if they are within a tolerance of 5 units in each dimension. (For roll-fed media, the tolerance applies only to the amount by which the requested size may *exceed* the actual size of the physical medium; it may be smaller by an unlimited amount.) The order in which the dimensions are specified is immaterial; that is, a requested page size of [*a b*] is considered to match an input medium whose dimensions are specified as [*b a*]. Likewise, the physical orientation of the medium in the printing mechanism is unspecified and can vary from one device to another. The PostScript interpreter will set up the transformation from user space to device space so that the longer and shorter dimensions specified by **PageSize** are properly oriented with those of the physical medium.

To allow pages of the same dimensions in portrait (height greater than width) and landscape (width greater than height) orientations to be interspersed on the same physical medium, the **setpagedevice** operator rotates the default user space for landscape orientation 90 degrees counterclockwise with respect to that for portrait orientation. This relationship holds only for pages rendered on the same physical medium; no such relationship is guaranteed between different media. For roll-fed media, the page orientation is further determined by the **Orientation** page device parameter (see Table 6.3 on page 412).

MediaColor string or null

A string identifying the color of the medium.

MediaWeight number or null

The weight of the medium in grams per square meter. "Basis weight" or "ream weight" in pounds can be converted to grams per square meter by multiplying by 3.76; for example, 10-pound paper is approximately 37.6 grams per square meter.

MediaType string or null — An arbitrary string representing special attributes of the medium other than its size, color, and weight. This parameter can be used to identify special media such as envelopes, letterheads, or preprinted forms.

MediaClass string or null — *(LanguageLevel 3)* An arbitrary string representing attributes of the medium that may require special action by the output device, such as the selection of a color rendering dictionary. Devices should use the value of this parameter to trigger such media-related actions, reserving the **MediaType** parameter (above) for generic attributes requiring no device-specific action.

The **MediaClass** entry in the output device dictionary defines the allowable values for this parameter on a given device (see Section 6.4, "Output Device Dictionary"); attempting to set it to an unsupported value will cause a configuration error.

InsertSheet boolean — *(LanguageLevel 3)* A flag specifying whether to insert a sheet of some special medium directly into the output document. Media coming from a source for which this attribute is *true* are sent directly to the output bin without passing through the device's usual imaging mechanism (such as the fuser assembly on a laser printer). Consequently, nothing painted on the current page is actually imaged on the inserted medium. See "Special Media Handling" on page 407.

LeadingEdge integer or null — *(LanguageLevel 3)* A code specifying the edge of the input medium that will enter the printing engine or imager first and across which data will be imaged. Values reflect positions relative to a canonical page in portrait orientation (width smaller than height):

null	No request for media orientation
0	Short edge; top of canonical page
1	Long edge; right side of canonical page
2	Short edge; bottom of canonical page
3	Long edge; left side of canonical page

When duplex printing is enabled, the canonical page orientation refers only to the front (recto) side of the medium; the orientation of the back (verso) side depends on the **Tumble** parameter (see Section 6.2.3, "Page Image Placement") and is independent of the value of **LeadingEdge**.

ManualFeed boolean — A flag indicating whether the input medium is to be fed manually by a human operator (*true*) or automatically (*false*). A *true* value asserts that the human operator will manually feed media conforming to the specified attributes (**PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, and **InsertSheet**). Thus, those attributes are not used to select from available media sources in the normal way, although their values may be presented to the human operator as an aid in selecting the correct medium. On devices

		that offer more than one manual feeding mechanism, the attributes may select among them.
TraySwitch	boolean	*(LanguageLevel 3)* A flag specifying whether the output device supports automatic switching of media sources. When the originally selected source runs out of medium, some devices with multiple media sources can switch automatically, without human intervention, to an alternate source with the same attributes (such as **PageSize** and **MediaColor**) as the original. The choice of such an alternate media source is device-specific, and may or may not be influenced by the **Priority** array specified in the **InputAttributes** dictionary (see "Matching Requests with Attributes," below).
MediaPosition	integer or null	*(LanguageLevel 3)* The position number of the media source to be used. This parameter does not override the normal media selection process described in the text, but if specified it will be honored—provided it can satisfy the input media request in a manner consistent with normal media selection—even if the media source it specifies is not the best available match for the requested attributes.
DeferredMediaSelection	boolean	*(LanguageLevel 3)* A flag determining when to perform media selection. If *true*, media will be selected by an independent printing subsystem associated with the output device itself, at some time after the execution of the **setpagedevice** operator (see "Deferred Media Selection" on page 411). If *false*, media selection is to be done in the normal way, as described below.

Matching Requests with Attributes

The page device parameters include a special dictionary, **InputAttributes**, that describes the attributes of all physical media available on the current output device. Depending on the device, this information may either be discovered automatically by the PostScript interpreter or configured manually by a human operator or system administrator. The **setpagedevice** operator matches the media requirements specified by the page description against the attributes described in the **InputAttributes** dictionary to determine which media source to select.

The keys in the **InputAttributes** dictionary are integer position numbers representing media sources on the device. The value associated with each key is a subdictionary whose entries describe the attributes of the medium currently available from that source. Entries in this subdictionary include **PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, and **InsertSheet**. These keys have the same meanings as the correspondingly named page device parameters,

but they describe the actual attributes of the medium instead of the requirements of the page description.

Note: *All implementations of* ***setpagedevice*** *support input media selection by means of the* ***PageSize***, ***MediaColor***, ***MediaWeight***, ***MediaType***, ***MediaClass***, *and* ***InsertSheet*** *attributes, whether or not the device can sense these attributes automatically. On some devices, other attributes may also influence media selection; those attributes can appear in the* ***InputAttributes*** *dictionary as well.*

Each time **setpagedevice** is invoked, it uses the following algorithm to match the requested media attributes with those of the available media in order to select a media source:

1. Merge the entries in the request dictionary passed to **setpagedevice** with the existing parameter values in the page device dictionary (see Section 6.1.1, "Page Device Dictionary"). The resulting set of entries are considered together, without regard to which ones were specified in the request dictionary and which were inherited from the existing page device parameters.
2. Collect together those of the relevant entries (**PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, and **InsertSheet**) whose values are not *null* and treat them as an *input media request*. Ignore any entries whose values are *null*.
3. Compare the attributes specified in the input media request with those of each media source in the **InputAttributes** dictionary. If all of the corresponding attributes are equal, then the given media source matches the media request. (**PageSize** values need not match exactly but must only fall within a tolerance of 5 units of default user space in each dimension, as described under **PageSize** in Table 6.2.)
4. If step 3 identifies exactly one media source matching the media request, select that source. If there is more than one match, select the source with the highest priority (see below). If there are no matches at all, consult the **Policies** dictionary to determine what further action to take. (See Section 6.2.7, "Unsatisfied Parameter Requests," for information on the **Policies** dictionary in general, and "Recovery Policies and Media Selection" on page 436 for its use in media selection.)

For example, suppose the value of the **InputAttributes** dictionary for a device is as follows:

```
<<    0 << /PageSize [612 1008] >>
      1 << /PageSize [612 792] >>
>>
```

This describes a device with two paper trays: tray 0, containing legal-size (8.5-by-14-inch) paper, and tray 1, containing letter-size (8.5-by-11-inch) paper. Now suppose a page description executes the following PostScript code:

```
<< /PageSize [612 792] >> setpagedevice
```

Since the requested value of **PageSize** matches the **PageSize** attribute for tray 1 and there are no nonmatching requests, **setpagedevice** will select tray 1 as the media source for this page description.

Each **InputAttributes** subdictionary is required to have a **PageSize** entry, but other attributes are optional. A nonnull attribute value in the media request will not match a media source for which the corresponding attribute is absent. For example, consider an output device with two paper trays, both containing letter-size (8.5-by-11-inch) paper: a high-quality white office paper in tray 0 and a less expensive paper in tray 1. The contents of the **InputAttributes** dictionary for this device might be as follows:

```
<<    0  << /PageSize [612 792]
            /MediaColor (white)
            /MediaType (office)
         >>
      1  << /PageSize [612 792] >>
      /Priority [1 0]
>>
```

Note that the **MediaColor** and **MediaType** attributes are given for media source 0 but not for source 1, and that other attributes (**MediaWeight**, **MediaClass**, and **InsertSheet**) are not given for either source. (The meaning of the **Priority** entry is discussed below.) If a page description now issues the input media request

```
<<    /PageSize [612 792]
      /MediaColor (white)
>> setpagedevice
```

setpagedevice will select input tray 0. This is because, although the requested **PageSize** value matches those in both entries of the **InputAttributes** dictionary, the nonnull value specified for **MediaColor** cannot match source 1, for which that attribute is absent. Source 0 is therefore the only media source matching the request. The values of the **MediaWeight**, **MediaType, MediaClass,** and **InsertSheet** attributes in the request are *null* (assuming that nonnull values have not been inherited from the existing state of the device), and are therefore ignored for matching purposes. In effect, a *null* attribute value in a media request means "don't care."

Given the same **InputAttributes** dictionary as in the previous example, the media request

```
<<    /PageSize [612  792]
      /MediaColor (red)
>> setpagedevice
```

will not match either media source. The requested value of **MediaColor** does not match the **MediaColor** attribute for tray 0, and the **MediaColor** attribute for tray 1 is unspecified and therefore cannot satisfy any request for a specific color. Information in the **Policies** dictionary determines what to do when there is no match with any available medium; see Section 6.2.7, "Unsatisfied Parameter Requests."

Now consider the media request

```
<< /PageSize [612  792] >> setpagedevice
```

This request matches both tray 0 and tray 1 (again, assuming that nonnull values for **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass,** and **InsertSheet** have not been inherited from the existing state of the device). In this situation, the **Priority** entry in the **InputAttributes** dictionary determines which tray to select. Its value is an array of integer position numbers designating available media sources in order of decreasing priority. When an input media request matches two or more media sources, **setpagedevice** chooses the source that appears earliest in the **Priority** array. (If none of the matching sources appears in the array, or if no **Priority** entry is present, one of the sources is chosen arbitrarily.)

The effect of the **InputAttributes** dictionary in the example above is that a page description explicitly requesting a **MediaColor** of white or a **MediaType** of office (or both) will be printed on paper from tray 0, but a page description not re-

questing either of these attributes will be printed on the less expensive paper from tray 1.

Special Media Handling

Certain media are intended for special purposes, such as company letterhead or preprinted forms. Such media should be selected only if a page description specifically requests *all* of their attributes. For example, company letterhead should be selected only if a program explicitly requests it. If the program simply requests letter-size paper, it is inappropriate for **setpagedevice** to satisfy this request by selecting company letterhead, even if it happens to be the only available medium of the requested size.

Suppose the available media consist of legal-size paper (8.5 by 14 inches) in tray 0 and letter-size company letterhead (8.5 by 11 inches) in tray 1. The contents of the device's **InputAttributes** dictionary might be something like this:

```
<<    0  << /PageSize  [612  1008] >>
      1  << /PageSize  [612  792]
            /MediaType  (letterhead)
            /MatchAll  true
         >>
>>
```

The **MatchAll** attribute in entry 1 indicates that media source 1 can satisfy only requests that explicitly specify *all* of that source's attributes. That is, the request

```
<<    /PageSize [612  792]
      /MediaType (letterhead)
>> setpagedevice
```

will select media source 1, but

```
<< /PageSize [612  792] >> setpagedevice
```

will select *neither* media source; **setpagedevice** will consult the **Policies** dictionary to decide what to do.

The precise effects of **MatchAll** are as follows:

- If **MatchAll** is present in an **InputAttributes** subdictionary and its value is *true*, an input media request will match that media source only if it specifies matching (nonnull) values for *all* attributes present in the subdictionary (except the **MatchAll** attribute itself).
- If **MatchAll** is *false* or absent, an input media request will match the media source if it specifies *any subset* of the subdictionary's attributes and leaves the others *null* (meaning "don't care").

Some devices support media sources that insert a sheet of medium directly into the output bin, bypassing the normal printing mechanism. This can be useful for special media that require no imaging, such as divider pages made of heavy card stock, or for delicate, preprinted media such as photographic materials, which might be damaged if passed through a laser printer's high-temperature toner station. The boolean page device parameter **InsertSheet** provides support for media sources of this type. Setting **InsertSheet** to *true* in a source's **InputAttributes** entry signifies that it bypasses the printing mechanism and feeds media directly to the output bin; the source can then be selected by an input media request with **InsertSheet** equal to *true*. For example, the following PostScript code fragment inserts a sheet of special media as page $n + 1$ of a document:

```
… PostScript code for page n …
gsave                                   % Save previous media attributes
    << /InsertSheet true >> setpagedevice   % Select InsertSheet media source
    showpage                            % Send to output bin
grestore                                % Restore previous media attributes
… PostScript code for page n + 2 …
```

Managing the InputAttributes Dictionary

Although the **InputAttributes** dictionary is an ordinary page device parameter and can be altered with the **setpagedevice** operator, a page description should never do this. This dictionary is intended to describe the attributes of the available media sources; it should be changed only by a human operator or by system management software in control of the physical device.

Some devices can sense the attributes of the available media sources automatically. For example, many printers can sense the size of the paper loaded into an input paper tray. Some printers can sense other attributes as well, usually by reading coded tags attached to the trays.

When a PostScript implementation can sense media attributes, it automatically updates the contents of the **InputAttributes** dictionary to reflect the physical state of the hardware. How and when this is done is product-dependent, but the following conventions are typical:

- At the beginning of a job (see Section 3.7.7, "Job Execution Environment"), the job server senses the attributes of all available media sources. It then invokes **setpagedevice** to update the **InputAttributes** dictionary accordingly.
- Additionally, the job server selects a default media source. This default is used if a page description fails to specify any media requirements. (Nonnull attributes of the default medium will be inherited during a **setpagedevice** request that does not explicitly override those attributes.) How defaults are selected is device-dependent; a common method is to use the first element of the **Priority** array if one is present.
- Execution of **setpagedevice** at other times may also result in **InputAttributes** being updated to reflect the state of the hardware. In particular, this occurs if a **Policies** recovery policy specifies interaction with a human operator and the operator installs different media (see Section 6.2.7, "Unsatisfied Parameter Requests"). It can occur at other times as well.

Some devices cannot sense media attributes automatically, or they can sense page size but not other attributes. For such devices, **setpagedevice** must be invoked explicitly to update the **InputAttributes** dictionary whenever media are changed. This is usually done by a system management program submitted by a human operator and executed as an unencapsulated job. Some devices provide a "front panel" user interface to accomplish this.

Changes to the contents of the **InputAttributes** dictionary are cumulative. **setpagedevice** combines the entries supplied to it with those in the existing page device dictionary, replacing or adding entries as appropriate. However, this cumulative behavior does not extend to the contents of the subdictionaries that are

the values of individual entries in **InputAttributes**. For example, suppose the contents of the device's **InputAttributes** dictionary are as follows:

```
<<    0  << /PageSize  [612  1008] >>
      1  << /PageSize  [612  792]
            /MediaType  (letterhead)
            /MatchAll  true
         >>
>>
```

If a program executes

```
<< /InputAttributes
      << 1  << /PageSize [612  792] >>
         /Priority  [1  0]
      >>
>> setpagedevice
```

then the device's **InputAttributes** dictionary becomes

```
<<    0 << /PageSize  [612  1008] >>
      1 << /PageSize  [612  792] >>
      /Priority  [1  0]
>>
```

In other words, entry 0 is left undisturbed, entry 1 is replaced by the one given in the request dictionary supplied to **setpagedevice**, and the new **Priority** entry is inserted. Note that the *contents* of the subdictionary in entry 1 are not merged cumulatively, but are simply replaced.

Note: *If an entry in* ***InputAttributes*** *has a null value instead of a dictionary, it represents a media source that is unavailable for use—for example, no paper tray is installed. If a single execution of* ***setpagedevice*** *includes changes to* ***InputAttributes*** *as well as requests for other features, the merging of the* ***InputAttributes*** *dictionary occurs before the processing of other features.*

Deferred Media Selection

On some output devices, printing is performed by an independent subsystem that is not under the direct control of the PostScript interpreter. The actual printing of pages may, in fact, take place long after the PostScript program describing them has finished executing. For example, the result of executing a PostScript page description may be to produce an electronic representation of the document in some other format, such as Portable Document Format (PDF). In such cases, the **setpagedevice** operator's usual media selection process may not be feasible, because the needed information about the available media may not be available at the time the operator is invoked.

The boolean page device parameter **DeferredMediaSelection** indicates that media selection is to be performed by the output device itself, rather than by the **setpagedevice** operator. When this parameter is *true*, **setpagedevice** merely saves the contents of the PostScript program's input media request and passes them to the device's printing subsystem along with the contents of the rendered page image. The device assumes responsibility for all media selection decisions, including any needed interactions with the human operator and any error processing in the event that the media request cannot be satisfied.

When media selection is deferred, **setpagedevice** must nevertheless make certain decisions immediately. The most important of these is establishing the proper transformation matrix from default user space to device space. This precludes deferring decisions such as the orientation of the media or application of the adjustments required by **PageSize** recovery policies 3 and 4 (see Section 6.2.7, "Unsatisfied Parameter Requests").

Note: *Usually, a given device will support either immediate or deferred media selection, but not both. However, some devices can support either model, depending on the value of the* ***DeferredMediaSelection*** *parameter. Changing* ***DeferredMediaSelection*** *may cause certain other page device parameters to be reinitialized to default values instead of being inherited from their previous values. The details of this behavior are device-dependent.*

6.2.2 Roll-Fed Media

Some types of medium, such as photographic film, are fed into the printing mechanism from a continuous roll, rather than as individual sheets. Typically, the medium is cut after printing to produce a separate physical sheet for each page image, though this may vary from one medium or device to another. Table 6.3 shows page device parameters pertaining to media of this type.

TABLE 6.3 Page device parameters related to roll-fed media

KEY	TYPE	VALUE
RollFedMedia	boolean	*(LanguageLevel 3)* A flag specifying whether the medium is roll-fed. The value of this flag may affect the matching criteria used for the **PageSize** page device parameter (see the description of **PageSize** in Table 6.4 on page 414).
Orientation	integer	A code specifying the orientation of the page image on the medium. (The **PageSize** page device parameter may not determine this orientation unambiguously, since pages on a roll-fed medium have no inherent orientation.) 0 Use normal default orientation as specified by **PageSize**. 1 Rotate the image on the medium 90 degrees counterclockwise with respect to the default orientation. 2 Rotate the image 180 degrees with respect to the default orientation. 3 Rotate the image 270 degrees counterclockwise (equivalent to 90 degrees clockwise) with respect to the default orientation.
AdvanceMedia	integer	A code indicating whether and when to advance the medium by an extra amount (specified by **AdvanceDistance**) in addition to the space occupied by the page images themselves: 0 Do not advance the medium. 1 Advance the medium at device deactivation. 2 Advance the medium at the end of the job. Advancing between jobs is controlled by the value of **AdvanceMedia** for the page device that is current between jobs. Thus, this feature can be turned on or off only by executing **setpagedevice** as part of an unencapsulated job. 3 Advance the medium after each page set, as defined by the **Collate** parameter (see Section 6.2.4, "Page Delivery"). 4 Advance the medium after each **showpage** or **copypage** operation.

AdvanceDistance	integer	The extra distance, in units of the default user space (72nds of an inch), by which to advance the medium as indicated by **AdvanceMedia**.
CutMedia	integer	A code indicating when to cut the medium: 0 Do not cut the medium. 1 Cut the medium at device deactivation. 2 Cut the medium at the end of the job. Cutting between jobs is controlled by the value of **CutMedia** for the page device that is current between jobs. Thus, this feature can be turned on or off only by executing **setpagedevice** as part of an unencapsulated job. 3 Cut the medium after each page set, as defined by the **Collate** parameter (see Section 6.2.4, "Page Delivery"). 4 Cut the medium after each **showpage** or **copypage** operation.

6.2.3 Page Image Placement

The page device parameters shown in Table 6.4 control the placement of page images on the physical medium. This includes such things as:

- The size and location of the image's outer boundary
- The size of the margins, if any
- Whether and how far to shift right-hand and left-hand pages to allow extra space for binding or stapling
- Whether to reflect the page image, either horizontally or vertically, for photographic reproduction
- Whether to generate photographic negatives
- Whether to print on one side (simplex) or both sides (duplex) of the medium

TABLE 6.4 Page device parameters related to page image placement

KEY	TYPE	VALUE
HWResolution	array	An array of two numbers, [*x y*], specifying the resolution of the physical device in pixels per inch along the horizontal and vertical dimensions of device space. Most devices support only a single resolution or certain specific resolutions, not arbitrary ones. The **HWResolution** entry in the output device dictionary (see Section 6.4, "Output Device Dictionary") defines the allowable values for this parameter on a given device; attempting to set it to an unsupported value will cause a configuration error.
ImagingBBox	array or null	An optional bounding box defining an outer boundary for each page image. If not *null*, the value is an array of four numbers in the default user coordinate system giving the left, bottom, right, and top coordinates, respectively, of the bounding box. By specifying a page bounding box, a PostScript program asserts that it will not paint any marks outside the box. Marks that do fall outside the bounding box may or may not be rendered on the output medium. **ImagingBBox** does not necessarily produce the same effect as the **clip** operator, and it should not be used in place of that operator for clipping page content. The page bounding box should lie entirely within the overall page boundaries defined by the **PageSize** parameter (see Table 6.2 on page 400). A value of *null* denotes the largest bounding box that is possible for the given value of **PageSize**. (This may not encompass the entire sheet of physical medium, however, since many devices are incapable of placing marks close to the edges of the medium.) If a program specifies **PageSize** but not **ImagingBBox**, it should explicitly set **ImagingBBox** to *null* to prevent it from inheriting an inappropriate value from the previous device state. Applications should provide a page bounding box whenever possible, since it can improve performance by freeing raster memory for other purposes. For example, if an application knows that all pages will carry an unpainted border, it should indicate this by excluding the unpainted area from **ImagingBBox**.
ImageShift	array	*(LanguageLevel 3)* An array of two numbers, [*x y*], indicating the distance, in default user space units, that each page image is to be shifted horizontally (x) and vertically (y) with respect to the physical medium. For page images appearing on the front (recto) side of the medium, the horizontal shift is to the right if $x > 0$ and to the left if $x < 0$; the vertical shift is upward if $y > 0$ and downward if $y < 0$. For page images appearing on the back (verso) side, these directions are reversed.

PageOffset	array	*(LanguageLevel 3)* An array of two numbers, [*x y*], that reposition the page image on the physical medium by *x* units horizontally and *y* units vertically, in the direction of increasing coordinate values in the device coordinate system. *x* and *y* are expressed in units of the default user space (72nds of an inch). This parameter is typically used on imagesetters to control the placement of the page image on the medium. It differs from **Margins** (below) in that the repositioning is typically accomplished by altering the current transformation matrix, although on some devices it may instead be accomplished by device-dependent means that are independent of the graphics state (in particular, of the CTM).
Margins	array	An array of two numbers, [*x y*], that reposition the page image on the physical medium by *x* units horizontally and *y* units vertically, in the direction of increasing coordinate values in the device coordinate system. *x* and *y* are expressed in device-specific units, usually units of device space at one of the supported resolutions. The purpose of this parameter is to compensate for mechanical misalignments in the device, not to perform purposeful positioning of the page image itself. It differs from **PageOffset** (above) in that the repositioning is typically accomplished by device-dependent means that are independent of the graphics state (in particular, of the current transformation matrix). The range and precision of the parameter values may be restricted by the physical implementation.
MirrorPrint	boolean	A flag specifying whether the page image should be reflected along one of the axes of device space. The reflection is typically accomplished by device-dependent means that are independent of the graphics state (in particular, of the current transformation matrix). This feature is supported only by certain devices, such as imagesetters, that produce output intended for further photographic processing. For example, when output is produced on transparent film, **MirrorPrint** controls whether the page image should be viewed with the film emulsion face up or face down.
NegativePrint	boolean	A flag specifying, if *true*, that the page image should be produced in color-inverted (negative) form. The inversion is typically accomplished by device-dependent means that are independent of the graphics state (in particular, of the transfer functions). The *entire* page is inverted, perhaps including portions that lie outside the imageable area or that are generated independently of the PostScript interpreter. This feature is supported only by certain devices, such as imagesetters, that produce output intended for further photographic processing.

Duplex	boolean	A flag determining whether the output is to be printed *duplex* (on both sides of the physical medium) or *simplex* (on one side of the medium only). If this flag is *true*, pairs of consecutive page images will be printed on opposite sides of a single sheet of medium; if *false*, each page will be printed on a separate sheet. On device activation, a duplex device always prints the first page on a new sheet of medium; on deactivation, it automatically delivers the last sheet of medium if it has been printed on only one side.
Tumble	boolean	A flag specifying the relative orientation of page images on opposite sides of a sheet of medium in duplex output (that is, when **Duplex** is *true*). If **Tumble** is *false*, the default user spaces of the two pages are oriented suitably for binding at the left or right, with vertical (y) coordinates in the two spaces increasing toward the same edge of the physical medium. If **Tumble** is *true*, the default user spaces are oriented suitably for binding at the top or bottom, with vertical coordinates increasing toward opposite edges of the medium. If **Duplex** is *false*, the **Tumble** parameter has no effect. Note that **Tumble** is defined in terms of default user space—the user space established by **setpagedevice**. The orientation of default user space with respect to the medium is determined by the **PageSize** and **Orientation** parameters, possibly altered by the **Install** procedure. Consistent results are obtained across all devices that support duplexing, regardless of how the medium moves through the printing mechanism. However, if a page description alters user space by invoking operators such as **rotate**, the visual orientation of the material printed on the page may differ from the orientation of default user space.

6.2.4 Page Delivery

Just as a device can draw its input medium from one or more media sources, it can have one or more *media destinations*, such as output trays or collating bins, to which to deliver the finished output. The selection of an output destination is similar to that of an input source, as described above (see "Matching Requests with Attributes" on page 403). In place of **InputAttributes**, destination selection uses a device parameter named **OutputAttributes** to describe the media destinations available on the device. This is a dictionary similar in structure to **InputAttributes**, except that the subdictionaries describing individual media destinations contain only the single attribute **OutputType**. A page description can request a specific value for this attribute, which **setpagedevice** matches against those of the available destinations to determine which destination to select.

OutputAttributes and **OutputType** are supported only for those devices that provide multiple output destinations. Table 6.5 summarizes these parameters, as well as others pertaining to the disposition of output, such as the number of copies to be printed and the direction in which printed pages are stacked in the output tray.

TABLE 6.5 Page device parameters related to page delivery

KEY	TYPE	VALUE
OutputDevice	name or string	*(LanguageLevel 3)* The name of the output device for which this page description is destined. The name should match that of an instance of the **OutputDevice** resource category (see Section 6.4, "Output Device Dictionary"). In environments in which the PostScript interpreter can generate output for multiple page devices, this parameter can be used to select between devices of different types (such as a printer and an imagesetter) or between similar devices (such as two or more imagesetters). In an interpreter that supports only a single page device, the **OutputDevice** parameter is typically absent. When a request dictionary passed to **setpagedevice** changes the value of **OutputDevice**, all other page device parameters not explicitly specified in the request dictionary are reinitialized to default values specific to the new device, rather than simply inheriting their previous values in the usual way. The set of valid page device parameters themselves may also change, since different devices have different configurable features.
OutputType	string or null	An arbitrary string representing special attributes of the output destination, analogous to the **MediaType** parameter for input (see Table 6.2 on page 400). If not *null*, this parameter is used in conjunction with **OutputAttributes** to select an appropriate output destination.
NumCopies	integer or null	The number of copies to produce, either of each page individually or of the document as a whole, depending on the value of the **Collate** parameter (see below). A null value indicates that the interpreter should consult the value associated with the name **#copies** in the current dictionary stack each time a page is printed (by **showpage**, **copypage**, or device deactivation); this is compatible with the LanguageLevel 1 convention.
Collate	boolean	A flag specifying how the output is to be organized when multiple copies are requested (via **NumCopies** or **#copies**) for a multiple-page document. Output consists of *page sets* that are delivered together. If **Collate** is *true*, a page set consists of one copy of all pages of the document; if it is *false*, a page set consists of all copies of one page of the document.

If the **Collate** flag is *true*, the exact manner in which collation is performed is device-dependent. If the device has a physical sorting mechanism and the number of copies requested is no greater than the number of available sorting bins, the device itself handles the collation by mechanical means. Otherwise, the interpreter may need to reorder the output in order to deliver all pages of a set together. This may potentially require executing all page descriptions for the entire document and storing the results so that they can be delivered repeatedly to the printing engine in the correct order. This method of collation is supported by relatively few devices and is subject to resource limits in those that do support it.

A collated page set can span multiple invocations of **setpagedevice** within a single job, as long as the requested number of copies does not change and the device is physically capable of delivering the output in collated form. If this is not possible, the document is broken into sections determined by the device's collating capacity; pages are collated only within each section.

Jog integer

A code specifying when output pages should be "jogged" (physically shifted in the output tray):

0 Do not jog pages at all.

1 Jog at device deactivation.

2 Jog at the end of the job. Jogging between jobs is controlled by the value of **Jog** for the page device that is current between jobs. Thus, this feature can be turned on or off only by executing **setpagedevice** as part of an unencapsulated job.

3 Jog after each page set (as defined by the **Collate** parameter).

OutputFaceUp boolean

A flag specifying the order in which pages are stacked in the output tray. If this flag is *true*, pages are stacked with the back side of each page placed against the front of the previous page; the resulting stack is thus ordered backwards from normal reading order. If the flag is *false*, the pages are stacked with the front side of each page placed against the back of the previous page; this places the pages in correct reading order. These are the effects usually produced by face-up and face-down stackers, respectively; however, the name **OutputFaceUp** is a misnomer, since the parameter actually indicates stacking order rather than orientation. Most devices support only one or the other of these two stacking methods; relatively few can support both. The value of **OutputFaceUp** typically indicates the single stacking direction that the device supports.

OutputAttributes	dictionary	A dictionary specifying the attributes of all media destinations currently available for use by this output device. The dictionary contains an entry for each available destination—for example, each output paper tray on a printer. The key for each entry is an arbitrary integer position number; the value is a subdictionary describing the attributes of that destination. Each such subdictionary must include an **OutputType** entry, with the same meaning as the corresponding page device parameter described in this table. Two other optional entries, **Priority** and **MatchAll**, control details of the matching algorithm in a way analogous to the corresponding entries in an **InputAttributes** subdictionary; see "Matching Requests with Attributes" on page 403 for more information. The cumulative merging convention and the interaction with deferred media selection are also the same as for **InputAttributes**.

One parameter in the table that is particularly worth noting is **Collate**. When multiple copies of a document are requested (**NumCopies** > 1), pages are delivered to the output destination in *page sets.* The boolean value of the **Collate** parameter determines the makeup of these sets and thus the order in which the pages are delivered:

- When **Collate** is *true*, each page set consists of one complete copy of the entire document—one copy of each page, arranged in their correct order. The number of page sets delivered is equal to the number of copies requested.
- When **Collate** is *false*, each page set consists of the requested number of copies of a single page of the document. The number of page sets delivered is equal to the number of pages in the document.

This notion of a page set is important because it affects the behavior of several other page device parameters, such as **Jog** in Table 6.5 and **AdvanceMedia** and **CutMedia** in Table 6.3.

6.2.5 Color Support

The page device parameters discussed in this section control various aspects of color output on devices capable of producing it. These include:

- The basic model used for rendering process colors on the device, such as CMYK (cyan-magenta-yellow-black)

- Whether the device produces multiple separations (one for each colorant) or composite output (all colorants combined)
- The number and names of the available colorants
- The subset of colorants that are applied or the order in which separations are generated

Table 6.6 shows the page device parameters related to color. See Section 4.8, "Color Spaces," and Chapter 7, "Rendering," for more information on the subject of color in general.

TABLE 6.6 Page device parameters related to color support

KEY	TYPE	VALUE
PageDeviceName	string, name, or null	*(LanguageLevel 3)* The name of the device configuration represented by this page device dictionary. The **GetPageDeviceName** procedure in the **ColorRendering** procedure set returns this value, which is used by the **findcolorrendering** operator to construct the name of a color rendering dictionary for a requested rendering intent. See Section 7.1.3, "Rendering Intents," and "Customizing CRD Selection" on page 472 for information on **findcolorrendering** and **GetPageDeviceName**, respectively. See also the **MediaClass** page device parameter in Table 6.2 on page 400, which can affect CRD selection.
ProcessColorModel	name or string	*(LanguageLevel 3)* The model used for rendering process colors on the device (see "Process Color Model" on page 422). This parameter controls the rendering process only; it does not affect the interpretation of color values in any color space. The **ProcessColorModel** entry in the output device dictionary (see Section 6.4, "Output Device Dictionary") defines the allowable values for this parameter on a given device; attempting to set it to an unsupported value will cause a configuration error. Valid values are **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, **DeviceCMY**, **DeviceRGBK**, and (in LanguageLevel 3) **DeviceN**.
Separations	boolean	A flag specifying whether the device should produce separations or composite output. If this flag is *true*, multiple color separations will be generated for each page of output; that is, a separate sheet of physical medium will be produced for each individual device colorant (primary or spot color) specified by the **SeparationOrder** parameter. If the flag is *false*, all colorants will be combined into a single com-

posite page on a single sheet of physical medium. See "Separations and Device Colorants" on page 424.

In LanguageLevel 2, the availability of this feature is device-dependent; most devices cannot produce separations. In LanguageLevel 3, every device is capable of producing at least one separation; the maximum number of separations is given by the **MaxSeparations** parameter.

MaxSeparations integer

(Read-only; LanguageLevel 3) The maximum number of separations that the device is capable of producing for each page. This limit is a static property of the device, and is independent of the values of page device parameters that control the production of colorants and separations, such as **Separations** and **SeparationColorNames**. However, its value can depend on other parameters affecting memory needs, such as **PageSize** and **HWResolution**. Valid range: 1 to 250.

SeparationColorNames array

(LanguageLevel 3) An array specifying the names of all colorants that are valid values for **Separation** or (in LanguageLevel 3) **DeviceN** color spaces. Colorants implied by the process color model (see "Process Color Model" on page 422) are available automatically and need not be explicitly declared. The **DeviceN** process color model *(LanguageLevel 3)* has no such implicit colorant names; all device colorants in that model must be declared explicitly in **SeparationColorNames**. The **DeviceN** entry in the output device dictionary (see Section 6.4, "Output Device Dictionary") defines the allowable sets of process colorants for the **DeviceN** process color model.

Array elements may be either name or string objects, which are treated equivalently. The order of elements is not significant, and duplicate elements are ignored.

SeparationOrder array

(LanguageLevel 3) An array specifying the colorants to be applied when generating output on the device. Array elements may be either name or string objects, which are treated equivalently. Legal element values are the process colorant names implied by the process color model (see "Process Color Model" on page 422), as well as any additional colorant names declared by the **SeparationColorNames** parameter. The presence of any other names will result in a configuration error.

If physical separations are being produced (the **Separations** parameter is *true*), a separation will be generated for each colorant named in the **SeparationOrder** array, in the order specified. Although all colorants implied by the process color model or explicitly named

		in **SeparationColorNames** are defined and can be specified via **Separation** or **DeviceN** color spaces without recourse to their *alternativeSpace* and *tintTransform* parameters, only those colorants named in **SeparationOrder** will generate physical separations. Multiple occurrences of the same colorant name in the **SeparationOrder** array will produce multiple separations for the same colorant. If composite output is being produced (**Separations** is *false*), **SeparationOrder** merely specifies which colorants are to be applied to the medium. The order of application is unspecified; the order in which the colorants are named in the **SeparationOrder** array is ignored. An empty **SeparationOrder** array specifies that all colorants implied by the process color model or named explicitly in **SeparationColorNames** are to be applied in an unspecified order.
UseCIEColor	boolean	*(LanguageLevel 3)* A flag that enables or disables the remapping of colors from device color spaces to device-independent CIE-based color spaces (see "Remapping Device Colors to CIE" on page 237).
Trapping	boolean	*(LanguageLevel 3)* A flag that enables or disables in-RIP trapping (see Section 6.3, "In-RIP Trapping"). Trapping is enabled if the flag is *true* and the device supports multiple colorants, whether in the form of physical separations or composite output. If the flag is *false* or the device supports only one colorant, trapping is disabled.
TrappingDetails	dictionary	*(LanguageLevel 3)* A dictionary containing parameters that control the operation of in-RIP trapping on this device. See Table 6.10 on page 442 for the contents of this dictionary.

Process Color Model

The page device parameter **ProcessColorModel** specifies the *process color model* used for rendering colors on an output device. *Process colors* are ones that are produced by combinations of one or more standard *process colorants*. Colors specified in any color space except **Separation** or **DeviceN** are rendered as process colors. The process color model defines the set of standard process colorants available on the device, as well as the *native color space* into which program-specified colors are converted, if necessary (see Section 7.2, "Conversions among Device Color Spaces").

The value of **ProcessColorModel** can be either a name object or an equivalent string. Valid values are as follows:

- **DeviceCMYK** specifies **Cyan**, **Magenta**, **Yellow**, and **Black** as process colorants, with **DeviceCMYK** as the native color space.
- **DeviceCMY** specifies **Cyan**, **Magenta**, and **Yellow** as process colorants, with **DeviceRGB** as the native color space. Additive (RGB) color values will be rendered into equivalent combinations of the complementary subtractive (CMY) process colorants.
- **DeviceRGB** specifies **Red**, **Green**, and **Blue** as process colorants, with **DeviceRGB** as the native color space.
- **DeviceRGBK** specifies **Red**, **Green**, **Blue**, and **Gray** as process colorants, with **DeviceRGB** as the native color space. RGB color values representing pure shades of gray (that is, with all three color components equal) will be rendered into equivalent levels of the gray colorant.
- **DeviceGray** specifies **Gray** as the only process colorant, with **DeviceGray** as the native color space.
- **DeviceN** *(LanguageLevel 3)* specifies **DeviceN** as the native color space. No standard process colorants are defined; all colorants must be explicitly declared via the **SeparationColorNames** page device parameter (see "Separations and Device Colorants," below). This model is used for devices whose available process colorants do not correspond to one of the standard device color spaces.

The standard colorant names implied by the process color model are automatically assumed to be available on the output device. These names can be used to produce separations or isolate the control of individual color components in a **Separation** color space (see "Separation Color Spaces" on page 241) or to select halftones in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498). Additional colorant names can be specified with the **SeparationColorNames** and **SeparationOrder** page device parameters, as described in "Separations and Device Colorants," below.

Note that the process color model applies only to color *rendering*, not to color *specification*. A PostScript program can use any convenient color space to specify colors in a page description; in particular, colors can be specified in a **DeviceN** color space in LanguageLevel 3 implementations, even on devices that do not support the **DeviceN** process color model.

Note: *Because conversions from standard device color spaces to a* ***DeviceN*** *color model are device-specific, different devices that support such a model may produce different results, even for the same colorants: the output will appear similar, but the separations or composite colorants used to produce it may differ. If the* ***DeviceN*** *process color model is requested for a device that does not support it, a configuration error will occur. If the recovery policy for dealing with an unsatisfied* ***ProcessColorModel*** *request is to ignore the request, the previous value will remain in effect. This may cause the printing results to differ significantly from expectations (if, for example, the device is a monochrome printer with a native process color model of* ***DeviceGray****). For this reason, explicitly requesting a* ***DeviceN*** *process color model in a page description is not recommended;* ***DeviceN*** *is useful primarily as a device's default color model.*

Separations and Device Colorants

The boolean page device parameter **Separations** specifies whether an output device produces separations or composite output:

- If **Separations** is *true*, each single page of a document will generate multiple pieces of physical output medium, or *separations*—one for each individual device colorant to be applied during the final printing run. These separations can then be used to create a separate printing plate for each colorant. Because output is generated for only one colorant at a time, the **ProcessColorModel** parameter may be set to models other than the one normally used to render process colors on the device. For example, a monochrome device can use the **DeviceCMYK** color model to produce separations for **Cyan**, **Magenta**, **Yellow**, and **Black** colorants (or some subset of these colorants, subject to the **MaxSeparations** limit discussed below).
- If **Separations** is *false*, the device will combine all colorants to form a single sheet of *composite* output for each page. Any value of **ProcessColorModel** other than the one native to the device will result in a configuration error.

As noted in the section "Process Color Model" above, the standard process colorants implied by a device's process color model (such as **Cyan**, **Magenta**, **Yellow**, and **Black** in the **DeviceCMYK** model) are always available for use in **Separation** or **DeviceN** color spaces, as well as for selecting halftones in a type 5 halftone dictionary. The page device parameter **SeparationColorNames** declares the names of additional *spot colorants* to be used for these purposes. (*All* colorants used in the **DeviceN** process color model must be declared in **SeparationColorNames**, since

that model has no implicit colorant names of its own.) Any colorant name used in a **Separation** or **DeviceN** color space that is not either implied by the process color model or explicitly declared in **SeparationColorNames** will be emulated using the color space's *tintTransform* and *alternativeSpace* parameters (see "Separation Color Spaces" on page 241) and rendered into the process color model of the output device.

Some devices support the additional colorants needed to produce high-fidelity color, as described in "DeviceN Color Spaces" on page 245. For example, consider a six-color device using the PANTONE® Hexachrome™ system, whose colorants are cyan, magenta, yellow, black, orange, and green. This can be modeled in either of two ways:

- Set **ProcessColorModel** to **DeviceCMYK** and name orange and green as spot colorants in the **SeparationColorNames** array. Any colors specified in one of the other device color spaces (**DeviceRGB** or **DeviceGray**) or in a CIE-based color space will be transformed into **DeviceCMYK** using the normal color conversion rules (see Section 7.2, "Conversions among Device Color Spaces"). The orange and green colorants are not considered process colorants in this case; they will be used only if they are named explicitly in a **Separation** or **DeviceN** color space.
- Set **ProcessColorModel** to **DeviceN** and name all six colorants explicitly in **SeparationColorNames**. Colors specified in any device or CIE-based color space will be converted to the six **DeviceN** colorants in a device-specific manner.

Because these two models differ in their definition of the process colorants, source colors that are rendered using process colorants will produce different results in the two models. On the other hand, source colors specified in a **DeviceN** color space will produce the same results in both models, so long as they use only colorants named in the **SeparationColorNames** array.

The page device parameter **SeparationOrder** is an array of colorant names specifying which colorants should be applied in the final output and in what order. This parameter can be used to limit the set of separations generated or colorants applied to a subset of those available (either implied by **ProcessColorModel** or declared in **SeparationColorNames**). The **MaxSeparations** parameter specifies the maximum number of separations that a device is capable of producing for each page. When **Separations** is *true*, the number of uniquely named colorants in the

SeparationOrder array must not exceed the value of **MaxSeparations**, or a configuration error will occur.

Note: *An error will be raised if the set of colorants named in* ***SeparationColorNames*** *is inconsistent with the capabilities of the device. The net effect of such an error may be to emulate any named colorants using the available process colorants, via the tintTransform and alternativeSpace parameters of the current color space. If the device cannot resolve a set of requested colorants in the* ***DeviceN*** *process color model, it may revert to a previous state or (for some devices) to some new state in which it can convert all color spaces. All color spaces will always be printed, but the colorants actually used may differ from those requested.*

6.2.6 Device Initialization and Page Setup

The page device parameters shown in Table 6.7 are PostScript procedures that are called by the interpreter at certain critical times: at the beginning and end of each page of a document and whenever the identity of the current page device changes. PostScript programs can use these parameters to perform needed tasks such as initializing the graphics state or painting recurrent elements that appear on every page.

TABLE 6.7 Page device parameters related to device initialization and page setup

KEY	TYPE	VALUE
Install	procedure	A procedure to install parameter values in the graphics state during each invocation of **setpagedevice**. **setpagedevice** calls this procedure after setting up the device and installing it as the current device in the graphics state, but before invoking **erasepage** and **initgraphics**. The **Install** procedure should not do anything besides alter parameters in the graphics state. In general, it can usefully alter only device-dependent parameters, because the succeeding **initgraphics** operation resets all device-independent parameters to their standard values. The only exception is the current transformation matrix; see the discussion following this table. In a device that supports in-RIP trapping (see Section 6.3, "In-RIP Trapping"), the **Install** procedure can invoke **settrapzone** to establish default trapping zones to be reestablished at the beginning of each page.

BeginPage	procedure	A procedure to be executed at the beginning of each page. Before calling the procedure, the interpreter initializes the graphics state, erases the current page if appropriate, and pushes an integer on the operand stack indicating how many times **showpage** has been invoked since the current device was activated.
EndPage	procedure	A procedure to be executed at the end of each page. Before calling the procedure, the interpreter pushes two integers on the operand stack—a count of previous **showpage** executions for this device and a *reason code* indicating the circumstances under which this call is being made: 0 During **showpage** or *(LanguageLevel 3)* **copypage** 1 During **copypage** *(LanguageLevel 2 only)* 2 At device deactivation The procedure must return a boolean value specifying whether to transmit the page image to the physical output device.

Each time the **setpagedevice** operator is invoked, it calls the currently defined **Install** procedure. This gives the PostScript program an opportunity to initialize device-dependent graphics state parameters such as the halftone screen, color rendering dictionary, and flatness tolerance (see Table 4.2 on page 180). In general, the **Install** procedure cannot usefully alter device-independent graphics state parameters such as the current path or color (Table 4.1 on page 179), since these are reinitialized with the **initgraphics** operator after the procedure is executed. An exception is the current transformation matrix; any new CTM set by the **Install** procedure becomes the default matrix for the device and will be used by **initgraphics** in reinitializing the graphics state.

The PostScript interpreter calls the current device's **BeginPage** and **EndPage** procedures, respectively, before beginning and after completing the execution of each page description. With suitable definitions, these procedures can:

- Cause multiple virtual pages within a document to be printed on a single physical page ("2-up" or "*n*-up" printing)
- Shift the positions of even- and odd-page images differently for binding
- Add marks to each page that either underlie or overprint the material provided by the page description

Note: *The use of* ***BeginPage*** *and* ***EndPage*** *to achieve effects spanning multiple pages sacrifices any page independence the document may have. In general, a page descrip-*

tion should not include definitions of ***BeginPage*** *or* ***EndPage*** *in its invocations of* ***setpagedevice****. Instead, a software print manager should prepend such commands to the page description when printing is requested.*

Note: *The following descriptions refer to the* ***showpage*** *and* ***copypage*** *operators. The distinction between those operators exists only in LanguageLevel 2; in LanguageLevel 3, the effects of* ***copypage*** *on the device are the same as those of* ***showpage****.*

The **BeginPage** procedure is called at the beginning of each page:

- **setpagedevice** normally calls **BeginPage** as its last action before returning (except for a possible call to the **PolicyReport** procedure, if needed; see Section 6.2.7, "Unsatisfied Parameter Requests"). This indicates the beginning of the *first* page to be rendered on the device.
- **showpage** and **copypage** call **BeginPage** as their last action before returning. This indicates the beginning of the *next* page, following the one that **showpage** or **copypage** has just ended.
- Operators that reactivate an existing page device call **BeginPage** as their last action before returning.

When **BeginPage** is called, the graphics state has been initialized and the current page erased, if appropriate, in preparation for beginning the execution of the PostScript commands describing a page. The operand stack contains an integer stating the number of executions of **showpage** (but not **copypage**) that have occurred since the current device was activated; that is, the operand is 0 at the first call to **BeginPage**, 1 at the call that occurs during the first execution of **showpage**, and so on. The **BeginPage** procedure is expected to consume this operand. The procedure is permitted to alter the graphics state and to paint marks on the current page.

The **EndPage** procedure is called at the end of each page:

- **showpage** and **copypage** call **EndPage** as their first action. This indicates the end of the preceding page.
- Operators that deactivate the page device call **EndPage** as their first action.

When **EndPage** is called, the PostScript commands describing the preceding page have been completely executed, but the contents of raster memory have not yet

been transferred to the medium and the graphics state is undisturbed. The operand stack contains two integers:

- The number of executions of **showpage** (but not **copypage**) that have occurred since the device was activated, *not* including this one. That is, the operand is 0 at the call to **EndPage** during the first execution of **showpage**, 1 during the second execution of **showpage**, and so on.
- A *reason code* indicating the circumstances under which **EndPage** is being called: 0 during **showpage**, 1 during **copypage**, 2 during device deactivation.

The **EndPage** procedure is expected to consume these operands. The procedure is permitted to alter the graphics state and to paint marks on the current page; such marks are added to the page just completed.

EndPage must return a boolean result specifying the disposition of the current page:

- The value *true* means transfer the contents of raster memory to the medium. Then, if **showpage** is being executed, execute the equivalent of **initgraphics** and **erasepage** in preparation for the next page. (The latter actions are not performed during **copypage**.)
- The value *false* means do not transfer the contents of raster memory to the medium or erase the current page. (If **showpage** is being executed, **initgraphics** is still performed; if the device is being deactivated, the page is still erased.)

The normal definition of **EndPage** returns *true* during **showpage** or **copypage** (reason code 0 or 1) but *false* during device deactivation (reason code 2). That is, normally every **showpage** or **copypage** operation causes a physical page to be produced, but an incomplete last page (not ended by **showpage** or **copypage**) produces no output. Other behavior can be obtained by defining **EndPage** differently.

When **setpagedevice** is executed or when **restore**, **grestore**, **grestoreall** or **setgstate** causes a page device to be deactivated and a *different* page device to be activated, the interpreter takes the following actions:

1. Calls the **EndPage** procedure of the device being deactivated, passing it a reason code of 2. At the time this call is made, the current device in the graphics

state is still the old device. This enables any necessary cleanup actions to be performed, such as printing an incomplete "*n*-up" page.

2. Performs any actions needed on device deactivation, such as those indicated by the **Jog**, **AdvanceMedia**, and **CutMedia** page device parameters.
3. If the **Duplex** page device parameter is *true* and the last sheet has been printed on only one side, delivers this final sheet.
4. Calls the **BeginPage** procedure of the device being activated. At the time this call is made, the current device in the graphics state is the new one. Its count of previous **showpage** executions is reset to 0.

With the exception of step 4 (which **setpagedevice** always performs), these actions occur only when switching *from one page device to another.* They do not occur when the current device remains unchanged or when switching to or from devices of other kinds, such as the cache device or the null device set up by the **setcachedevice** or **nulldevice** operator. Usually, the latter devices are installed only temporarily; for example, **setcachedevice** and the operations for rendering a character into the font cache are bracketed by **gsave** and **grestore**, thereby reinstating the page device that was previously in effect. The page device's **BeginPage** and **EndPage** procedures are not called in such cases and the current page is not erased or otherwise disturbed.

A few examples will illustrate this distinction. Example 6.1 switches between two page devices. All of the activations and deactivations cause the devices' **BeginPage** and **EndPage** procedures to be called, as described above.

Example 6.1

```
dict1 setpagedevice                % BeginPage for device 1
gsave
    dict2 setpagedevice            % EndPage for device 1, BeginPage for device 2
grestore                           % EndPage for device 2, BeginPage for device 1
```

In Example 6.2, on the other hand, temporary activation of the null device does not cause the page device's **EndPage** procedure to be called, nor does reactivation of the page device cause its **BeginPage** procedure to be called. In fact, the state of the page device is not disturbed in any way, since the null device is not a page device.

Example 6.2

```
dict3 setpagedevice                % BeginPage for device 3
gsave
    nulldevice
grestore
```

It is possible to switch devices in an order that prevents a page device's **EndPage** procedure from ever being called. Example 6.3 switches from a page device to a null device without saving a graphics state that refers to the page device. Thus, there is no possibility of reactivating the page device in order to call its **EndPage** procedure. This sequence of operations is *not recommended.*

Example 6.3 (not recommended)

```
gsave
    dict4 setpagedevice            % BeginPage for device 4
    nulldevice
grestore
```

Example 6.4 shows the skeleton structure of a simple two-page document. For clarity, it includes some of the recommended document structuring conventions, (described in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*). The comments to the right indicate the points at which the interpreter calls **BeginPage** and **EndPage** and the arguments it passes to each.

Example 6.4

```
%!PS-Adobe-3.0
… Document prolog …
%%BeginSetup
%%BeginFeature: *Duplex DuplexNoTumble
    << /Duplex true
       /Tumble false
    >> setpagedevice               % 0 BeginPage
%%EndFeature

%%BeginFeature: *PageSize Letter
    << /PageSize [612  792]
       /ImagingBBox null
    >> setpagedevice               % 0 2 EndPage 0 BeginPage
%%EndFeature
%%EndSetup
```

```
%%Page: 1 1
save
    … PostScript language description for page 1 …
restore
showpage                                % 0 0 EndPage 1 BeginPage

%%Page: 2 2
save
    … PostScript language description for page 2 …
restore
showpage                                % 1 0 EndPage 2 BeginPage

%%EOF
… Job server executes restore, which deactivates the page device …
                                        % 2 2 EndPage
```

6.2.7 Unsatisfied Parameter Requests

The **setpagedevice** operator is not always able to satisfy a page description's requests for particular settings of the page device parameters. It may be unable to do so for either of two reasons:

- The device does not support the requested parameter at all; the PostScript interpreter has no idea what it means. For example, if the page description attempts to set a value for the **Duplex** parameter but that parameter is not defined in the page device dictionary, **setpagedevice** treats it as a request for an unknown feature—even if the requested value is *false*.
- The device supports the requested parameter but cannot achieve the requested value at the moment—for example, an A4-size page is requested when the A4 paper tray is not currently installed.

The interpreter can respond to such an unsatisfied parameter request in a variety of ways, such as by ignoring it, raising a PostScript error, or displaying a message on the front panel of the device requesting intervention by the human operator. The **Policies** page device parameter (Table 6.8) is a dictionary specifying which of these actions to take, depending on the particular page device parameter involved in the request.

TABLE 6.8 Page device parameter related to recovery policies

KEY	TYPE	VALUE
Policies	dictionary	A dictionary specifying recovery policies for responding to unsatisfied parameter requests. Entries in this dictionary are keyed by the names of individual page device parameters; the corresponding value for each parameter is an integer code specifying what action to take when a requested setting for that parameter cannot be satisfied. The entry for any given parameter is optional; the dictionary also includes an overall default policy for parameters for which no specific recovery policy is defined. Changes to the contents of the **Policies** dictionary are cumulative; new entries are merged with those already present.

For most entries in the **Policies** dictionary, the key is the name of a page device parameter; the corresponding value is an integer code specifying the *recovery policy* for handling unsatisfied requests for that parameter. For most parameters, there are three policy choices: generate an error (**configurationerror**), ignore the request, or interact with a human operator or a software print manager. For **PageSize** requests, there are additional policy choices. Table 6.9 describes the entries that can appear in the **Policies** dictionary.

TABLE 6.9 Entries in the Policies dictionary

KEY	TYPE	VALUE
PolicyNotFound	integer	A code specifying the recovery policy to use when a requested setting for a page device parameter cannot be satisfied and no specific entry for that parameter is present in the **Policies** dictionary: 0 Generate a PostScript **configurationerror**—that is, do not attempt recovery, but simply abandon the execution of **setpagedevice**, leaving the current values of all page device parameters undisturbed. Before generating the error, **setpagedevice** inserts an **errorinfo** entry into the **$error** dictionary. Error handling in general and **errorinfo** in particular are discussed in Section 3.11, "Errors." 1 Ignore the request and do not consider this parameter in media selection. This is the usual default policy in most products. Subsequent calls to **currentpagedevice** will return a dictionary in which the entry for this parameter is modified as follows:

- Replaced by *null* if it is a media selection request
- Unchanged from its former value if the parameter (other than a media selection request) is known to the device
- Absent if the parameter is unknown to the device

2 Interact with a human operator or print management software to determine what to do. The precise effects of this policy vary among different output devices and environments. Some devices may issue a message (on a front panel, for instance) indicating an operator action that is required, then wait for confirmation. Other devices have no ability to interact with an operator and may generate a **configurationerror** in this case.

PageSize integer

A code specifying the recovery policy to use when a requested value for the **PageSize** parameter cannot be matched (within a tolerance of ±5 units) by any available medium:

0 Generate a **configurationerror**, as described above for **PolicyNotFound**. This is the usual default policy on most devices.

1 Do not consider the **PageSize** parameter in media selection. Subsequent calls to **currentpagedevice** will return a dictionary whose **PageSize** entry describes the medium that was actually selected.

2 Interact with a human operator or print management software, as described above for **PolicyNotFound**.

3 Select the nearest available medium and adjust the page to fit, as described below under "Recovery Policies and Media Selection."

4 Select the next larger available medium and adjust the page to fit.

5 Select the nearest available medium but do not adjust the page.

6 Select the next larger available medium but do not adjust the page.

7 *(LanguageLevel 3)* If the requested page size is within ±5 units of any page size supported by the device, disable media selection altogether and impose the requested page size on the previously selected medium without adjustment; otherwise, generate a **configurationerror**. In the former case, the page device is set up as if the selected medium were of the requested size, ignoring the actual size of the medium. Positioning of the page image on the medium is device-dependent and unpredictable.

This policy exists solely for use in the emulations of certain LanguageLevel 1 compatibility operators that perform media selection and page device setup separately. Unlike all other policies (which

take effect only if a request cannot be satisfied), this policy takes effect during every execution of the **setpagedevice** operator. Because its behavior violates the PostScript page device model, documents that use this policy are not portable; for this reason, it should never be used directly in a page description.

Note that if **DeferredMediaSelection** is *true*, policies 3 and 4 may result in a **configurationerror**, since the needed adjustments to the current transformation matrix cannot be deferred; the effect of policy 7 under these circumstances is device-dependent.

any feature name	integer	A code specifying the recovery policy to use when a requested setting for the designated parameter cannot be satisfied. Keys of this type are not limited to page device parameters recognized by the device, but may include any key that can appear in a request dictionary supplied to **setpagedevice**. The **Policies** dictionary is consulted in the same way for an unknown feature as for a known feature whose requested value cannot be achieved. Except for **PageSize** (which has its own special set of policy codes, as described above), the possible values associated with any such key are the same as those described above for **PolicyNotFound**.
PolicyReport	procedure	A procedure to be called on successful completion of **setpagedevice** if it encountered one or more unsatisfied parameter requests requiring consultation of the **Policies** dictionary. Before calling this procedure, the interpreter pushes a dictionary on the stack identifying the parameters for which unsatisfied requests were encountered and the corresponding recovery actions specified in the dictionary; see "PolicyReport Procedure" on page 438 for details. The procedure can report the actions that were taken or perform alternative actions. Default value: `{pop}`.

Note: *In addition to generating a* ***configurationerror****, the* ***setpagedevice*** *operator can also generate a* ***typecheck****,* ***rangecheck****,* ***undefined****,* ***limitcheck****, or* ***invalidaccess*** *error; see the descriptions of these errors in Chapter 8 for further information.*

Because the **Policies** dictionary is itself a page device parameter, it can be altered with **setpagedevice** like any other such parameter. Ordinarily, a page description composed by an application program should not do this; recovery policies should be changed only by a human operator or by system management software in control of the physical device. However, if a user requests special policies when submitting a print job, it is appropriate for the print management software to insert a **setpagedevice** command to change the contents of the **Policies** dictionary at the beginning of the page description. For example, the user might consider it essen-

tial that a particular job use certain features; if they are not available, the job should be rejected with a **configurationerror** instead of being executed with the requests for those features ignored.

Changes to the contents of the **Policies** dictionary are cumulative. The **setpagedevice** operator merges the entries in the request dictionary supplied to it with those in the existing page device dictionary, replacing or adding page device entries as appropriate. If a single invocation of **setpagedevice** includes changes to **Policies** as well as requests for other parameters, the merging of **Policies** entries occurs before the processing of the other parameters. Thus, the *revised* **Policies** dictionary governs the recovery policy if one of the other parameter requests cannot be satisfied. For example, the code

```
<<    /Duplex true
      /Policies  << /Duplex 0 >>
>> setpagedevice
```

requests duplex printing and generates a **configurationerror** if the device does not support this feature.

Recovery Policies and Media Selection

If a media request fails to match any of the available media sources or destinations described in **InputAttributes** or **OutputAttributes**, **setpagedevice** consults the **Policies** dictionary in an attempt to make an alternative media selection. For each relevant page device parameter (**PageSize**, **MediaColor**, **MediaWeight**, **MediaType**, **MediaClass**, and **InsertSheet** for a source; **OutputType** for a destination), if the recovery policy specified in **Policies** is 1 (ignore), **setpagedevice** replaces the media request with *null.* It then repeats the matching algorithm (steps 2 through 4 on page 404).

Note: *If a media source or destination has a* ***MatchAll*** *attribute of* true*, its attributes will not be matched by media requests that have been ignored.*

If **setpagedevice** can satisfy a nonmatching request in multiple ways, it employs an unspecified algorithm to decide which way is best. In general, for media matching, the strategy is to try to minimize the number of parameter requests that must be ignored in order to achieve a match. **PageSize** is typically considered to be more important than the others, and **Priority** (if present) is used to break ties.

If this second attempt at media selection succeeds, the resulting page device dictionary will contain null values for all parameters other than **PageSize** that were ignored. If **PageSize** was ignored, the dictionary will contain the **PageSize** attribute of the media source that was actually selected.

If the second attempt at media selection fails, the next action depends on whether any of the nonmatching parameters have a recovery policy of 2 (interact with a human operator or print management software). If so, **setpagedevice** performs such interaction, which may cause new media to be installed and **InputAttributes** or **OutputAttributes** to be updated. It then restarts the media selection process from the beginning. If no recovery policy specifies user interaction or if user interaction is not possible, **setpagedevice** terminates unsuccessfully and generates a **configurationerror**.

For **PageSize**, there are additional policy choices that permit compromises to be made in matching the requested page size to the available media. These include all four combinations of the following pair of choices:

- Select an alternative medium that is either nearest in size or the next larger size to the one requested.
- Either adjust the page (by scaling and centering) to fit the alternative medium or perform no adjustment.

The *nearest size* is the one closest in area to the requested size. The *next larger size* is the one smallest in area that is at least as large as the requested size in both width and height. If the specified policy is to select the next larger size but no larger size is available, the nearest size is used.

Once an alternative medium has been selected, the adjustment option determines how the page image is to be placed on the medium—in other words, how the transformation matrix defining the device's default user space is to be computed. To *adjust* the page means to scale the page image (if necessary) to fit the medium, then center the image on the medium. More precisely, adjustment consists of the following two steps:

1. If the selected medium is smaller than the requested page size in either dimension, scale the page image to fit the medium in the more restrictive dimension. Use the same scale factor in both dimensions, so as to preserve the page's as-

pect ratio (height to width). Perform no scaling if the selected medium is at least as large as the requested page size in both dimensions.

2. Center the page image on the medium along both dimensions.

The effect is to set up a "virtual page" conforming to the requested page size (scaled down if necessary) and centered on the physical medium. The origin of user space is the lower-left corner of the virtual page, not that of the physical medium. The value of **PageSize** in the resulting page device dictionary is the **PageSize** value that was requested, not that of the physical medium.

In the case where the page is *not* adjusted, the default user space is not scaled and is aligned with its origin at the lower-left corner of the medium. The effect is precisely as if the medium's actual **PageSize** value had been requested in the first place. If the actual page size is smaller than the requested one along either dimension, the page image will be clipped.

The limited set of built-in policies for handing unsatisfied parameter requests can be augmented by judicious use of the **PolicyReport** procedure in the **Policies** dictionary (see the next section). Additional adjustments to the current transformation matrix can be implemented as part of the device's **Install** procedure (see Section 6.2.6, "Device Initialization and Page Setup").

PolicyReport Procedure

The **Policies** dictionary contains an entry named **PolicyReport**, whose value is a procedure. **setpagedevice** calls this procedure on successful completion if it encountered one or more unsatisfied parameter requests for which it needed to consult **Policies** during its execution. (**setpagedevice** does *not* call **PolicyReport** if it was able to satisfy all requests without consulting **Policies** or if it terminated unsuccessfully with a **configurationerror**.)

Before calling the **PolicyReport** procedure, **setpagedevice** constructs a dictionary and pushes it on the operand stack. The dictionary contains one entry for each parameter request that was initially unsatisfied—that is, each parameter in the original request that was ignored or whose value was adjusted. Each entry's key is the name of the requested parameter; the value is the integer policy code obtained from the **Policies** dictionary. The **PolicyReport** procedure is expected to consume this dictionary from the stack.

For example, suppose the request dictionary supplied to **setpagedevice** includes a request for duplex printing that cannot be met and a page size that does not match any available medium. Suppose further that the **Policies** dictionary specifies policy codes of 1 (ignore the request) for the **Duplex** parameter and 5 (select the nearest available medium and do not adjust) for **PageSize**. Then, on successful completion, **setpagedevice** will call the **PolicyReport** procedure with the following dictionary on the operand stack:

```
<< /Duplex 1
   /PageSize 5
>>
```

There are two main uses for a **PolicyReport** procedure:

- It can transmit a notification to the human operator or print management software, warning that one or more parameter requests were unsatisfied and that substitute actions have been taken.
- It can inspect the resulting page device dictionary and perhaps make additional alterations. This provides additional flexibility when the standard set of policy choices is inadequate.

At the time **setpagedevice** calls the **PolicyReport** procedure, it has completed setting up the new page device and installing it as the current device in the graphics state. It has also called the device's **BeginPage** procedure (see Section 6.2.6, "Device Initialization and Page Setup"). Thus, invoking **currentpagedevice** within the **PolicyReport** procedure will return the page device dictionary for the newly installed device. It is permissible for the **PolicyReport** procedure to invoke **setpagedevice** recursively.

6.3 In-RIP Trapping

On devices such as offset printing presses, which mark multiple colorants on a single sheet of physical medium, mechanical limitations of the device can cause imprecise alignment, or *misregistration*, between colorants. This can produce unwanted visual artifacts such as brightly colored gaps or bands around the edges of printed objects. In high-quality reproduction of color documents, such artifacts are commonly avoided by creating an overlap, called a *trap*, between areas of adjacent color.

Figure 6.1 shows an example of trapping. The light and medium grays represent two different colorants, which are used to paint the background and the glyph denoting the letter A. The first figure shows the intended result, with the two colorants properly registered. The second figure shows what happens when the colorants are misregistered. In the third figure, traps have been overprinted along the boundaries, obscuring the artifacts caused by the misregistration. (For emphasis, the traps are shown here in dark gray; in actual practice, their color would be similar to one of the adjoining colors.)

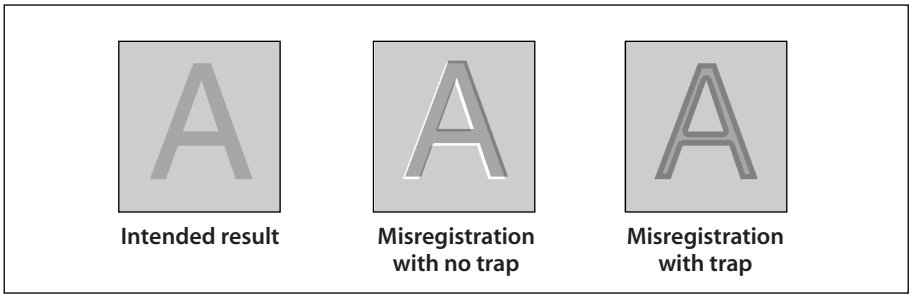


FIGURE 6.1 *Trapping example*

Although trapping can be implemented by the application generating the PostScript page description, such *application-level trapping* suffers from several disadvantages:

- The final set of colorants to be used is not known to the application.
- There may be mismatches with the resources (including fonts) available on the final output device.
- The trapping computations are inherently device-dependent.

In LanguageLevel 3, the optional *in-RIP trapping* feature allows trapping to take place in the raster image processor (RIP) of the output device itself, rather than in the application. This has the advantage of applying consistent trapping techniques to the entire page as late in the output process as possible, when the exact colorants and resources available on the device are known.

Only certain devices support the in-RIP trapping feature, primarily those used in the production of plates for printing presses. On devices that do support it, it is controlled by the following features:

- The **Trapping** page device parameter (see Table 6.6 on page 420) enables or disables trapping as a whole.
- The **TrappingDetails** page device parameter consists of a *trapping details dictionary* (described in the next section) containing additional device-level parameters related to the operation of the trapping facility.
- Independent of the page device dictionary, there is a **Trapping** procedure set containing **settrapzone** and **settrapparams** operators, which specify *trapping zones* and their associated *trapping parameters* to control how trapping is performed in various regions of a page (see Sections 6.3.2, "Trapping Zones," and 6.3.3, "Trapping Parameters").
- Instances of the predefined resource category **InkParams** are colorant details dictionaries (see Table 6.11 on page 443); instances of the **TrapParams** resource category are trapping parameter dictionaries (see Table 6.13 on page 447). These categories ordinarily do not have any predefined instances; they are for the convenience of PostScript programs in managing commonly used sets of colorant details and trapping parameters.

Note: *The set of process color models to which in-RIP trapping applies is implementation-dependent. In general, trapping makes sense only for subtractive color models such as* ***DeviceCMYK****,* ***DeviceCMY****, or* ***DeviceN****; at the time of publication, it is implemented only for* ***DeviceCMYK****.*

6.3.1 Trapping Details Dictionary

The **TrappingDetails** page device parameter holds a *trapping details dictionary* containing information about the operation of trapping on a particular output device. The entries in this dictionary apply globally to all trapping operations on the device. Section 6.3.3, "Trapping Parameters," describes additional trapping parameters that provide finer control over the details of trapping in specific regions of specific pages in a document.

Every trapping details dictionary must contain a **Type** entry, which identifies the particular *trapping type* to which the dictionary applies and determines the format and meaning of its remaining entries (see Section 6.1.2, "Details Dictionaries"). At the time of publication, only one trapping type, type 1001, has been defined. Table 6.10 shows the contents of the trapping details dictionary for this trapping type.

TABLE 6.10 Entries in a Type 1001 trapping details dictionary

KEY	TYPE	VALUE
Type	integer	*(Required)* A code identifying the trapping type to which this dictionary applies. The value of this type code determines not only the format and meaning of other entries in the trapping details dictionary, but also the contents of the trapping parameter dictionary and the **Trapping** procedure set. The **TrappingDetailsType** entry in the output device dictionary (see Section 6.4, "Output Device Dictionary") defines the allowable trapping types for a given device. The only valid value defined at the time of publication is 1001; the entries described in this table correspond to that type. Changes to the contents of a type 1001 trapping details dictionary are cumulative, and this cumulative behavior applies recursively to all levels of included subdictionaries.
TrappingOrder	array	An array specifying the order in which colorants are assumed, for trapping purposes, to be applied to the output medium. This may differ from the order in which colorants are named in the **SeparationOrder** page device parameter (see Section 6.2.5, "Color Support"). Array elements may be either name or string objects, which are treated equivalently. Colorants are assumed to be applied to the medium in the order in which they appear in the array. After all colorants named in the array have been applied, any remaining device colorants specified or implied by the **ProcessColorModel** and **SeparationColorNames** page device parameters are assumed to be applied in an unspecified order. Colorants named in the **TrappingOrder** array that are neither implied by **ProcessColorModel** nor explicitly declared in **SeparationColorNames** are ignored. If the array is empty, all device colorants are assumed to be applied in an unspecified order.
ColorantDetails	dictionary	A dictionary defining trapping-related properties of individual device colorants. See Table 6.11 for the contents of this dictionary.

The value of the **ColorantDetails** entry in the trapping details dictionary is a subsidiary *colorant details dictionary* (see Table 6.11) holding information on the properties of the available device colorants. This information is used, for example, to determine which of two adjacent colors is darker or when a color should be considered black and thus subject to special treatment for trapping purposes.

TABLE 6.11 Entries in a colorant details dictionary

KEY	TYPE	VALUE
ColorantSetName	name, string, or null	The name of the **InkParams** resource instance from which the contents of this dictionary were most recently updated, or *null* if the dictionary does not correspond to such a resource instance. This entry is strictly for information; it is ignored by the PostScript trapping machinery.
any device colorant name	dictionary	A colorant subdictionary describing the properties of the device colorant named by the key. There must be one such subdictionary for each device colorant implied by **ProcessColorModel** or explicitly declared in **SeparationColorNames**. See Table 6.12 for the contents of a colorant subdictionary.

Dictionaries representing commonly used sets of colorants can be stored as instances of the **InkParams** resource category (see Section 3.9, "Named Resources"). Each instance takes the form of a colorant details dictionary whose **ColorantSetName** value is the same as the resource instance's key. A PostScript program can then retrieve an instance with the **findresource** operator and use it to construct a request to update the colorant details dictionary for the page device. The **ColorantSetName** entry in the colorant details dictionary identifies the **InkParams** resource from which the dictionary was most recently updated.

The remaining contents of the colorant details dictionary consist of a set of *colorant subdictionaries*, one for each process colorant implied by **ProcessColorModel** and each spot colorant explicitly declared in **SeparationColorNames**. Each colorant subdictionary is keyed by the name of the colorant it represents. Table 6.12 shows the contents of these subdictionaries.

TABLE 6.12 Entries in a colorant subdictionary

KEY	TYPE	VALUE
ColorantName	name, string, or null	A name identifying the colorant that this subdictionary describes. The name is unrelated to the key that identifies this colorant in the colorant details dictionary; it can be any arbitrary string, such as a vendor name or part number. This entry is strictly for information; it is ignored by the PostScript trapping machinery.
ColorantType	name	A name object specifying how areas marked with this colorant are affected by trapping:
		Normal — Marks made with, covered by, or placed on top of this colorant are subject to trapping.
		Transparent — Marks made with this colorant are not subject to trapping.
		Opaque — Marks made with or covered by this colorant are never spread (see "Normal Trapping Rule" on page 449). Other colorants may trap to this colorant, and colorants covered by this colorant may choke back from it (see "Black Trapping Rule" on page 451).
		OpaqueIgnore — Marks made with or covered by this colorant are never spread, and other colorants do not trap to it.
NeutralDensity	number	The neutral density of this colorant (see below) when it is at its maximum concentration (tint value 1.0). Valid range: 0.001 to 10.0.

A colorant's *neutral density* is used in comparing colors for lightness or darkness. Neutral density is a measure of a colorant's capacity to absorb light—that is, of how dark it is. A neutral density of 0.0 represents the lightest possible color, a pure white that absorbs no light at all and thus reflects all of the light falling on it. The overall neutral density of a composite color is found by adding the neutral densities of its component colorants. The neutral density of a component colorant is given by

$$neutralDensity = -1.7 \times \log(1 - (c \times (1 - 10^{-0.6 \times d})))$$

where c is the concentration (tint value) of the colorant and d is the **NeutralDensity** value given in the colorant subdictionary for that colorant.

In addition to its neutral density, each colorant also has a *colorant type* that specifies how it is affected by the trapping rules. Marks made with a *transparent* colorant, such as a colorless varnish, are exempt from any application of the trapping rules. (Such colorants are also excluded from the neutral density calculation.) An *opaque* colorant, such as a metallic ink that obscures anything it covers, is always treated as if it were black, even if its neutral density would not ordinarily be considered dark enough for such treatment. Any marks in other colorants that are covered by the subsequent application of an opaque colorant (according to the trapping order specified in the trapping details dictionary) are considered to be invisible and are not subject to trapping.

6.3.2 Trapping Zones

Trapping always takes place within a *trapping zone* specified by the PostScript program. Each zone defines an area of the page and a set of *trapping parameters* that govern the way trapping is performed within that area. A graphical object is subject to trapping if it falls within one or more trapping zones, but only those parts of the object that lie within at least one zone will be trapped. If two or more trapping zones overlap, the one most recently defined takes precedence within the area of intersection.

Trapping zones are specified with the **settrapzone** operator, part of the **Trapping** procedure set. The area covered by a zone is defined by the current path in the graphics state at the time **settrapzone** is invoked. The zone's trapping parameters are copied from the current contents of the trapping parameter dictionary, described in the next section. Trapping zones are cumulative within a page, and are unaffected by subsequent changes to the graphics state (except ones that activate a different page device).

Trapping zones established as part of a page description apply only to that particular page and are erased when the page is printed (or when a different page device is activated). A PostScript program can establish default trapping zones that apply to all pages of a document by defining them in the page device's **Install** procedure (see Section 6.2.6, "Device Initialization and Page Setup"). They will then be reestablished before the **BeginPage** procedure is executed at the beginning of each new page. Restoring a deactivated page device via **restore**, **grestore**, or **setgstate** will reestablish only these default trapping zones; any page-specific zones associated with the current page will be lost.

Note: *Trapping zones should be established before any of the objects that fall within them are painted. The results of defining a trapping zone over existing marks already on the page are implementation-dependent and unpredictable.*

6.3.3 Trapping Parameters

Each trapping zone has its own set of *trapping parameters.* Unlike the parameters in the trapping details dictionary, which apply globally to all trapping performed on a given device, the zone-specific trapping parameters govern trapping behavior only within a single zone.

Trapping parameters are stored, altered, and read similarly to page device parameters. The current values of the trapping parameters are stored in a global *trapping parameter dictionary* analogous to the page device dictionary. The keys in this dictionary designate features or options pertaining to a zone's trapping behavior; the associated values represent the current settings of those features or options. The dictionary resides in virtual memory, and hence is subject to **save** and **restore**.

When a new trapping zone is created, its trapping parameters are taken from the current contents of the trapping parameter dictionary. A PostScript program can thus specify each zone's trapping parameters by setting the desired values in the trapping parameter dictionary before invoking **settrapzone** to create the zone. The trapping parameters are fixed at the time the zone is created, and cannot be changed thereafter; subsequent changes to the trapping parameter dictionary have no effect on zones already in existence.

The trapping parameter dictionary, like the page device dictionary, is not directly accessible to a PostScript program. Its contents can be altered and read only indirectly, using the **settrapparams** and **currenttrapparams** operators (both defined in the **Trapping** procedure set). These work similarly to **setpagedevice** and **currentpagedevice**:

- **settrapparams** takes a single operand, a *request dictionary* whose entries specify desired settings or values for one or more trapping parameters. The operator uses the contents of the request dictionary to alter the current trapping parameters, but it does not retain the request dictionary itself.

- The effects of **settrapparams** are cumulative over multiple executions: it merges new parameter requests into the existing trapping parameter dictionary. (However, this cumulative behavior does not apply recursively to the contents of any subsidiary dictionaries contained as entries within the trapping parameter dictionary.)
- Omitting a parameter key from the request dictionary has a different meaning than including the key with a null value. Omitting the key leaves the parameter's previous value unchanged; specifying a null value sets it to the null object, canceling any previous value it may have had.
- The PostScript language does not prescribe a default value for any trapping parameter; all default values are device-specific. A PostScript program can change the defaults by invoking **settrapparams** as part of an unencapsulated job.
- **currenttrapparams** returns a dictionary whose entries reflect the current contents of the trapping parameter dictionary.

The structure and meaning of the entries in the trapping parameter dictionary are determined by the particular trapping type to which they refer, as specified by the **Type** entry in the trapping details dictionary (see Section 6.3.1, "Trapping Details Dictionary"). Table 6.13 shows the trapping parameters for trapping type 1001, the only type defined at the time of publication.

TABLE 6.13 Entries in a trapping parameter dictionary

KEY	TYPE	VALUE
TrapSetName	name, string, or null	The name of the **TrapParams** resource instance from which the contents of this dictionary were most recently updated, or *null* if the dictionary does not correspond to such a resource instance. This entry is strictly for information; it is ignored by the PostScript trapping machinery.
Enabled	boolean	A flag that enables or disables trapping for this zone.
StepLimit	number	The step limit governing the creation of traps by the normal trapping rule (see "Normal Trapping Rule" on page 449). The value of this parameter must be greater than 0. The step limit can be overridden for an individual colorant by including a **StepLimit** entry for that colorant in the **ColorantZoneDetails** dictionary (see "Zone-Specific Colorant Details" on page 454).

TrapWidth	number	The width of traps created under the normal trapping rule, expressed in units of the default user space (72nds of an inch). The size of the unit is not affected by any scaling performed by the current transformation matrix. This parameter also defines the unit used by the **BlackWidth** trapping parameter (see below). The value of this parameter must be greater than 0, and may also be subject to an upper bound imposed by the implementation. If this upper bound is exceeded, the maximum allowed value is substituted without error indication.
TrapColorScaling	number	A scaling factor for reducing the total amount of colorant applied to a trap (see "Normal Trapping Rule" on page 449). Valid range: 0.0 to 1.0. The scaling factor can be overridden for an individual colorant by including a **TrapColorScaling** entry in the **ColorantZoneDetails** dictionary (see "Zone-Specific Colorant Details" on page 454).
BlackDensityLimit	number	The minimum neutral density required for a colorant to be considered black for purposes of the black trapping rule. The value of this parameter must be greater than 0.
BlackColorLimit	number	The minimum concentration of a black colorant needed in order to invoke the black trapping rule. Valid range: 0.0 (no colorant) to 1.0 (full concentration).
BlackWidth	number	The width of traps created under the black trapping rule, expressed in the units defined by the **TrapWidth** trapping parameter (see above). The value of this parameter must be greater than 0.
SlidingTrapLimit	number	A threshold value specifying when a trap should begin to straddle the boundary between two colors. This is useful for creating sliding traps when one of a pair of adjacent objects is painted with a shading pattern whose color shifts along the boundary (see "Sliding Traps" on page 452). The trap will slide to a straddling position when the ratio between the neutral densities of the lighter and darker colors exceeds the specified limit. The higher the limit is, the closer the neutral densities must be for the trap to slide. Valid range: 0.0 (always slide) to 1.0 (never slide).
ImageToObjectTrapping	boolean	A flag specifying whether trapping should be performed between sampled images and other graphical objects, such as filled or stroked paths or character glyphs (see "Image Trapping" on page 453).
ImageInternalTrapping	boolean	A flag specifying whether trapping should be performed between individual colorants within a sampled image (see "Image Trapping" on page 453).

ImageTrapPlacement	name	A name object specifying the positioning of traps between sampled images and adjacent graphical objects (see "Image Trapping" on page 453): Normal — Within the area of either the image or the adjacent object, depending on their respective colors according to the normal trapping rule Spread — Always within the area of the adjacent object Choke — Always within the area of the image Center — Centered on the boundary between the image and the adjacent object
ImageResolution	integer	The minimum resolution, in pixels per inch, to which images will be downsampled. *Downsampling* is the conversion of a sampled image to a resolution lower than the one at which it is specified. The image is marked on the output medium at its original resolution; the downsampled version is used only for calculating traps.
ColorantZoneDetails	dictionary	A dictionary containing colorant-specific parameters for this trapping zone (see "Zone-Specific Colorant Details" on page 454).

Dictionaries representing commonly used sets of trapping parameters can be stored as instances of the **TrapParams** resource category (see Section 3.9, "Named Resources"). Each instance takes the form of a trapping parameter dictionary whose **TrapSetName** value is the same as the resource instance's key. A PostScript program can then retrieve an instance with the **findresource** operator and use it to construct a **settrapparams** request. The **TrapSetName** entry in the trapping parameters dictionary identifies the **TrapParams** resource from which the dictionary was most recently updated.

Normal Trapping Rule

Trapping can occur wherever two distinct colors meet along a common boundary. Depending on the specific situation, various *trapping rules* may be applied to minimize the effects of possible misregistration. The most common of these, the *normal trapping rule*, is to extend, or *spread*, some of the colorants making up the lighter color into the area occupied by the darker. If all colorants are applied in proper alignment, the spread of the lighter color will be obscured by the darker color on the opposite side of the boundary and will not be perceptible to the eye.

In the event of misregistration, the spread will fill the gap where the two colors fail to align precisely, avoiding a visible separation between them.

When two graphical objects share a boundary within a trapping zone, the normal trapping rule compares their colors to determine whether to create a trap. Colorant concentrations are compared separately for each individual device colorant, and a trap is created if the concentrations of at least two colorants differ sufficiently in opposite directions across the boundary. Colorant concentrations are considered to differ sufficiently if they meet both of the following conditions:

- The magnitude of the *absolute difference* between the two concentrations is greater than 0.05.
- The magnitude of the *relative difference* between the two concentrations, in proportion to the lower of the two, is greater than a specified *step limit.*

These two conditions can be expressed mathematically as follows:

$$highLevel - lowLevel \geq 0.05$$

$$\frac{highLevel - lowLevel}{lowLevel} \geq stepLimit$$

where *highLevel* and *lowLevel* are the higher and lower of the concentrations of the given colorant on either side of the boundary. The **StepLimit** trapping parameter establishes a general step limit for a given trapping zone, but this value can be overridden for any individual colorant by including a **StepLimit** entry for that colorant in the **ColorantZoneDetails** dictionary (see "Zone-Specific Colorant Details" on page 454).

For example, suppose two adjacent objects within a trapping zone have the color values shown in Table 6.14, on an output device that uses the **DeviceCMYK** process color model. Because both objects have the same concentration of magenta, that colorant is not a candidate for trapping by the normal trapping rule. The black colorant also is not eligible for trapping, because, although the concentrations of black differ between the two objects, the magnitude of the absolute difference is below the threshold value of 0.05. Both cyan and yellow do meet the first condition for trapping. however, because their concentrations differ by more than 0.05 and in opposite directions. Dividing each of the absolute differences by the smaller of the two concentrations for that colorant yields relative differences of 0.90 in one direction for cyan and 0.75 in the other direction for yellow. If the

step limit for this zone is, say, 0.50, then a trap will be created between these two objects; if the step limit were 0.80, the relative difference for the yellow colorant would be too small and no trapping would occur.

TABLE 6.14 Example of normal trapping rule

COLORANT	COLOR 1	COLOR 2	ABSOLUTE DIFFERENCE	RELATIVE DIFFERENCE
Cyan	0.95	0.50	0.45	0.90
Magenta	0.80	0.80	0.00	0.00
Yellow	0.20	0.35	–0.15	–0.75
Black	0.10	0.12	–0.02	–0.20

Once it is determined that trapping should take place between two objects, the **TrapWidth** trapping parameter specifies how wide a trap to create. The width is expressed in units of the default user space, and so is unaffected by any scaling performed by the current transformation matrix.

Spreading colorants from one side of a trap boundary to the other may sometimes result in a conspicuously dark color within the area of the trap. The **TrapColorScaling** trapping parameter is intended to avoid this problem by scaling back the colorant concentrations to achieve a specified neutral density. The desired density is expressed proportionally on a scale from 0.0 to 1.0, with 0.0 representing the neutral density of the darker of the two adjacent colors and 1.0 representing that of the lighter and darker colors combined.

Black Trapping Rule

A special rule, the *black trapping rule*, governs the case when one of a pair of adjacent colors is black and the other is not. In these circumstances, trapping will always spread the nonblack color toward the black, regardless of their comparative neutral densities. This ensures that the visible edge of the color boundary will always be defined by the black color. The **BlackWidth** trapping parameter specifies the width of traps created under the black trapping rule.

A color need not contain any of the **DeviceCMYK** process black colorant to be considered black for purposes of the black trapping rule. Rather, this determination is based on a pair of trapping parameters of the zone in which the trap falls. The **BlackDensityLimit** parameter defines the minimum neutral density required for any colorant to be considered black; the **BlackColorLimit** parameter defines the minimum concentration of such a colorant that is needed in order to invoke the black trapping rule.

Because process black colorants are typically translucent rather than fully opaque, and because it is difficult to apply solid colorants smoothly over large areas, true process black is often combined with one or more *support colorants* to achieve a more intense black result. When such a "rich black" region adjoins an unpainted region or one whose color lacks any of the support colorants, a *reverse trap* is created: the support colorants of the rich black color are *choked back* from the boundary, thereby preventing a misregistration from creating a halo on one side of the rich black region.

Sliding Traps

The use of shading patterns (described in Section 4.9.3, "Shading Patterns") can lead to still another unusual situation, illustrated in Figure 6.2. As the color varies continuously across the shading, it may change from lighter to darker than the color on the opposite side of the boundary. Under the normal trapping rule, this would cause the trap to jump suddenly from one side of the boundary to the other at the point of the transition, as shown in the figure on the left. This undesirable effect is avoided by creating a *sliding trap* that shifts gradually, rather than abruptly, from one side of the boundary to the other, as in the figure on the right.

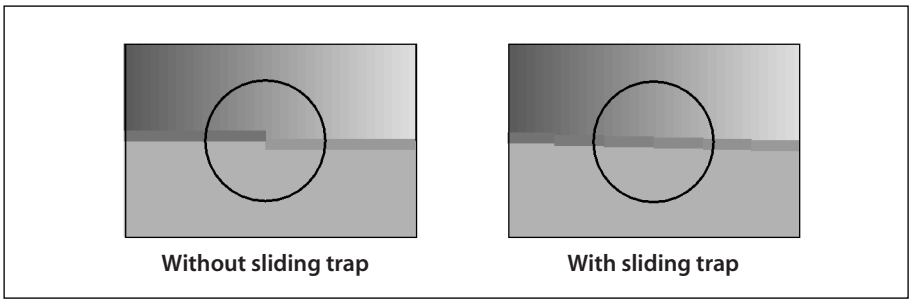


FIGURE 6.2 *Sliding trap*

The **SlidingTrapLimit** trapping parameter controls the position of the trap between the adjoining colors. The value of this parameter is a number in the range 0.0 to 1.0. As long as the ratio between the neutral densities of the lighter and darker colors is less than the specified limit, the trap will be positioned on the darker side of the boundary according to the normal trapping rule. When the color of the shading pattern darkens sufficiently for the ratio of the neutral densities to exceed the limit, the trap will begin to slide to a straddling position on the boundary. The trap will be centered at the point where the neutral densities are equal, then continue to slide fully to the opposite side of the boundary at the point where the ratio (now reversed) is equal to the sliding trap limit. The higher the value of the limit, the closer the neutral densities of the two colors must be for the trap to slide. A limit of 1.0 prevents any sliding traps from being created; a limit of 0.0 causes all traps within the given trapping zone to be centered on the boundary between two colors.

Note: *Although the problem addressed by sliding traps arises mainly from the use of smooth shading patterns, sliding traps in fact will be created any time the normal trapping rule is in effect and the relative neutral densities dictate their creation.*

Image Trapping

Traps between a sampled image and an adjacent graphical object require further special treatment. Because the colors in the image can vary wildly from one sample to the next, the normal trapping rule can result in traps switching rapidly from one side of the boundary to the other. Sliding traps do not help in this case, because the color transitions are not gradual but sudden and unpredictable.

The boolean trapping parameter **ImageToObjectTrapping** can be used to suppress the creation of traps between images and other graphical objects. Alternatively, another trapping parameter, **ImageTrapPlacement**, controls the placement of traps between images and other objects. The available options are:

- Follow the normal trapping rule, placing the trap on the side of the boundary where the color is darker. This option is vulnerable to the uneven trap placement described above.
- Always spread from the image toward the other object, placing the trap within the object. This is useful when dark text is painted on top of a light image, since it avoids enlarging the character shapes.

- Always choke the image data, spreading the other object and placing the trap within the image. This is useful when light text is painted on top of a dark image, since it avoids shrinking the character shapes.
- Center the trap on the boundary between the image and the other object. This is the most generally useful option.

In addition, depending on the nature of the image, it may or may not be appropriate to create traps between individual colorants within the image itself. The **ImageInternalTrapping** parameter enables or disables such internal image trapping. This is not ordinarily done; it is useful for unusually high-contrast images, such as screen shots, where internal color abutments occur along sharp edges.

Note that all of these parameters apply only to true sampled images; images used as stencil masks (see "Stencil Masking" on page 302) are trapped according to the normal trapping rule.

Zone-Specific Colorant Details

The **ColorantZoneDetails** trapping parameter allows certain settings to be specified independently for individual colorants, providing greater control over trapping behavior than the zone-level settings in the main trapping parameter dictionary. The value of **ColorantZoneDetails** is a dictionary that can contain zero or more entries, each keyed by the name of a single device colorant. The value of each entry is in turn a subdictionary containing either or both of the entries shown in Table 6.15. These specify the step limit and the color scaling factor for that individual colorant, overriding the zone-level settings specified by the **StepLimit** and **TrapColorScaling** trapping parameters. If no subdictionary is defined for a given colorant, the zone-level values are used instead.

TABLE 6.15 Entries in a ColorantZoneDetails dictionary

KEY	TYPE	VALUE
StepLimit	number	*(Optional)* The step limit governing the creation of traps for this colorant by the normal trapping rule (see "Normal Trapping Rule" on page 449 and the **StepLimit** entry in Table 6.13 on page 447).
TrapColorScaling	number	*(Optional)* A scaling factor for reducing the total amount of this colorant applied to a trap (see "Normal Trapping Rule" on page 449 and the **TrapColorScaling** entry in Table 6.13 on page 447).

6.4 Output Device Dictionary

In LanguageLevel 3, certain capabilities of a page device, such as the possible page sizes or resolutions, can be summarized in an *output device dictionary* that is stored as an instance of the **OutputDevice** resource category. This enables applications to query device capabilities directly and also maintains functional equivalence with LanguageLevel 1 (where information on page size capabilities is indicated by the presence of **userdict** entries such as **letter**, **legal**, and **a4**).

Most PostScript interpreters support only a single page device; the **OutputDevice** resource category contains a single instance (whose key is arbitrary) describing that device. However, some interpreters support multiple devices, which can be selected by the **OutputDevice** parameter in the page device dictionary (see Table 6.5 on page 417). In that case, the **OutputDevice** resource category contains multiple instances, whose keys are the same as the allowed values of the **OutputDevice** page device parameter.

An output device dictionary does not represent the current state of the page device; it simply provides a static list of some of its capabilities. Table 6.16 shows the entries in an output device dictionary as of the time of publication; additional entries may exist in some products.

TABLE 6.16 Entries in an output device dictionary

KEY	TYPE	VALUE
MediaClass	array	*(Optional)* An array of names or strings specifying the values supported by this device for the **MediaClass** page device parameter (see Table 6.2 on page 400).
PageSize	array	*(Required)* An array specifying page sizes that can be fed automatically on this device. Page dimensions are expressed in units of the default user space (72nds of an inch). Each array element is in turn a subarray, which may consist of two numbers [*width height*], denoting the width and height of a supported page size, or of four numbers [$width_1$ $height_1$ $width_2$ $height_2$], indicating that page sizes in the range [$width_1$ $height_1$] to [$width_2$ $height_2$] are supported. Page sizes may be specified redundantly within the array.
ManualSize	array	*(Optional)* An array specifying page sizes that can be fed manually on this device. Array elements are in the same format as for **PageSize** (see above). On devices that do not support the **ManualFeed** page device parameter (see Table 6.2 on page 400), the **ManualSize** array should be empty or absent.

HWResolution	array	*(Optional)* An array specifying pixel resolutions supported by this device. The contents of this array define the allowable values for the **HWResolution** page device parameter (see Table 6.4 on page 414). Most devices support only a single pixel resolution. Resolutions are expressed in pixels per inch. Each array element is in turn a subarray consisting either of two numbers [$x\ y$], denoting a supported horizontal and vertical pixel resolution, or of four numbers [$x_1\ y_1\ x_2\ y_2$], indicating that resolutions in the range [$x_1\ y_1$] to [$x_2\ y_2$] are supported. Available resolutions may be specified redundantly within the array.
ProcessColorModel	array	*(Optional)* An array of names or strings specifying the process color models supported by this device. The contents of this array define the allowable values for the **ProcessColorModel** page device parameter (see Table 6.6 on page 420). Valid values for elements of the array are **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, **DeviceCMY**, **DeviceRGBK**, and (in LanguageLevel 3) **DeviceN**. This entry is required if the device supports more than one possible value for the **ProcessColorModel** parameter.
DeviceN	array	*(Optional; LanguageLevel 3)* An array specifying sets of colorants supported by this device under the **DeviceN** process color model. Each array element is in turn a subarray identifying a set of **DeviceN** colorants. Subarray elements may be either name or string objects, which are treated equivalently. This entry is not required for devices that support only standard process color models with implied colorants of their own. Because the **DeviceN** color model has no such implied colorants, however, all colorants for that model must be declared explicitly. In that case, the contents of this array define the allowable sets of values for the **SeparationColorNames** page device parameter (see Table 6.6 on page 420).
TrappingDetailsType	array	*(Required if in-RIP trapping is supported; LanguageLevel 3)* An array of integers specifying the trapping types supported by this device. The contents of this array define the allowable values of the **Type** entry in the trapping details dictionary (see Table 6.10 on page 442). At the time of publication, only one trapping type, 1001, has been defined.

CHAPTER 7

Rendering

THE POSTSCRIPT LANGUAGE separates *graphics* (the specification of shapes and colors) from *rendering* (controlling a raster output device). Figures 4.5 and 4.6 on pages 212 and 213 illustrate this division. Chapter 4 describes the facilities for specifying the appearance of pages in a device-independent way. This chapter describes the facilities for controlling how shapes and colors are rendered on the raster output device. All of the facilities discussed here depend on the specific characteristics of the output device; PostScript programs that are intended to be device-independent should limit themselves to the general graphics facilities described in Chapter 4.

Nearly all of the rendering facilities that are under program control have to do with the reproduction of color. The interpreter renders colors by a multiple-step process outlined below. (Depending on the current color space and on the characteristics of the device, it is not always necessary to perform every step.)

1. If a color has been specified in a CIE-based color space (as described in Section 4.8.3, "CIE-Based Color Spaces"), the interpreter must first transform it to the native color space of the raster output device. For devices using the standard device color spaces (**DeviceRGB**, **DeviceCMYK**, or **DeviceGray**), this transformation is controlled by a CIE-based *color rendering dictionary.*
2. If a color has been specified in a device color space that is inappropriate for the output device (for example, RGB color with a CMYK or grayscale device), the interpreter invokes a *color conversion function.* A PostScript program can also request explicit conversions between device color spaces.
3. The interpreter now maps the device color values through *transfer functions*, one for each color component. The transfer functions compensate for peculiarities of the output device, such as nonlinear gray-level response. This step is sometimes called *gamma correction.*

4. If the device cannot reproduce continuous tones, but only certain discrete colors such as black and white pixels, the interpreter invokes a *halftone function*, which approximates the desired colors by means of patterns of pixels.
5. Finally, the interpreter performs *scan conversion* to mark the appropriate pixels of the raster output device with the requested colors.

Depending on the LanguageLevel, PostScript implementations differ in the facilities they offer for rendering:

- LanguageLevel 2 supports CIE-based color rendering dictionaries to control the conversion of CIE-based colors to a device color space.
- LanguageLevel 3 supports the selection of color rendering dictionaries based on a *rendering intent* that expresses the program's priorities in rendering colors for a given output device.
- LanguageLevel 3 also supports the **UseCIEColor** page device parameter, which systematically remaps colors originally specified in a device color space into a corresponding CIE-based color space; see "Remapping Device Colors to CIE" on page 237.
- LanguageLevel 3 introduces an explicit *process color model* parameter that determines the device's native color space, and it includes a new family of native color spaces, **DeviceN** (see Section 6.2.5, "Color Support").
- Most LanguageLevel 1 implementations support only a single transfer function, controlled by the **settransfer** operator, and a single halftone function, controlled by the **setscreen** operator.
- LanguageLevel 1 implementations with the color extensions support multiple transfer functions controlled by **setcolortransfer**, multiple halftone functions controlled by **setcolorscreen**, and various color conversion facilities. These operators provide independent rendering control for each individual color component. LanguageLevel 1 products containing this feature also support the **setcmykcolor** and **colorimage** operators.
- LanguageLevel 2 also supports *halftone dictionaries* as a means of specifying halftone screen thresholds, transfer functions, and many other rendering details. Halftone dictionaries are more general and more flexible than the LanguageLevel 1 facilities, and they override those facilities when used. LanguageLevel 3 offers several additional types of halftone dictionary beyond those available in LanguageLevel 2. Of course, LanguageLevels 2 and 3 support all LanguageLevel 1 facilities.

Note: *Many of the rendering-related operators discussed in this chapter can install composite objects, such as arrays or dictionaries, as parameters in the graphics state. To ensure predictable behavior, a PostScript program should thereafter treat all such objects as if they were read-only.*

7.1 CIE-Based Color to Device Color

To render CIE-based colors on a device, the PostScript interpreter must convert from the specified CIE-based color space to the device's native color space, taking into account the known properties of the device. As discussed in Section 4.8.3, "CIE-Based Color Spaces," CIE-based color is based on a model of human color perception. The goal of CIE-based color rendering is to produce output in the device's native color space that accurately reproduces the requested CIE-based color values as perceived by a human observer. Typically, the native color space is one of the standard PostScript device color spaces (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**) and the conversion is performed by means of a CIE-based color rendering dictionary. CIE-based color specification and rendering are a feature of LanguageLevel 2 (**CIEBasedABC**, **CIEBasedA**) and LanguageLevel 3 (**CIEBasedDEF**, **CIEBasedDEFG**).

The conversion from CIE-based color to device color is complex, and the theory on which it is based is beyond the scope of this book; see the Bibliography for sources of further information. The algorithm has many parameters, including an optional, full three-dimensional color lookup table. The color fidelity of the output depends on having these parameters properly set, usually by a method that includes some form of calibration. Each product includes a default set of color rendering parameters that have been chosen to produce reasonable output based on the nominal characteristics of the device. The PostScript language does not prescribe methods for calibrating the device or for computing a proper set of color rendering parameters.

Conversion from a CIE-based color value to a device color value requires two main operations:

1. Adjust the CIE-based color value according to a *CIE-based gamut mapping function.* A *gamut* is a subset of all possible colors in some color space. A page description has a *source gamut* consisting of all the colors it uses. A device has a *device gamut* consisting of all the colors it can reproduce. This step transforms colors from the source gamut to the device gamut in a way that attempts to

preserve color appearance, visual contrast, or some other explicitly specified *rendering intent* (see Section 7.1.3, "Rendering Intents").

2. Generate a corresponding device color value according to a *CIE-based color mapping function*. For a given CIE-based color value, this function computes a color value in the device's native color space.

The CIE-based gamut and color mapping functions are applied only to color values presented in a CIE-based color space. By definition, color values in device color spaces directly control the device color components (though this can be altered by the **UseCIEColor** page device parameter; see "Remapping Device Colors to CIE" on page 237).

The source gamut is specified by a page description when it selects a CIE-based color space by invoking the **setcolorspace** operator. This specification, which includes the values defined by the **WhitePoint** and **BlackPoint** entries of the color space dictionary, is device-independent.

Together, the device gamut, the gamut mapping function, and the color mapping function are described by a *CIE-based color rendering dictionary*, a parameter of the graphics state that is set when the device is installed or recalibrated. Everything in this dictionary is device-dependent. The **setcolorrendering** operator installs a color rendering dictionary in the graphics state; **currentcolorrendering** returns the current color rendering dictionary.

7.1.1 CIE-Based Color Rendering Dictionaries

The CIE-based gamut and color mapping functions, embodied by the color rendering dictionary, are defined in an extensible way. The PostScript language supports one standard type of color rendering dictionary, which works in all implementations. Some output devices support additional types that define other, possibly proprietary, gamut and color mapping methods. The set of available types and the meanings of specific color rendering dictionaries are device-dependent; they are not described in this book, but in individual product documentation.

Most of the entries in a color rendering dictionary together define a composite color rendering function that transforms CIE-based color values to device color values by applying the gamut and color mapping functions. The output from this

color rendering function is subject to further transformations: device color space conversion, transfer function, and halftoning.

Every color rendering dictionary must have a **ColorRenderingType** entry whose value is an integer. The value specifies the architecture of the composite color rendering function as a whole. The remaining entries in the dictionary are interpreted according to this value.

7.1.2 Type 1 Color Rendering Dictionary

The type 1 color rendering dictionary is a standard part of the PostScript language. Some output devices support other types, and the default color rendering dictionary for any particular device may have a type other than 1.

Type 1 color rendering is based on the **CIEBasedABC** color space, which is a two-stage, nonlinear transformation of the CIE 1931 XYZ space. This space is called the *render color space.* Values in this space can be treated in one of two ways:

- Used directly as color values in the **DeviceRGB** or **DeviceGray** color space
- Used to index a three-dimensional lookup table, which in turn contains color values to be interpreted in the **DeviceRGB** or **DeviceCMYK** color space

The first method usually works well with additive, linear color devices, which include many black-and-white and color displays. The second method is required for high-fidelity reproductions with most color printers, whose color rendition cannot be described by a simple formula.

Conceptually, conversion of a color value from a CIE-based color space to a device color space involves the following steps. In practice, the implementation does not perform these steps in sequence, but in combination. Furthermore, there are important special cases in which the effects of two or more of the steps cancel out. The implementation detects these cases and omits the unnecessary transformations.

1. Transform the CIE-based color value from its original color space (**CIEBasedABC**, **CIEBasedA**, **CIEBasedDEF**, or **CIEBasedDEFG**) to the CIE 1931 XYZ space. This transformation depends on various parameters of the color space, as described in Section 4.8.3, "CIE-Based Color Spaces."

2. Adjust the *X*, *Y*, and *Z* values to account for differences in the diffuse white and black points of the source and the device. This transformation attempts to satisfy the requested rendering intent, according to the **MatrixPQR** and **TransformPQR** entries of the color rendering dictionary. The diffuse white and black points of the source are given by the **WhitePoint** and **BlackPoint** entries in the color space dictionary; those of the device are given by the same entries in the color rendering dictionary. If the corresponding **WhitePoint** and **BlackPoint** entries in both dictionaries are equal, the **MatrixPQR** and **TransformPQR** entries are ignored and this step reduces to the identity transformation.

3. Transform the color value from the CIE 1931 XYZ space into the render color space according to the **EncodeLMN**, **MatrixLMN**, **EncodeABC**, and **MatrixABC** entries in the CIE-based color rendering dictionary, producing three components *A*, *B*, and *C*. (These are unrelated to the *A*, *B*, and *C* components of color values in the source **CIEBasedABC** or **CIEBasedA** color space.)

4. If a **RenderTable** entry is present in the color rendering dictionary, use the *A*, *B*, and *C* components to index into this three-dimensional lookup table, yielding an interpolated color value. This value consists of three or four color components, depending on how the table is defined. Each component is then transformed by a procedure to produce color components in device color space. If there are three components, they specify red, green, and blue values in the **DeviceRGB** color space; if there are four components, they specify cyan, magenta, yellow, and black in the **DeviceCMYK** color space.

   If the color rendering dictionary has no **RenderTable** entry, use the *A*, *B*, and *C* components as the device color value directly. If the device's native color space is **DeviceGray**, the *A* component specifies the gray value and the *B* and *C* components are ignored. Otherwise, the *A*, *B*, and *C* components specify the red, green, and blue values, respectively, in the **DeviceRGB** color space.

If the output of the color rendering dictionary is in a color space different from the device's native color space, a further conversion will take place, as described in Section 7.2, "Conversions among Device Color Spaces." Note that such a conversion will always occur when the native color space is a **DeviceN** space.

Table 7.1 describes the entries that a type 1 color rendering dictionary can contain and describes the details of the transformations.

TABLE 7.1 Entries in a type 1 CIE-based color rendering dictionary

KEY	TYPE	VALUE
ColorRenderingType	integer	*(Required)* A code identifying the type of color rendering that this dictionary describes; must be 1 for a standard color rendering dictionary.
MatrixLMN	array	*(Optional)* An array of nine numbers [$L_X\ M_X\ N_X\ L_Y\ M_Y\ N_Y\ L_Z\ M_Z\ N_Z$] specifying the linear interpretation of the *X, Y,* and *Z* components of the CIE 1931 XYZ space with respect to an intermediate LMN representation; see **EncodeLMN** below for further explanation. Default value: the identity matrix [1 0 0 0 1 0 0 0 1].
EncodeLMN	array	*(Optional)* An array of three PostScript procedures [$E_L\ E_M\ E_N$] that encode the *L, M,* and *N* components of the intermediate representation. Default value: the array of identity procedures [{} {} {}]. Each of the three procedures is called with an unencoded *L, M,* or *N* component on the operand stack and must return the corresponding encoded value. The result must be a monotonically nondecreasing function of the operand. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeLMN** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects. The transformation defined by the **MatrixLMN** and **EncodeLMN** entries is $L = E_L(X \times L_X + Y \times L_Y + Z \times L_Z)$ $M = E_M(X \times M_X + Y \times M_Y + Z \times M_Z)$ $N = E_N(X \times N_X + Y \times N_Y + Z \times N_Z)$ In other words, the *X, Y,* and *Z* components of the CIE 1931 XYZ space are treated as a three-element vector and multiplied by **MatrixLMN** (a 3-by-3 matrix). The results are then individually transformed by the **EncodeLMN** procedures to obtain the *L, M,* and *N* components of the intermediate representation.
RangeLMN	array	*(Optional)* An array of six numbers [$L_0\ L_1\ M_0\ M_1\ N_0\ N_1$] specifying the range of valid values for the *L, M,* and *N* components of the intermediate representation: that is, $L_0 \le L \le L_1$, $M_0 \le M \le M_1$, and $N_0 \le N \le N_1$. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].
MatrixABC	array	*(Optional)* An array of nine numbers [$A_L\ B_L\ C_L\ A_M\ B_M\ C_M\ A_N\ B_N\ C_N$] specifying the linear interpretation of the encoded *L, M,* and *N* components of the intermediate representation with respect to the render color space; see

EncodeABC below for further explanation. Default value: the identity matrix [1 0 0 0 1 0 0 0 1].

EncodeABC array *(Optional)* An array of three PostScript procedures $[E_A\ E_B\ E_C]$ that encode the *A*, *B*, and *C* components of the color space. Default value: the array of identity procedures [{} {} {}].

Each of the three procedures is called with an unencoded *A*, *B*, or *C* component on the operand stack and must return the corresponding encoded value. The result must be a monotonically nondecreasing function of the operand. The procedures must be prepared to accept operand values outside the ranges specified by the **RangeABC** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.

The transformation defined by the **MatrixABC** and **EncodeABC** entries is

$$\begin{aligned} A &= E_A\ (L \times A_L + M \times A_M + N \times A_N) \\ B &= E_B\ (L \times B_L + M \times B_M + N \times B_N) \\ C &= E_C\ (L \times C_L + M \times C_M + N \times C_N) \end{aligned}$$

In other words, the *L*, *M*, and *N* components of the intermediate representation are treated as a three-element vector and multiplied by **MatrixABC** (a 3-by-3 matrix). The results are then individually transformed by the **EncodeABC** procedures to obtain the *A*, *B*, and *C* components of the render color space.

RangeABC array *(Optional)* An array of six numbers $[A_0\ A_1\ B_0\ B_1\ C_0\ C_1]$ specifying the range of valid values for the *A*, *B*, and *C* components: that is, $A_0 \le A \le A_1$, $B_0 \le B \le B_1$, and $C_0 \le C \le C_1$. If there is no **RenderTable** entry, these ranges must lie within the range 0.0 to 1.0, since the render color space maps directly onto a device color space. If a **RenderTable** entry is present, these ranges define the boundaries of the three-dimensional lookup table. Default value: [0.0 1.0 0.0 1.0 0.0 1.0].

WhitePoint array *(Required)* An array of three numbers $[X_W\ Y_W\ Z_W]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the device's diffuse white point. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.

WhitePoint is assumed to represent the device's diffuse achromatic highlight, and hence its value must correspond to the nearly lightest achromatic color that the device can produce. A color somewhat darker than the absolutely lightest color may be used to avoid blocking of highlights and to provide some flexibility for rendering specular highlights.

BlackPoint array *(Optional)* An array of three numbers $[X_B\ Y_B\ Z_B]$ specifying the tristimulus value, in the CIE 1931 XYZ space, of the device's diffuse black point. All three of these numbers must be nonnegative. Default value: `[0 0 0]`.

BlackPoint is assumed to represent the device's diffuse achromatic shadow. Its value is defined by the nearly darkest, nearly achromatic color that the device can produce. A color somewhat lighter than the absolutely darkest color may be used to avoid blocking of shadows; a slightly chromatic color may be used to increase dynamic range in situations where the darkest color that the device can produce is not purely achromatic.

MatrixPQR array *(Optional)* An array of nine numbers $[P_X\ Q_X\ R_X\ P_Y\ Q_Y\ R_Y\ P_Z\ Q_Z\ R_Z]$ specifying the linear interpretation of the *X, Y,* and *Z* components of the CIE 1931 XYZ space with respect to an intermediate PQR representation; see **TransformPQR** below for further explanation. Default value: the identity matrix `[1 0 0 0 1 0 0 0 1]`.

RangePQR array *(Optional)* An array of six numbers $[P_0\ P_1\ Q_0\ Q_1\ R_0\ R_1]$ specifying the range of valid values for the *P, Q,* and *R* components of the intermediate representation: that is, $P_0 \le P \le P_1$, $Q_0 \le Q \le Q_1$, and $R_0 \le R \le R_1$. Default value: `[0.0 1.0 0.0 1.0 0.0 1.0]`.

TransformPQR array *(Required)* An array of three PostScript procedures $[T_P\ T_Q\ T_R]$ that transform the *P, Q,* and *R* components of the intermediate representation to compensate for the differences between the source's and the device's diffuse white and black points, while observing the requested rendering intent.

Let X_{Ws}, Y_{Ws}, Z_{Ws} and X_{Bs}, Y_{Bs}, Z_{Bs} be the tristimulus values, in the CIE 1931 XYZ space, of the source's diffuse white and black points, respectively, and let X_{Wd}, Y_{Wd}, Z_{Wd} and X_{Bd}, Y_{Bd}, Z_{Bd} be those of the device's diffuse white and black points. Then the source and device tristimulus values X_s, Y_s, Z_s and X_d, Y_d, Z_d are related by the **MatrixPQR** and **TransformPQR** entries as follows:

$$\begin{aligned} P_s &= X_s \times P_X + Y_s \times P_Y + Z_s \times P_Z \\ Q_s &= X_s \times Q_X + Y_s \times Q_Y + Z_s \times Q_Z \\ R_s &= X_s \times R_X + Y_s \times R_Y + Z_s \times R_Z \end{aligned}$$

$$\begin{aligned} P_d &= T_P(W_s, B_s, W_d, B_d, P_s) \\ Q_d &= T_Q(W_s, B_s, W_d, B_d, Q_s) \\ R_d &= T_R(W_s, B_s, W_d, B_d, R_s) \end{aligned}$$

$$\begin{aligned} X_d &= P_d \times X_P + Q_d \times X_Q + R_d \times X_R \\ Y_d &= P_d \times Y_P + Q_d \times Y_Q + R_d \times Y_R \\ Z_d &= P_d \times Z_P + Q_d \times Z_Q + R_d \times Z_R \end{aligned}$$

where

$$
\begin{aligned}
W_s &= [X_{Ws}\, Y_{Ws}\, Z_{Ws}\, P_{Ws}\, Q_{Ws}\, R_{Ws}] \\
B_s &= [X_{Bs}\, Y_{Bs}\, Z_{Bs}\, P_{Bs}\, Q_{Bs}\, R_{Bs}] \\
W_d &= [X_{Wd}\, Y_{Wd}\, Z_{Wd}\, P_{Wd}\, Q_{Wd}\, R_{Wd}] \\
B_d &= [X_{Bd}\, Y_{Bd}\, Z_{Bd}\, P_{Bd}\, Q_{Bd}\, R_{Bd}]
\end{aligned}
$$

$$
\begin{aligned}
P_{Ws} &= X_{Ws} \times P_X + Y_{Ws} \times P_Y + Z_{Ws} \times P_Z \\
Q_{Ws} &= X_{Ws} \times Q_X + Y_{Ws} \times Q_Y + Z_{Ws} \times Q_Z \\
R_{Ws} &= X_{Ws} \times R_X + Y_{Ws} \times R_Y + Z_{Ws} \times R_Z
\end{aligned}
$$

$$
\begin{aligned}
P_{Bs} &= X_{Bs} \times P_X + Y_{Bs} \times P_Y + Z_{Bs} \times P_Z \\
Q_{Bs} &= X_{Bs} \times Q_X + Y_{Bs} \times Q_Y + Z_{Bs} \times Q_Z \\
R_{Bs} &= X_{Bs} \times R_X + Y_{Bs} \times R_Y + Z_{Bs} \times R_Z
\end{aligned}
$$

$$
\begin{aligned}
P_{Wd} &= X_{Wd} \times P_X + Y_{Wd} \times P_Y + Z_{Wd} \times P_Z \\
Q_{Wd} &= X_{Wd} \times Q_X + Y_{Wd} \times Q_Y + Z_{Wd} \times Q_Z \\
R_{Wd} &= X_{Wd} \times R_X + Y_{Wd} \times R_Y + Z_{Wd} \times R_Z
\end{aligned}
$$

$$
\begin{aligned}
P_{Bd} &= X_{Bd} \times P_X + Y_{Bd} \times P_Y + Z_{Bd} \times P_Z \\
Q_{Bd} &= X_{Bd} \times Q_X + Y_{Bd} \times Q_Y + Z_{Bd} \times Q_Z \\
R_{Bd} &= X_{Bd} \times R_X + Y_{Bd} \times R_Y + Z_{Bd} \times R_Z
\end{aligned}
$$

$$
\begin{bmatrix} X_P & Y_P & Z_P \\ X_Q & Y_Q & Z_Q \\ X_R & Y_R & Z_R \end{bmatrix} = \begin{bmatrix} P_X & Q_X & R_X \\ P_Y & Q_Y & R_Y \\ P_Z & Q_Z & R_Z \end{bmatrix}^{-1}
$$

In other words, the X_s, Y_s, and Z_s components of the source color in CIE 1931 XYZ space are treated as a three-element vector and multiplied by **MatrixPQR** (a 3-by-3 matrix), yielding the P_s, Q_s, and R_s components of the source color with respect to the intermediate PQR representation. These components are then individually transformed by the **TransformPQR** procedures to obtain the P_d, Q_d, and R_d components of the corresponding device color. Each of the components is transformed separately; there is no interaction between components. Finally, the P_d, Q_d, and R_d components of the device color are treated as a three-element vector and multiplied by the inverse of **MatrixPQR**, yielding the X_d, Y_d, and Z_d components of the device color in the CIE 1931 XYZ space.

The transformation embodied by the **TransformPQR** procedures typically consists of two conceptually separate processes. The first allows for chromatic adaptation when the two diffuse white points differ, the second for contrast

adaptation when the dynamic ranges between the two sets of diffuse white and black points differ.

Besides the appropriate P_s, Q_s, or R_s component, each procedure takes four additional operands, W_s, B_s, W_d, and B_d, which specify the diffuse white and black points of the source and the device, respectively. Each of these operands is an array of six elements giving the white or black point twice: once in CIE 1931 XYZ space and again in PQR space.

Each of these three procedures is called with the W_s, B_s, W_d, and B_d arrays and the appropriate P_s, Q_s, or R_s component on the operand stack, in that order, and must return the corresponding transformed P_d, Q_d, or R_d component. The result must be a monotonically nondecreasing function of the last operand. The procedures must be prepared to accept operand values outside the ranges specified by the **RangePQR** entry and to deal with such values in a robust way. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.

RenderTable array *(Optional)* An array of the form [N_A N_B N_C *table m* T_1 T_2 … T_m] defining a three-dimensional lookup table that maps colors in render color space into device color space.

The first three elements of the **RenderTable** array, N_A, N_B, and N_C, must be integers greater than 1 specifying the dimensions of the lookup table. The table contains $N_A \times N_B \times N_C$ entries, each consisting of m encoded color components making up a device color value (where m, the fifth element of the **RenderTable** array, must be the integer 3 or 4).

The fourth array element, *table*, holds the contents of the lookup table itself in the form of an array of N_A strings, each containing $m \times N_B \times N_C$ bytes. Like all PostScript arrays, the *table* array is indexed from 0; index a thus varies from 0 to $N_A - 1$, index b from 0 to $N_B - 1$, and index c from 0 to $N_C - 1$. The table entry for coordinates (a, b, c) is found in the string at index a in the lookup table, in the m bytes starting at position $m \times (b \times N_C + c)$. This entry corresponds to the following color value in the render color space:

$$\begin{aligned} A &= A_0 + a \times (A_1 - A_0)/(N_A - 1) \\ B &= B_0 + b \times (B_1 - B_0)/(N_B - 1) \\ C &= C_0 + c \times (C_1 - C_0)/(N_C - 1) \end{aligned}$$

The limiting values A_0, A_1, B_0, B_1, C_0, and C_1 are specified by the color rendering dictionary's **RangeABC** entry. The m encoded device color component values $e_1, e_2, \ldots, e_m$ corresponding to a given color in ABC space are computed by locating the nearest adjacent table entries and then interpolating among the encoded byte values contained in those entries.

The elements $T_1, T_2, \ldots, T_m$ are PostScript procedures that transform the interpolated, encoded components to device color components. These transformations are

$$\begin{aligned} d_1 &= T_1(e_1/255) \\ d_2 &= T_2(e_2/255) \\ &\vdots \\ d_m &= T_m(e_m/255) \end{aligned}$$

In other words, the interpreter divides each encoded component by 255, producing a number in the range 0.0 to 1.0, and pushes it on the operand stack. It then calls the corresponding *T* procedure, which is expected to consume its operand and produce a result in the range 0.0 to 1.0. Because these procedures are called at unpredictable times and in unpredictable environments, they must operate as pure functions without side effects.

The values $d_1, d_2, \ldots, d_m$ constitute the components of the final device color value. That is, if m is 3, then d_1, d_2, and d_3 are the red, green, and blue components; if m is 4, then d_1, d_2, d_3, and d_4 are the cyan, magenta, yellow, and black components.

CreationDate string *(Optional; LanguageLevel 3)* The date and time at which the color rendering dictionary was created or most recently modified. The value of this entry should be coordinated with the **calibrationDateTimeTag** attribute of any associated ICC profile, and its syntax should conform to the international standard ASN.1, defined in the document ITU X.208 or ISO/IEC 8824. (ICC profiles are a standard means, defined by an organization called the International Color Consortium, for describing the color properties of a device; they can be translated in a straightforward way to or from a PostScript rendering dictionary.) This entry is included for information only; the PostScript interpreter does not consult it.

RenderingIntent name or string *(Optional; LanguageLevel 3)* The rendering intent that this color rendering dictionary is designed to achieve (see Section 7.1.3, "Rendering Intents"). This entry is included for information only; the PostScript interpreter does not consult it.

7.1.3 Rendering Intents

Although CIE-based color specifications are theoretically device-independent, they are subject to practical limitations in the color reproduction capabilities of the output device. Such limitations may sometimes require compromises to be made among various properties of a color specification when rendering colors for a given device. Specifying a *rendering intent (LanguageLevel 3)* allows a PostScript program to set priorities regarding which of these properties to preserve and which to sacrifice. For example, the program might request that in-gamut colors be reproduced exactly while sacrificing the accuracy of out-of-gamut colors, or that a scanned image such as a photograph be rendered in a perceptually "pleasing" manner at the cost of strict colorimetric accuracy.

Table 7.2 lists the standard rendering intents recognized in the initial release of LanguageLevel 3 interpreters from Adobe Systems. These have been deliberately chosen to correspond closely with the rendering intents defined by the International Color Consortium (ICC), an industry organization that has developed standards for device-independent color. Note, however, that the exact set of rendering intents supported may vary from one output device to another; a particular device may not support all possible intents, or may support additional ones beyond those listed in the table. As discussed below, the PostScript interpreter provides reasonable default behavior even for rendering intents that it does not recognize.

Rendering intents are specified with the **findcolorrendering** operator. This operator accepts a single operand, a name or string identifying the desired rendering intent. The operator uses this information, along with the current values of the page device and the current halftone, to select a suitable color rendering dictionary (CRD) for rendering colors on the device according to the requested intent. If no appropriate CRD is available, the operator proposes an alternate CRD instead. In either case, it returns the name of the CRD on the stack, along with a boolean value indicating whether the CRD satisfies the requested rendering intent or is merely an alternate. The CRD name can then be passed to the **findresource** operator to find the corresponding dictionary in the **ColorRendering** resource category. See the description of the **findcolorrendering** operator in Chapter 8 for further details.

TABLE 7.2 Rendering intents

RENDERING INTENT	DESCRIPTION
AbsoluteColorimetric	Colors are represented solely with respect to the light source; no correction is made for the medium's white point (such as the color of unprinted paper). Thus, for example, a monitor's white point, which is bluish compared to that of a printer's paper, would be reproduced with a blue cast. In-gamut colors are reproduced exactly; out-of-gamut colors are mapped to the nearest value within the reproducible gamut. This style of reproduction has the advantage of providing exact color matches from one output medium to another. It has the disadvantage of causing colors with *Y* values between the medium's white point and 1.0 to be out of gamut. A typical use might be for logos and solid colors that require exact reproduction across different media.
RelativeColorimetric	Colors are represented with respect to the combination of the light source and the medium's white point (such as the color of unprinted paper). Thus, for example, a monitor's white point would be reproduced on a printer by simply leaving the paper unprinted, ignoring color differences between the two media. In-gamut colors are reproduced exactly; out-of-gamut colors are mapped to the nearest value within the reproducible gamut. This style of reproduction has the advantage of adapting for the varying white points of different output media. It has the disadvantage of not providing exact color matches from one medium to another. A typical use might be for vector graphics.
Saturation	Colors are represented in a manner that preserves or emphasizes saturation. Reproduction of in-gamut colors may or may not be colorimetrically accurate. A typical use might be for business graphics, where saturation is the most important attribute of the color.
Perceptual	Colors are represented in a manner that provides a pleasing perceptual appearance. This generally means that both in-gamut and out-of-gamut colors are modified from their precise colorimetric values in order to preserve color relationships. A typical use might be for scanned images.

Note: *The only graphics state parameters considered by the* ***findcolorrendering*** *operator are the current device and halftone. Other parameters of the graphics state, such as black-generation, undercolor-removal, and transfer functions, are not taken into account, since (unlike halftoning) they depend only on the output device and not on the particular graphical element being rendered.*

To ensure that all relevant parameters are correctly accounted for, the invocation of the **findcolorrendering** operator should follow any other operations that may influence either the halftone or the device configuration. Example 7.1 illustrates the use of **findcolorrendering** to request the **Perceptual** rendering intent.

Example 7.1

```
/Perceptual findcolorrendering
        {   % CRD found that satisfies combination of rendering intent,
            % device configuration, and halftone

            /ColorRendering findresource setcolorrendering
        }
        {   % Exact match for CRD not found. Use it or find a CRD another way.
            % In this example, we'll use it if it's not DefaultColorRendering.

            dup /DefaultColorRendering eq
                    {pop}
                    {/ColorRendering findresource setcolorrendering}
                ifelse
        }
    ifelse
```

The example first uses the **findcolorrendering** operator to request a color rendering dictionary for the **Perceptual** rendering intent. If successful, **findcolorrendering** returns the name of an appropriate CRD for this intent on the stack, along with a boolean result of *true;* the example then proceeds to retrieve the CRD with **findresource** and install it in the graphics state with **setcolorrendering**.

If the boolean result from **findcolorrendering** is *false*, then no appropriate color rendering dictionary could be found and the name returned on the stack is merely a proposed alternate. In this case, three actions are possible:

- Use the alternate CRD name that is returned.
- Select a different CRD using your own method.
- Leave the graphics state's current CRD installed.

If **findcolorrendering** returns *false*, the code in Example 7.1 first tests whether the CRD returned is equal to the default CRD, **DefaultColorRendering**. In general, this signifies that no useful substitution is possible to achieve the requested rendering intent. In this case, the example simply leaves the current CRD installed in the graphics state. If **findcolorrendering** returns an alternate CRD other than **DefaultColorRendering**, the example uses **findresource** and **setcolorrendering** to retrieve it and install it in the graphics state.

Customizing CRD Selection

Although the **findcolorrendering** operator itself is not meant to be overridden, it delegates portions of its task to the procedures **GetPageDeviceName**, **GetHalftoneName**, and **GetSubstituteCRD**, which *can* be overridden. These are not operators residing in **systemdict**, but procedures defined in the **ColorRendering** procedure set (an instance of the **ProcSet** resource category). Adobe supplies baseline versions that satisfy the stated requirements for the three procedures and serve as templates for customizing them to a particular output device. See the descriptions of these procedures in Chapter 8 for further information.

The **findcolorrendering** operator forms the name of a color rendering dictionary from the requested rendering intent, the current device configuration, and the halftone. The resulting name takes the form

renderingintent.deviceconfig.halftone

The *renderingintent* portion of the name is taken verbatim from the operand supplied on the stack by the PostScript program; *deviceconfig* and *halftone* are found by calling the **GetPageDeviceName** and **GetHalftoneName** procedures, respectively. If no CRD with this name exists in the **ColorRendering** resource category, **findcolorrendering** then calls the **GetSubstituteCRD** procedure to supply the name of an alternate CRD.

In general, the **GetPageDeviceName** procedure first looks in the page device dictionary for a **PageDeviceName** entry. If this entry is found, the procedure returns its value. If the entry is not present or if its value is *null*, the procedure may construct a name for the device configuration from the current page device parameters—for example, **MediaType** or **MediaClass**—or may simply return the name **none**. A particular device can override this default behavior by replacing the definition of the **GetPageDeviceName** procedure.

Similarly, the **GetHalftoneName** procedure looks in the current halftone dictionary for a **HalftoneName** entry and returns its value if found. If the entry is not present, **GetHalftoneName** may analyze the current halftone and attempt to form a name or may simply return none. Again, a device can customize the behavior of the **GetHalftoneName** procedure by overriding its definition.

The behavior of the default **GetSubstituteCRD** procedure is device-dependent. As a last resort, **GetSubstituteCRD** returns the name of some built-in color rendering dictionary, such as **DefaultColorRendering**.

7.2 Conversions among Device Color Spaces

Each raster output device has a *native color space*, which typically is one of the standard device color spaces (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**). In other words, most devices support reproduction of colors according to a grayscale (monochrome), red-green-blue, or cyan-magenta-yellow-black model. If the device supports continuous-tone output, reproduction occurs directly. Otherwise, it is accomplished by means of halftoning.

Note: *In LanguageLevel 3, a device's native color space may be a* ***DeviceN*** *space rather than one of the standard device color spaces. See Section 7.2.5, "Conversion to DeviceN Spaces," for details on the color conversion process in this case.*

A device's native color space is determined by the **ProcessColorModel** entry in its page device dictionary (see Section 6.2.5, "Color Support"). Knowing the native color space and other output capabilities of the device, the PostScript interpreter can automatically convert the color values specified in a PostScript program to those appropriate for the device's native color space. For example, if a PostScript program specifies colors in the **DeviceRGB** color space, but the device supports grayscale (such as a monochrome display) or CMYK (such as a color printer), the interpreter performs the necessary conversions. If the program specifies colors directly in the device's native color space, no conversions are necessary.

A program can also request explicit conversions among device color spaces by invoking the operators **currentgray**, **currentrgbcolor**, **currenthsbcolor**, or **currentcmykcolor**, which return color values according to specific color spaces. These operators are described in Section 4.8.2, "Device Color Spaces." All PostScript implementations support conversions between **DeviceRGB** and **DeviceGray**; conversions to and from **DeviceCMYK** are a LanguageLevel 2 feature (also available in LanguageLevel 1 implementations with the color extensions).

Note: *These operators can convert colors only among device color spaces, not to or from CIE-based or special color spaces. If the current color space is an* ***Indexed*** *color space or is a* ***Separation*** *or* ***DeviceN*** *color space with its alternative color space selected, these operators are applied to the underlying color space.*

The conversions described here do not involve the use of transfer functions or halftone functions. When colors are to be rendered on the output device, the transfer functions and halftone functions are applied at a later stage to the *output* of the color conversion operation. When colors are simply read back by a PostScript program with one of the operators mentioned above, transfer functions and halftone functions are not applied at all.

The algorithms used to convert among device color spaces are very simple. As perceived by a human viewer, the conversions produce only crude approximations of the original colors. More sophisticated control over color conversion can be achieved by means of CIE-based color specification and rendering. Additionally, device color spaces can be remapped into CIE-based color spaces (see "Remapping Device Colors to CIE" on page 237).

7.2.1 Conversion between DeviceRGB and DeviceGray

Black, white, and intermediate shades of gray can be considered special cases of RGB color. A grayscale value is described by a single number: 0.0 corresponds to black, 1.0 to white, and intermediate values to different gray levels.

A gray level is equivalent to an RGB value with all three components the same. In other words, the RGB color value equivalent to a specific gray value is simply

$$\begin{aligned} red &= gray \\ green &= gray \\ blue &= gray \end{aligned}$$

The gray value for a given RGB value is computed according to the NTSC video standard. This standard determines how a color television signal is rendered on a black-and-white television set.

$$gray = 0.3 \times red + 0.59 \times green + 0.11 \times blue$$

Colors specified according to the HSB (hue-saturation-brightness) model are equivalent to those specified in the RGB model, but expressed in a different coordinate system called the *hexcone* model; see the Bibliography for sources of fur-

ther information. Either form of specification produces colors in the **DeviceRGB** color space; HSB is not a color space in its own right.

7.2.2 Conversion between DeviceCMYK and DeviceGray

Nominally, a gray level is the complement of the black component of CMYK. Therefore, the CMYK color value equivalent to a specific gray level is simply

$$\begin{aligned} cyan &= 0.0 \\ magenta &= 0.0 \\ yellow &= 0.0 \\ black &= 1.0 - gray \end{aligned}$$

To obtain the equivalent gray level for a given CMYK value, the contributions of all components must be taken into account:

$$gray = 1.0 - \min(1.0, 0.3 \times cyan + 0.59 \times magenta + 0.11 \times yellow + black)$$

The interactions between the black component and the other three are elaborated below.

7.2.3 Conversion from DeviceRGB to DeviceCMYK

Conversion of a color value from RGB to CMYK is a two-step process. The first step is to convert the red-green-blue value to equivalent cyan, magenta, and yellow components. The second step is to generate a black component and alter the other components to produce a better approximation of the original color.

The subtractive color primaries *cyan*, *magenta*, and *yellow* are the complements of the additive primaries *red*, *green*, and *blue*. For example, a cyan ink subtracts the red component of white light. In theory, the conversion is very simple:

$$\begin{aligned} cyan &= 1.0 - red \\ magenta &= 1.0 - green \\ yellow &= 1.0 - blue \end{aligned}$$

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.

Logically, only cyan, magenta, and yellow are needed to generate a printing color. An equal level of cyan, magenta, and yellow should create the equivalent level of black.

In practice, colored printing inks do not mix perfectly; such combinations often form dark brown shades instead of true black. To obtain a truer color rendition on a printer, it is often desirable to substitute true black ink for the mixed-black portion of a color. Most color printers support a black component (the *K* component of CMYK). Computing the quantity of this component requires some additional steps:

1. *Black generation* calculates the amount of black to be used when trying to reproduce a particular color.
2. *Undercolor removal* reduces the amounts of the cyan, magenta, and yellow components to compensate for the amount of black that was added by black generation.

The complete conversion from RGB to CMYK is as follows, where $BG(k)$ and $UCR(k)$ are invocations of the black-generation and undercolor-removal functions, respectively:

$$\begin{aligned} c &= 1.0 - red \\ m &= 1.0 - green \\ y &= 1.0 - blue \\ k &= \min(c, m, y) \end{aligned}$$

$$\begin{aligned} cyan &= \min(1.0, \max(0.0, c - UCR(k))) \\ magenta &= \min(1.0, \max(0.0, m - UCR(k))) \\ yellow &= \min(1.0, \max(0.0, y - UCR(k))) \\ black &= \min(1.0, \max(0.0, BG(k))) \end{aligned}$$

The black-generation and undercolor-removal functions are defined as PostScript procedures. The **setblackgeneration** and **setundercolorremoval** operators set these parameters in the graphics state. The interpreter calls these procedures when it needs to perform RGB-to-CMYK conversion. Each procedure is called with a single numeric operand and is expected to return a single numeric result. Because these procedures are called at unpredictable times, they must operate as pure functions without side effects.

The operand of both procedures is *k*, the minimum of the intermediate *c*, *m*, and *y* values that have been computed by subtracting the original *red*, *green*, and *blue*

components from 1.0. Nominally, *k* is the amount of black that can be removed from the cyan, magenta, and yellow components and be substituted as a separate black component.

The black-generation function computes the black component as a function of the nominal *k* value. It can simply return its *k* operand unchanged, or it can return a larger value for extra black, a smaller value for less black, or 0.0 for no black at all.

The undercolor-removal function computes the amount to subtract from each of the intermediate *c*, *m*, and *y* values to produce the final cyan, magenta, and yellow components. It can simply return its *k* operand unchanged, or it can return 0.0 (so no color is removed), some fraction of the black amount, or even a negative amount, thereby adding to the total amount of ink.

The final component values that result after applying black generation and undercolor removal are expected to be in the range 0.0 to 1.0. If a value falls outside this range, the nearest valid value is substituted automatically, without error indication. This is indicated explicitly by invocations of *min* and *max* operations in the formulas given above.

The correct choice of black-generation and undercolor-removal functions depends on the characteristics of the output device—for example, how inks mix. Each device is configured with default values that are appropriate for that device.

7.2.4 Conversion from DeviceCMYK to DeviceRGB

Conversion of a color value from CMYK to RGB is a simple operation that does not involve black generation or undercolor removal:

$$\begin{aligned} red &= 1.0 - \min(1.0, cyan + black) \\ green &= 1.0 - \min(1.0, magenta + black) \\ blue &= 1.0 - \min(1.0, yellow + black) \end{aligned}$$

In other words, the black component is simply added to each of the other components, which are then converted to their complementary colors by subtracting them each from 1.0.

7.2.5 Conversion to DeviceN Spaces

In LanguageLevel 3, a device's native color space can be a **DeviceN** space. The set of process colorants used to render such a color space is device-dependent. The **SeparationColorNames** entry in the page device dictionary (see Section 6.2.5, "Color Support") lists the possible device colorants from which they are drawn.

Conversions from **DeviceRGB**, **DeviceCMYK**, or **DeviceGray** to a **DeviceN** space are performed by device-dependent means, which generally are not under PostScript program control. Some devices may use the defined undercolor-removal and black-generation functions when converting from **DeviceRGB** to a four-component **DeviceN** color space.

Note: *There is no corresponding conversion from* ***DeviceN*** *to standard device color spaces. If a PostScript program invokes* ***setcolorspace*** *to specify a* ***DeviceN*** *color space, all components of that color space will map directly to correspondingly named device colorants, regardless of the native color space of the device. If the device does not support all of those colorants (as indicated in* ***SeparationColorNames****), the* ***DeviceN*** *color space will revert to its alternative space (see "DeviceN Color Spaces" on page 245).*

7.3 Transfer Functions

A *transfer function* adjusts the values of color components to compensate for nonlinear response in an output device and in the human eye. Each component of a device color space—for example, the red component of the **DeviceRGB** space—is intended to represent the perceived lightness or intensity of that color component in proportion to the component's numeric value. Many devices do not actually behave this way, however; the purpose of a transfer function is to compensate for the device's actual behavior. This operation is sometimes called *gamma correction* (not to be confused with the *CIE-based gamut mapping function* performed as part of CIE-based color rendering).

In the sequence of steps for processing colors, the PostScript interpreter applies the transfer function *after* performing any needed conversions between color spaces, but *before* applying a halftone function, if necessary. Each color component has its own separate transfer function; there is no interaction between components.

Transfer functions always operate in the *native* color space of the output device, regardless of the color space in which colors were originally specified. For example, for a CMYK device, the transfer functions apply to the device's cyan, magenta, yellow, and black color components, even if the colors were originally specified in, say, the **DeviceRGB** or **CIEBasedABC** color space.

There are three ways to specify transfer functions:

- The **settransfer** operator establishes a single transfer function to be applied to all color components. Most LanguageLevel 1 implementations support only a single transfer function.
- The **setcolortransfer** operator establishes four separate transfer functions, one each for red, green, blue, and gray or their complements cyan, magenta, yellow, and black. An RGB device uses the first three; a monochrome device uses the gray transfer function only; and a CMYK device uses all four. **setcolortransfer** is supported in LanguageLevel 2 and in some LanguageLevel 1 implementations, primarily those in color printers.
- The **sethalftone** operator can establish transfer functions as optional entries in *halftone dictionaries* (see Section 7.4.3, "Halftone Dictionaries"). This is the only way to set transfer functions for nonprimary color components, or for any component in devices whose native color space is a **DeviceN** space. Transfer functions specified in halftone dictionaries override those specified by **settransfer** or **setcolortransfer**. Halftone dictionaries are a LanguageLevel 2 feature.

A transfer function is a PostScript procedure that can be called with a numeric operand in the range 0.0 to 1.0 on the operand stack and returns a number in the same range. The operand is the value of a color component in the output device's native color space, either specified directly or produced by conversion from some other color space. The procedure's result is the transformed component value to be transmitted to the device (after halftoning, if necessary). Both the operand and the result are always interpreted as if the color component were additive (red, green, blue, or gray): the greater the numeric value, the lighter the color. If the component is subtractive (cyan, magenta, yellow, black, or a spot color), the PostScript interpreter converts the operand to additive form by subtracting it from 1.0 before passing it to the transfer function. The result returned by the transfer function is always in additive form, and is passed on to the halftone function in that form.

Because the transfer function is called at unpredictable times and in unpredictable environments, it must operate as a pure function: it must not depend on variable data other than its operand, and must have no side effects.

In addition to their intended use for gamma correction, transfer functions can be used to produce a variety of special, device-dependent effects. For example, on a monochrome device, the transfer function

```
{1 exch sub}
```

inverts the output colors, producing a negative rendition of the page. In general, this method does not work for color devices; inversion can be more complicated than merely inverting each of the components. Because transfer functions produce device-dependent effects, a page description that is intended to be device-independent should not alter them.

Note: *When the current color space is* ***DeviceGray*** *and the output device's native color space is* ***DeviceCMYK****, the interpreter uses only the gray transfer function. The normal conversion from* ***DeviceGray*** *to* ***DeviceCMYK*** *produces 0.0 for the cyan, magenta, and yellow components. These components are not passed through their respective transfer functions, but are rendered directly, producing output containing no colored inks. This special case exists for compatibility with existing applications that use* ***settransfer*** *to obtain special effects on monochrome devices, and applies only to colors specified in the* ***DeviceGray*** *color space.*

7.4 Halftones

Halftoning is a process by which continuous-tone colors are approximated on an output device that can achieve only a limited number of discrete colors. Colors that the device cannot produce directly are simulated by using patterns of pixels in the colors available. Perhaps the most familiar example is the rendering of gray tones with black and white pixels, as in a newspaper photograph.

Some output devices can reproduce continuous-tone colors directly. Halftoning is not required for such devices; after gamma correction by the transfer functions, the color components are transmitted directly to the device. On devices that do require halftoning, it occurs after all color components have been transformed by the applicable transfer functions. The input to the halftone function consists of continuous-tone, gamma-corrected color components in the device's native color space. Its output consists of pixels in colors the device can reproduce.

The PostScript language provides a high degree of control over details of the halftoning process. For example, in color printing, independent halftone screens must be specified for each of several colorants. When rendering on low-resolution displays, fine control over halftone patterns is needed to achieve the best approximations of gray levels or colors and to minimize visual artifacts.

Note: *Remember that everything pertaining to halftones is, by definition, device-dependent. In general, when an application provides its own halftone specifications, it sacrifices portability. Associated with every device is a default halftone definition that is appropriate for most applications. Only relatively sophisticated applications need to define their own halftones to achieve special effects.*

All halftones are defined in *device space*, unaffected by the current transformation matrix. For correct results, a PostScript program that defines a new halftone must know the resolution and orientation of device space. The best choice of halftone parameters often depends on specific physical properties of the output device, such as pixel shape, overlap between pixels, and effects of electronic or mechanical noise.

7.4.1 How Halftones Are Defined

In general, halftoning methods are based on the notion of a *halftone screen*, which divides the array of device pixels into *cells* that can be modified to produce the desired halftone effects. (Halftone screens are described in Section 7.4.2) There are three ways to specify halftones in a PostScript program:

- The **setscreen** operator establishes a single halftone screen to be applied to all color components. The halftone screen can be specified in only one way: by its frequency, angle, and spot function. (These operands have the same meanings as the **Frequency**, **Angle**, and **SpotFunction** entries in a type 1 halftone dictionary, described in Section 7.4.4, "Spot Functions.") Most LanguageLevel 1 implementations support only a single halftone screen.
- The **setcolorscreen** operator establishes four separate halftone screens, one each for red, green, blue, and gray or their complements cyan, magenta, yellow, and black. An RGB device uses the first three; a monochrome device uses the gray screen only; and a CMYK device uses all four. **setcolorscreen** is supported in LanguageLevel 2 and in some LanguageLevel 1 implementations, primarily those in color printers.

- The **sethalftone** operator installs a *halftone dictionary*, which can describe any of several types of halftones. The dictionary contains the parameters of the halftoning algorithm, either for all color components together or for each component separately. It may optionally contain other rendering controls as well, such as transfer functions.

sethalftone is the most general way to specify halftones. Any halftone that can be defined in the other two ways can also be defined as a halftone dictionary. Additionally, **sethalftone** is the only way to establish a halftone for a nonprimary color component, or for any color component on a device whose native device space is a **DeviceN** space. However, halftone dictionaries are a LanguageLevel 2 feature, whereas **setscreen** (and sometimes **setcolorscreen**) is available in all LanguageLevels.

For compatibility between LanguageLevels, the **setscreen**, **setcolorscreen**, **sethalftone**, **currentscreen**, **currentcolorscreen**, and **currenthalftone** operators interact in various ways to ensure reasonable behavior when a halftone that has been defined in one way is read out in a different way. Details of these interactions are given in the descriptions of the six operators in Chapter 8. Additionally, three user parameters affect the behavior of the halftone-setting operators:

- **AccurateScreens** *(LanguageLevel 2)* enables the accurate halftoning feature for halftones defined by **setscreen** and **colorscreen**. The effect is equivalent to that of setting the **AccurateScreens** flag in a type 1 halftone dictionary (see "Type 1 Halftone Dictionaries" on page 487).
- **HalftoneMode** *(LanguageLevel 3)* allows requested halftones to be overridden by a built-in device-specific halftone (see Section C.3.4, "Halftone Screens," on page 756).
- **MaxSuperScreen** *(LanguageLevel 3)* controls the maximum size of a supercell, which can increase the number of achievable gray levels (see Section 7.4.8, "Supercells.")

7.4.2 Halftone Screens

As noted above, PostScript halftone functions are based on the use of a *halftone screen.* A screen is defined by conceptually laying a uniform rectangular grid of *halftone cells* over the device pixel array. Each pixel belongs to one cell of the grid; a single cell usually contains many pixels. The screen grid is defined entirely in device space, and is unaffected by modifications to the current transformation

matrix (CTM). This property is essential to ensure that adjacent areas colored by halftones are properly stitched together without visible "seams."

On a black-and-white device, each cell of a screen can be made to approximate a shade of gray by painting some of the cell's pixels black and some white. Numerically, the gray level produced within a cell is the ratio of the cell's pixels that are white to the total number of pixels in that cell. If a cell contains n pixels, then it can render $n+1$ different gray levels: all pixels black, 1 pixel white, 2 pixels white, … $n-1$ pixels white, all n pixels white. A desired gray value g in the range 0.0 to 1.0 is produced by making i pixels white, where $i = \text{floor}(g \times n)$.

The foregoing description also applies to color output devices whose pixels consist of primary colors that are either completely on or completely off. Most color printers, but not color displays, work this way. Halftoning is applied to each color component independently, producing shades of that color.

Color components are presented to the halftoning machinery in additive form, regardless of whether they were originally specified in additive (RGB or gray) or subtractive (CMYK or tint) form. Larger values of a color component represent lighter colors—greater intensity in an additive device such as a display, and less ink in a subtractive device such as a printer. Transfer functions produce color values in additive form; see Section 7.3, "Transfer Functions."

7.4.3 Halftone Dictionaries

A *halftone dictionary* is a dictionary object whose entries are parameters to the halftoning machinery. The graphics state includes a *current halftone dictionary*, which specifies the halftoning process to be used by the painting operators. The operator **currenthalftone** returns the current halftone dictionary; **sethalftone** establishes a different halftone dictionary as the current one. The halftone dictionary is a LanguageLevel 2 feature; in LanguageLevel 1, the **setscreen** operator controls halftoning in a more limited way.

A halftone dictionary is a self-contained description of a halftoning process. Painting operations, such as **fill**, **stroke**, and **show**, consult the current halftone dictionary when they require information about the halftoning process. The interpreter consults the halftone dictionary at unpredictable times, and can cache the results internally for later use. For these reasons, once a halftone dictionary has been passed to **sethalftone**, its contents should be considered read-only. Some of the entries in the dictionary are procedures that are called to compute

the required information. Such procedures must compute results that depend only on information in the halftone dictionary and not on outside information such as the graphics state itself, and they must have no side effects.

Note: *This restriction rules out certain techniques, such as the "pattern fill" example in the* PostScript Language Tutorial and Cookbook, *that depend on the spot function (Section 7.4.4) being executed at predictable times. Such techniques work for halftones defined by* ***setscreen****, but not for those defined by halftone dictionaries. See Section 4.9, "Patterns," for recommended ways to create device-independent patterns.*

Every halftone dictionary must have a **HalftoneType** entry whose value is an integer. This specifies the overall type of halftoning process. The remaining entries in the dictionary are interpreted according to this type. Table 7.3 lists the standard halftone types.

7.4.4 Spot Functions

A common way of defining a halftone screen is by specifying a *frequency*, *angle*, and *spot function.* The frequency is the number of halftone cells per inch; the angle indicates the orientation of the grid lines relative to the device coordinate system. As a cell's desired gray level varies from black to white, individual pixels within the cell change from black to white in a well-defined sequence: if a particular gray level includes certain white pixels, lighter grays will include the same white pixels and some additional ones as well. The order in which pixels change from black to white for increasing gray levels is determined by a PostScript procedure called a *spot function*, which specifies the order of pixel whitening in an indirect way that minimizes interactions with the screen frequency and angle.

Consider a halftone cell to have its own coordinate system: the center of the cell is the origin and the corners are at coordinates ±1.0 horizontally and vertically. Each pixel in the cell is centered at horizontal and vertical coordinates that both lie in the range −1.0 to +1.0. For each pixel, the PostScript interpreter pushes the pixel's coordinates on the operand stack and calls the spot function procedure. The procedure must return a single number in the range −1.0 to +1.0 that defines the pixel's position in the whitening order.

TABLE 7.3 Types of halftone dictionaries

TYPE	MEANING
1	Defines a single halftone screen by a *frequency, angle,* and *spot function.* (The **setscreen** operator, available in all LanguageLevels, also defines halftones this way, but expects the parameters to be given as separate operands rather than bundled into a halftone dictionary.)
2	Defines four separate halftone screens, one for each primary color component. Each screen is defined by a frequency, angle, and spot function. (The **setcolorscreen** operator, available in some LanguageLevel 1 implementations, also defines halftones this way, but expects the parameters to be given as separate operands rather than bundled into a halftone dictionary.)
3	Defines a single halftone screen by a *threshold array* containing 8-bit sample values taken from a string.
4	Defines four separate halftone screens, one for each primary color component. Each screen is defined by a threshold array containing 8-bit sample values taken from a string.
5	Defines an arbitrary number of halftone screens, one for each color component (including both primary and spot color components). The keys in this dictionary are names of color components; the values are halftone dictionaries of other types, each defining the halftone screen for a single color component.
6	*(LanguageLevel 3)* Defines a single halftone screen by a threshold array containing 8-bit sample values taken from a file.
9	*(LanguageLevel 3)* Defines a single halftone screen whose data is proprietary. The halftone dictionary may contain only the standard entries defined by Adobe.
10	*(LanguageLevel 3)* Defines a single halftone screen by a threshold array representing a halftone cell that may have a nonzero screen angle.
16	*(LanguageLevel 3)* Defines a single halftone screen by a threshold array containing 16-bit sample values taken from a file.
100	*(LanguageLevel 3)* Defines a single halftone screen whose data is proprietary. The halftone dictionary may contain optional, proprietary entries in addition to the standard ones defined by Adobe.

The specific values the spot function returns are not significant; all that matters is the *relative* values returned for different pixels. As a cell's gray level varies from black to white, the first pixel whitened is the one for which the spot function returns the lowest value, the next pixel is the one with the next higher spot function value, and so on. If two pixels have the same spot function value, their relative order is chosen arbitrarily.

Figure 7.1 shows the effects of some relatively simple spot functions that define common halftone patterns.



FIGURE 7.1 *Various halftoning effects*

A spot function whose value is inversely related to the distance from the center of the cell produces a "dot screen" in which the black pixels are clustered within a circle whose area is inversely proportional to the gray level. An example of such a spot function is

```
{   180 mul cos
    exch 180 mul cos
    add
    2 div
}
```

A spot function whose value is the distance from a line through the center of the cell produces a "line screen" in which the white pixels grow away from that line. More complex patterns are occasionally useful as well.

Type 1 Halftone Dictionaries

Table 7.4 describes the contents of a halftone dictionary of type 1, which defines a halftone screen in terms of its frequency, angle, and spot function. Like any halftone dictionary, it is selected with the **sethalftone** operator. This type of halftone can also be set with the **setscreen** operator, available in all LanguageLevels, which accepts the frequency, angle, and spot function directly as operands rather than as entries in a halftone dictionary. Both **sethalftone** and **setscreen** may make slight adjustments to the requested frequency and angle to ensure that the patterns of enclosed pixels remain constant as screen cells are replicated over the entire page.

TABLE 7.4 Entries in a type 1 halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 1 for this type of halftone.
HalftoneName	name or string	*(Optional; LanguageLevel 3)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
Frequency	number	*(Required)* The screen frequency, measured in halftone cells per inch in device space.
Angle	number	*(Required)* The screen angle, in degrees of rotation counterclockwise with respect to the device coordinate system. (Note that most output devices have left-handed device spaces; on such devices, a counterclockwise angle in device space will correspond to a clockwise angle in default user space and on the physical medium.)
SpotFunction	procedure	*(Required)* A procedure defining the order in which device pixels within a screen cell are adjusted for different gray levels.
AccurateScreens	boolean	*(Optional)* A flag specifying whether to invoke a special halftone algorithm that is extremely precise, but computationally expensive.
ActualFrequency	number	*(Optional)* A numeric value to be set, if present, to the actual frequency achieved.

ActualAngle	number	*(Optional)* A numeric value to be set, if present, to the actual angle achieved.
TransferFunction	procedure	*(Optional)* A transfer function that overrides the one specified by **settransfer** or **setcolortransfer**. Required if this dictionary is an element in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498) and represents either a nonprimary color component or a component of a **DeviceN** native color space.

A type 1 halftone dictionary's required entries **Frequency**, **Angle**, and **SpotFunction** specify the basic parameters of the halftone screen. If the optional entries **ActualFrequency** and **ActualAngle** are present, the **sethalftone** operator sets their values to the actual frequency and angle achieved. The **Frequency** and **Angle** entries remain undisturbed, reflecting the values originally requested by the program. This feature is not available for screens defined with the **setscreen** operator.

If the optional entry **AccurateScreens** is present with a boolean value of *true*, a highly precise halftoning algorithm is substituted in place of the standard one; if the **AccurateScreens** entry is *false* or is not present, ordinary halftoning is used. Accurate halftoning achieves the requested screen frequency and angle with very high accuracy, whereas ordinary halftoning adjusts them so that a single screen cell is quantized to device pixels. High accuracy is important mainly for making color separations on high-resolution devices. However, it may be computationally expensive and so is ordinarily disabled. For screens defined with the **setscreen** operator, this feature can be requested with the **AccurateScreens** user parameter.

When **AccurateScreens** is *true*, the **sethalftone** operator intentionally defers calling the spot function until the screen is needed by some operator, such as **fill**, that renders marks on the current page. **sethalftone** itself executes quickly; the potentially high cost of building the screen is not incurred until the screen is actually used. This makes it convenient to obtain **ActualFrequency** and **ActualAngle** values for various candidate screens without incurring the cost of building them.

In principle, the PostScript language permits the use of halftone screens with arbitrarily large cells—in other words, arbitrarily low frequencies. However, cells that are very large relative to the device resolution or that are oriented at unfavorable angles may exceed the capacity of available memory. If this occurs, **setscreen** or **sethalftone** executes a **limitcheck** error. The **AccurateScreens** feature often requires very large amounts of memory to achieve the highest accuracy. (See

Appendix C for information on this and other user and system parameters affecting halftone screens.)

7.4.5 Threshold Arrays

Another way to define a halftone screen is with a *threshold array* that directly controls individual device pixels in a halftone cell. This technique provides a high degree of control over halftone rendering. It also permits halftone cells to be rectangular, whereas those controlled by a spot function are always square.

A threshold array is much like a sampled image—a rectangular array of pixel values—but is defined entirely in device space. Depending on the halftone type, the threshold values occupy 8 or 16 bits each. Threshold values nominally represent gray levels in the usual way, from 0 for black up to the maximum (255 or 65,535) for white. The threshold array is replicated to tile the entire device space: each pixel in device space is mapped to a particular sample in the threshold array. On a bilevel device, where each pixel is either black or white, halftoning with a threshold array proceeds as follows:

1. For each device pixel that is to be painted with some gray level, consult the corresponding threshold value from the threshold array.
2. If the requested gray level is less than the threshold value, paint the device pixel black; otherwise, paint it white. Gray levels in the range 0.0 to 1.0 correspond to threshold values from 0 to the maximum available (255 or 65,535).

Note: *A threshold value of 0 is treated as if it were 1; therefore, a gray level of 0.0 paints all pixels black, regardless of the values in the threshold array.*

This scheme easily generalizes to monochrome devices with multiple bits per pixel. For example, if there are 2 bits per pixel, each pixel can directly represent one of four different gray levels: black, dark gray, light gray, and white, encoded as 0, 1, 2, and 3, respectively. For any device pixel that is specified with some in-between gray level, the halftoning algorithm consults the corresponding value in the threshold array to determine whether to use the next-lower or next-higher representable gray level. In this situation, the threshold values do not represent absolute gray levels, but rather gradations between any two adjacent representable gray levels.

A halftone defined in this way can also be used with color displays that have a limited number of values for each color component. The red, green, and blue components are simply treated independently as gray levels, applying the appropriate threshold array to each. (This technique also works for a screen defined as a spot function, since the PostScript interpreter uses the spot function to compute a threshold array internally.)

Type 3 Halftone Dictionaries

Table 7.5 describes the contents of a halftone dictionary of type 3, which defines a halftone screen with a threshold array. The **Width** and **Height** entries specify the dimensions of the array in device pixels. The contents of the array are given by a string object in the dictionary's **Thresholds** entry, with each byte in the string representing a single 8-bit threshold value.

TABLE 7.5 Entries in a type 3 halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 3 for this type of halftone.
HalftoneName	name or string	*(Optional; LanguageLevel 3)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
Width	integer	*(Required)* The width of threshold array, in device pixels.
Height	integer	*(Required)* The height of threshold array, in device pixels.
Thresholds	string	*(Required)* A string containing threshold values. The string must contain **Width** × **Height** bytes of threshold data. Each byte represents a single threshold value, defined in the same order as samples in a sampled image: the first value is at device coordinates (0, 0), and horizontal coordinates change faster than vertical.
TransferFunction	procedure	*(Optional)* A transfer function that overrides the one specified by **settransfer** or **setcolortransfer**. Required if this dictionary is an element in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498) and represents either a nonprimary color component or a component of a **DeviceN** native color space.

Type 6 Halftone Dictionaries

The type 6 halftone dictionary *(LanguageLevel 3)* is similar to type 3, except that the contents of the threshold array are obtained from a file instead of a string object. This allows the threshold array to exceed the implementation limit for strings (typically 65,535 bytes), although smaller threshold arrays can also be defined in this way. Table 7.6 describes the contents of this type of halftone dictionary.

TABLE 7.6 Entries in a type 6 halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 6 for this type of halftone.
HalftoneName	name or string	*(Optional)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
Width	integer	*(Required)* The width of threshold array, in device pixels.
Height	integer	*(Required)* The height of threshold array, in device pixels.
Thresholds	file	*(Required)* An input file from which at least **Width** × **Height** bytes of threshold data can be read. Each byte represents a single threshold value, defined in the same order as samples in a sampled image: the first value is at device coordinates (0, 0), and horizontal coordinates change faster than vertical.
TransferFunction	procedure	*(Optional)* A transfer function that overrides the one specified by **settransfer** or **setcolortransfer**. Required if this dictionary is an element in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498) and represents either a nonprimary color component or a component of a **DeviceN** native color space.

When presented with a type 6 halftone dictionary, the **sethalftone** operator immediately reads **Width** × **Height** bytes from the file designated by the dictionary's **Thresholds** entry and saves the contents of the resulting threshold array in internal storage. (If the file is the same one returned by the **currentfile** operator, the threshold data is read in-line from the PostScript program itself.) The file must be open for reading, but the file object's access attribute is disregarded; **sethalftone** can read from an execute-only or no-access file object. **sethalftone**

closes the file if it encounters an end-of-file; otherwise, the file is left open. A **rangecheck** error is raised if the file ends prematurely (that is, if end-of-file is encountered before the requisite number of bytes have been read).

When the current halftone is of type 6, the **currenthalftone** operator returns a halftone dictionary whose **Thresholds** entry represents the contents of the threshold array as if it were a file. That is, the **Thresholds** file object in the dictionary returned by **currenthalftone** is different from that originally given to **sethalftone**. Its access attribute and VM allocation mode are the same as those of the original file. If the access attribute permits, a PostScript program can read the contents of the threshold array from this file. The file treats the contents of the threshold array as a circular buffer that can be read repeatedly; end-of-file will never be reached. Regardless of the file object's access attribute, the dictionary returned by **currenthalftone** can be presented to **sethalftone** to reinstall the same threshold array.

Type 10 Halftone Dictionaries

Although type 3 and 6 halftone dictionaries can be used to specify a threshold array with a zero screen angle, they make no provision for other angles. The type 10 halftone dictionary *(LanguageLevel 3)* removes this restriction and allows the use of threshold arrays for halftones with nonzero screen angles as well.

Halftone cells at nonzero angles can be difficult to specify, because they may not line up well with scan lines and because it may be difficult to determine where a given sampled point goes. The type 10 halftone dictionary addresses these difficulties by dividing the halftone cell into a pair of squares that line up at zero angles with the output device's pixel grid. The squares contain the same information as the original cell, but are much easier to store and manipulate. In addition, they can be mapped easily into the internal representation used for all rendering.

Figure 7.2 shows a halftone cell with a frequency of 38.4 cells per inch and an angle of 50.2 degrees, represented graphically in device space at a resolution of 300 dots per inch. Each asterisk in the figure represents a location in device space that is mapped to a specific location in the threshold array.

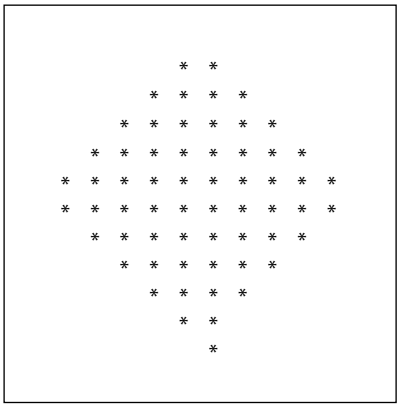


FIGURE 7.2 *Halftone cell with a nonzero angle*

Figure 7.3 shows how the halftone cell can be divided into two squares. If the squares and the original cell are tiled across device space, the area to the right of the upper square maps exactly into the empty area of the lower square, and vice versa (see Figure 7.4). The last row in the first square is immediately adjacent to the first row in the second, and starts in the same column.

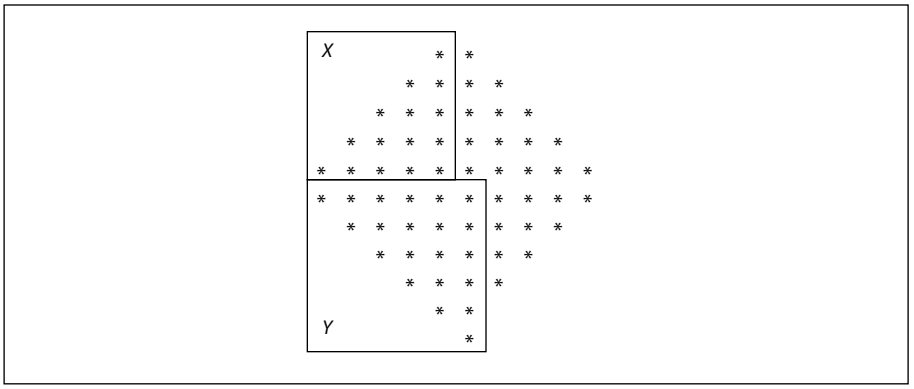


FIGURE 7.3 *Angled halftone cell divided into two squares*

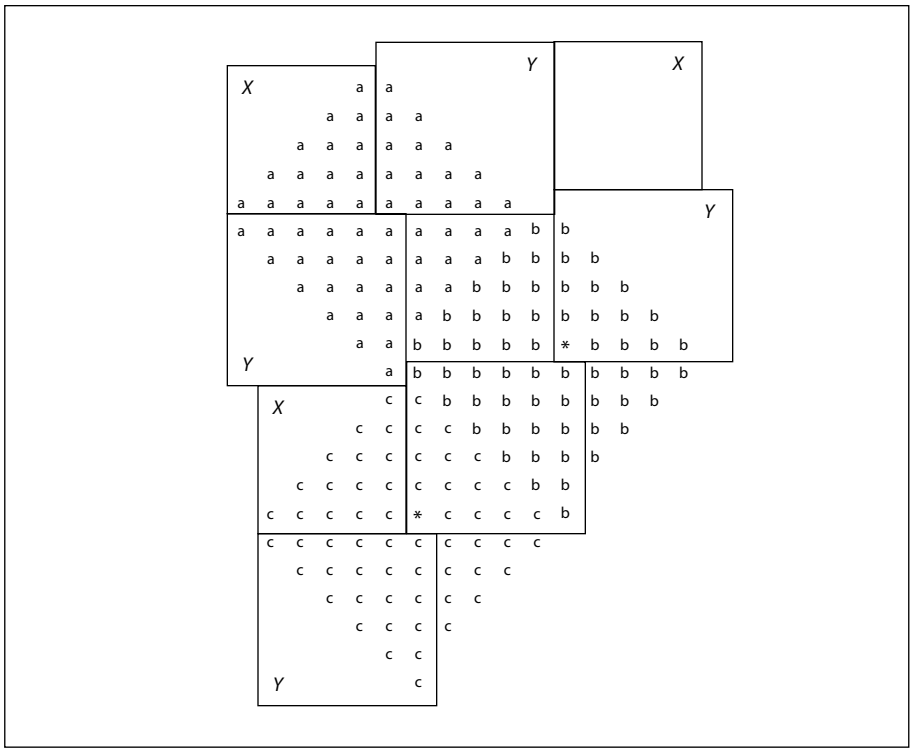


FIGURE 7.4 *Halftone cell and two squares tiled across device space*

Any halftone cell can be divided in this way. The side of the upper square (X) is equal to the horizontal displacement from a point in one halftone cell to the corresponding point in the adjacent cell, such as those marked by asterisks in Figure 7.4. The side of the lower square (Y) is the vertical displacement between the same two points. The frequency of a halftone screen constructed from squares X and Y is thus given by

$$frequency = \frac{resolution}{\sqrt{X^2 + Y^2}}$$

and the angle by

$$angle = \text{atan}\left(\frac{Y}{X}\right)$$

Table 7.7 lists the entries in a type 10 halftone dictionary.

TABLE 7.7 Entries in a type 10 halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 10 for this type of halftone.
HalftoneName	name or string	*(Optional)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
Xsquare	integer	*(Required)* The side of square *X*, in device pixels.
Ysquare	integer	*(Required)* The side of square *Y*, in device pixels.
Thresholds	string or file	*(Required)* A string or file containing threshold values, as in a type 3 or type 6 halftone, respectively. If a string, it must contain **Xsquare**^{2} + **Ysquare**^{2} bytes of threshold data; if a file, the stream must contain at least that many bytes. In either case, the contents of square *X* are specified first, followed by those of square *Y*. Threshold values within each square are defined in the same order as samples in a sampled image, with the first value at device coordinates (0, 0) and horizontal coordinates changing faster than vertical. If the value of **Thresholds** is a file, the **currenthalftone** operator will replace it with a new file object representing the threshold array, as described under "Type 6 Halftone Dictionaries" on page 491.
TransferFunction	procedure	*(Optional)* A transfer function that overrides the one specified by **settransfer** or **setcolortransfer**. Required if this dictionary is an element in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498) and represents either a nonprimary color component or a component of a **DeviceN** native color space.

Type 16 Halftone Dictionaries

Like type 10, a type 16 halftone dictionary *(LanguageLevel 3)* defines a halftone screen by specifying a threshold array whose contents are taken from a file, and it allows nonzero screen angles. In type 16, however, each element of the threshold array is 16 bits wide instead of 8. This allows the threshold array to distinguish 65,536 levels of color rather than only 256 levels. Table 7.8 describes the contents of this type of halftone dictionary.

TABLE 7.8 Entries in a type 16 halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 16 for this type of halftone.
HalftoneName	name or string	*(Optional)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
Width	integer	*(Required)* The width of the first (or only) rectangle in the threshold array, in device pixels.
Height	integer	*(Required)* The height of the first (or only) rectangle in the threshold array, in device pixels.
Width2	integer	*(Optional)* The width of the optional second rectangle in the threshold array, in device pixels. If present, then the **Height2** entry must be present as well; if absent, then the **Height2** entry must also be absent and the threshold array has only one rectangle.
Height2	integer	*(Optional)* The height of the optional second rectangle in the threshold array, in device pixels.
Thresholds	file	*(Required)* An input file from which at least 2 × **Width** × **Height** (or 2 × **Width** × **Height** + 2 × **Width2** × **Height2**) bytes of threshold data can be read. Each threshold value is 2 bytes (16 bits) wide, with the high-order byte stored first. The contents of the first rectangle are specified first, followed by those of the second rectangle. Threshold values within each rectangle are defined in the same order as samples in a sampled image, with the first value at device coordinates (0, 0) and horizontal coordinates changing faster than vertical.
TransferFunction	procedure	*(Optional)* A transfer function that overrides the one specified by **settransfer** or **setcolortransfer**. Required if this dictionary is an element in a type 5 halftone dictionary (see "Type 5 Halftone Dictionaries" on page 498) and represents either a nonprimary color component or a component of a **DeviceN** native color space.

The threshold array can consist of either one or two rectangles. The dimensions of the first (or only) rectangle are defined by the dictionary's **Width** and **Height** entries; those of the second, optional rectangle are defined by the optional entries **Width2** and **Height2**. If two rectangles are specified, they will tile the device space

as shown in Figure 7.5. The last row in the first rectangle is immediately adjacent to the first row in the second, and starts in the same column.

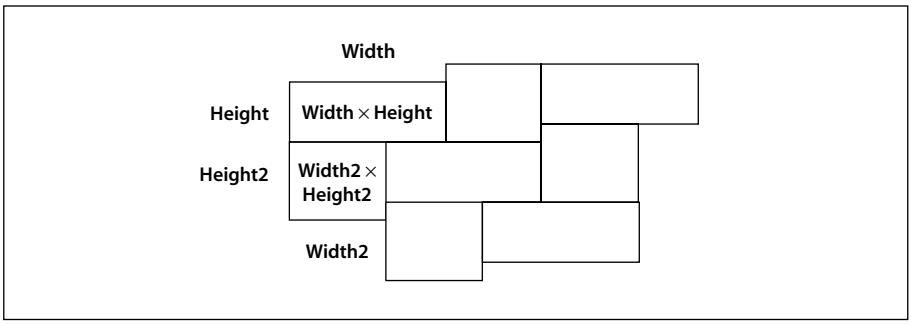


FIGURE 7.5 *Tiling of device space in a type 16 halftone dictionary*

When presented with a type 16 halftone dictionary, the **sethalftone** operator reads the contents of the first rectangle from the next $2 \times$ **Width** $\times$ **Height** bytes of the file designated by the dictionary's **Thresholds** entry. If **Width2** and **Height2** entries are present, the operator reads an additional $2 \times$ **Width2** $\times$ **Height2** bytes representing the contents of the second rectangle. In either case, the operator then saves the contents of the resulting threshold array in internal storage.

In most other respects, the **Thresholds** file and the internal threshold array are treated the same as for a type 6 halftone dictionary. However, the **currenthalftone** operator returns a halftone dictionary whose **Thresholds** file is no-access, regardless of the access attribute of the file originally presented to **sethalftone**. Consequently, a PostScript program cannot read the contents of the threshold array; however, it can present this dictionary to **sethalftone** to reinstall the same threshold array.

7.4.6 Halftone Dictionaries with Multiple Color Components

Some devices, particularly color printers, require separate halftones for each individual color component. Also, devices that can produce named separations may require individual halftones for each separation. Halftone dictionaries of types 2, 4, and 5 allow individual halftones to be specified for an arbitrary number of color components.

Type 2 and 4 Halftone Dictionaries

The type 2 halftone dictionary is similar to type 1 (see "Type 1 Halftone Dictionaries" on page 487), but defines four halftone screens—one for each primary color—instead of just one. Each of the four screens has its own frequency, angle, and spot function. In place of a single **Frequency** entry, the dictionary has separate entries named **RedFrequency**, **GreenFrequency**, **BlueFrequency**, and **GrayFrequency**. Likewise, in place of the **Angle** and **SpotFunction** entries, it has entries named **RedAngle**, **RedSpotFunction**, and so on. Of the optional entries listed in Table 7.4 on page 487 for a type 1 halftone dictionary, only **AccurateScreens** and (in LanguageLevel 3) **HalftoneName** are available in type 2.

Similarly, a type 4 halftone dictionary resembles type 3 ("Type 3 Halftone Dictionaries" on page 490), but defines four threshold arrays instead of just one. The **Width**, **Height**, and **Thresholds** entries are replicated for each color, just as in a type 2 halftone dictionary. The optional **TransferFunction** entry is not available.

Note: *The use of type 2 and 4 halftone dictionaries is not recommended, because the same effects can be obtained using the type 5 halftone dictionary (see "Type 5 Halftone Dictionaries," below). These types are supported only for compatibility with existing applications that use them.*

Type 5 Halftone Dictionaries

A type 5 halftone dictionary is a composite dictionary containing independent halftone definitions for multiple color components. Its keys are the names of individual colorants or color components; these may be either name or string objects, which are treated equivalently. The values associated with these keys are other halftone dictionaries, each defining the halftone screen and transfer function for a single colorant or color component. The component halftone dictionaries may be of any type except 2, 4, or 5.

The color components represented in a type 5 halftone dictionary fall into two categories:

- Primary color components for the standard native device color spaces, as indicated by the **ProcessColorModel** entry in the page device dictionary (**Red**,

Green, and **Blue** for **DeviceRGB**; **Cyan**, **Magenta**, **Yellow**, and **Black** for **DeviceCMYK**; **Gray** for **DeviceGray**).

- Nonstandard color components listed in the **SeparationColorNames** entry in the page device dictionary. These color components are available for use as spot colorants in **Separation** and **DeviceN** color spaces. Some of them may also be used as process colorants if the native color space is a **DeviceN** space. (Note that a type 5 halftone dictionary is the only way to specify independent halftones for colorants on a device whose native color space is a **DeviceN** space.)

The dictionary must also contain an entry whose key is **Default** and whose value is a halftone dictionary to be used for any color component that does not have an entry of its own. In LanguageLevel 3, there may optionally be an additional entry named **HalftoneName**, a name or string object specifying the name of the type 5 dictionary. If present, this value is returned by the **GetHalftoneName** procedure and used by the **findcolorrendering** operator in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).

When a halftone dictionary of some other type appears as the value of an entry in a type 5 halftone dictionary, it applies only to the single color component named by that entry's key. This is in contrast to such a dictionary being used as the main halftone dictionary (set with the **sethalftone** operator), which applies to all color components. If nonprimary color components are requested when the current halftone is defined by any means other than a type 5 halftone dictionary, the gray halftone screen and transfer function are used for all such components.

7.4.7 Proprietary Halftone Dictionaries

Some output devices support special halftone techniques in addition to the standard ones built into the PostScript language itself. Some of these techniques work well only with certain types of device technology or require special hardware to work efficiently. These device-specific halftoning techniques are not described in this book, but rather in the manufacturer's product documentation for individual devices.

Two special types of halftone dictionary, types 9 and 100 *(both LanguageLevel 3)*, support such device-dependent technologies. Both represent halftoning methods whose data is proprietary to a specific product. It is not possible for a PostScript program to gain any information about the contents or appearance of a type 9 or type 100 halftone.

A type 9 halftone dictionary contains only the generic entries **HalftoneType** and (optionally) **HalftoneName**, as shown in Table 7.9; a type 100 dictionary can also include additional entries that a particular device or halftoning technique may require. Any such optional entries are strictly private: although they are visible to PostScript programs in the dictionary returned by the **currenthalftone** operator, there is no way for a program to know what they control or what their permissible values are.

TABLE 7.9 Entries in a proprietary halftone dictionary

KEY	TYPE	VALUE
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 9 or 100 for a proprietary halftone.
HalftoneName	name or string	*(Optional)* The name of the halftone dictionary. Returned by the **GetHalftoneName** procedure and used by **findcolorrendering** in constructing the name of a color rendering dictionary (see "Customizing CRD Selection" on page 472).
other	any type	*(Optional; type 100 only)* Device-specific data.

Type 9 and 100 halftone dictionaries will be present only on those devices whose manufacturers have specifically requested this type of support. If present, they will usually be the device's default halftone dictionary; consequently, any printing that a PostScript program does will automatically take advantage of them (unless the program performs its own **sethalftone** call).

Note: *Ordinarily, a page description should not define halftone dictionaries with proprietary types; doing so makes the page description dependent on a specific output device. Indeed, any use of halftone dictionaries in a page description compromises device independence.*

7.4.8 Supercells

The use of *supercells (LanguageLevel 3)* can increase the number of achievable gray levels in most halftones by a factor of 4, up to 1016 grays, without increasing the frequency of the halftone. This can greatly reduce the incidence of rendering artifacts such as banding in shadings or contouring in images of faces or of the sky.

A supercell is an aggregate of four halftone cells that are manipulated as a single unit. A gray level that cannot be faithfully rendered with a single halftone cell can be more accurately achieved by slightly varying halftone cell values within a supercell. For example, suppose the halftone cells are 6 pixels wide by 6 high, and a gray level of 0.76 is specified. A single cell could mark 8 pixels for a gray level of 0.778, or 9 pixels for a gray level of 0.750; but a supercell could mark one of its cells with 8 pixels and three with 9, producing a gray level of 0.757.

The creation of supercells is controlled by certain user and system parameters (see Appendix C). The user parameter **MaxSuperScreen** sets an upper limit on the number of pixels in a supercell. The highest effective value is 1016 (254×4) for halftones other than type 16; higher parameter values will not add more printable gray levels. A change in the parameter value will affect subsequent executions of **sethalftone** or **setscreen**, but it will not affect halftones already in existence.

For a supercell to be created, the following conditions must hold:

- The number of pixels in the supercell must be less than or equal to **MaxSuperScreen**.
- The supercell must be within the limits of the **MaxScreenItem** user parameter and the **MaxScreenStorage** system parameter.
- The number of pixels in the original halftone cell must be less than the number of distinct colorant values that the device supports.

If any of these conditions is not met, the supercell is not created and the original halftone cell is used.

7.5 Scan Conversion Details

The final step of rendering is *scan conversion.* As discussed in Section 2.2, "Scan Conversion," the PostScript interpreter executes a scan conversion algorithm to paint graphics, text, and images in the raster memory of the output device.

The specifics of the scan conversion algorithm are not defined as part of the PostScript language. Different implementations can perform scan conversion in different ways; techniques that are appropriate for one device may be inappropriate for another. Most scan conversion details are not under program control.

Still, it is useful to have a general understanding of how scan conversion works. Developers creating applications intended to drive computer displays must pay some attention to scan conversion details. At the low resolutions typical of displays, variations of even one pixel's width can have a noticeable effect on the appearance of painted shapes.

The following sections describe the scan conversion algorithms that are typical of LanguageLevel 2 implementations from Adobe, including the basic rules and the effects of using the automatic stroke adjustment feature. Once again, these details are *not* a standard part of the PostScript language.

7.5.1 Scan Conversion Rules

The following rules determine which device pixels a painting operation will affect. All references to coordinates and pixels are in device space. A *shape* is a path to be painted with the current color or with an image. Its coordinates are mapped into device space, but not rounded to device pixel boundaries. At this level, curves have been flattened to sequences of straight lines, and all "insideness" computations have been performed.

Pixel boundaries always fall on integer coordinates in device space. A pixel is a square region identified by the location of its corner with minimum horizontal and vertical coordinates. The region is *half-open*, meaning that it includes its lower but not its upper boundaries. More precisely, for any point whose real-number coordinates are (x, y), let $i = \text{floor}(x)$ and $j = \text{floor}(y)$. The pixel that contains this point is the one identified as (i, j). The region belonging to that pixel is defined to be the set of points (x', y') such that $i \le x' < i + 1$ and $j \le y' < j + 1$. Like pixels, shapes to be painted by operators such as **fill** or **stroke** are also treated as half-open regions that include the boundaries along their "floor" sides, but not along their "ceiling" sides.

A shape is scan-converted by painting any pixel whose square region intersects the shape, no matter how small the intersection is. This ensures that no shape ever disappears as a result of unfavorable placement relative to the device pixel grid, as might happen with other possible scan conversion rules. The area covered by painted pixels is always at least as large as the area of the original shape. This rule applies both to fill operations and to strokes with nonzero width. Zero-width strokes are done in a device-dependent manner that may include fewer pixels than the rule specifies.

The region of device space to be painted by the **image** operator is determined similarly to that of a filled shape, though not identically. The interpreter transforms the image's source rectangle into device space and defines a half-open region, just as for fill operations. However, only those pixels whose *centers* lie within the region are painted. The position of the center of such a pixel—in other words, the point whose coordinate values have fractional parts of one-half—is mapped back into source space to determine how to color the pixel. There is no averaging over the pixel area; if the resolution of the source image is higher than that of device space, some source samples will not be used.

For clipping, the clipping region consists of the set of pixels that would be included by a fill operation. Subsequent painting operations affect a region that is the intersection of the set of pixels defined by the clipping region with the set of pixels for the region to be painted.

Scan conversion of character shapes is performed by a different algorithm from the one above. That font rendering algorithm uses hints in the character descriptions and techniques that are specialized to character rasterization.

7.5.2 Automatic Stroke Adjustment

When a stroke is drawn along a path, the scan conversion algorithm may produce lines of nonuniform thickness because of rasterization effects. In general, the line width and the coordinates of the endpoints, transformed into device space, are arbitrary real numbers not quantized to device pixels. A line of a given width can intersect with different numbers of device pixels, depending on where it is positioned. Figure 7.6 illustrates this effect.

For best results, it is important to compensate for the rasterization effects to produce strokes of uniform thickness. This is especially important in low-resolution display applications. To meet this need, LanguageLevel 2 provides an optional *stroke adjustment* feature. When stroke adjustment is enabled, the line width and the coordinates of a stroke are automatically adjusted as necessary to produce lines of uniform thickness. The thickness is as near as possible to the requested line width—no more than half a pixel different.

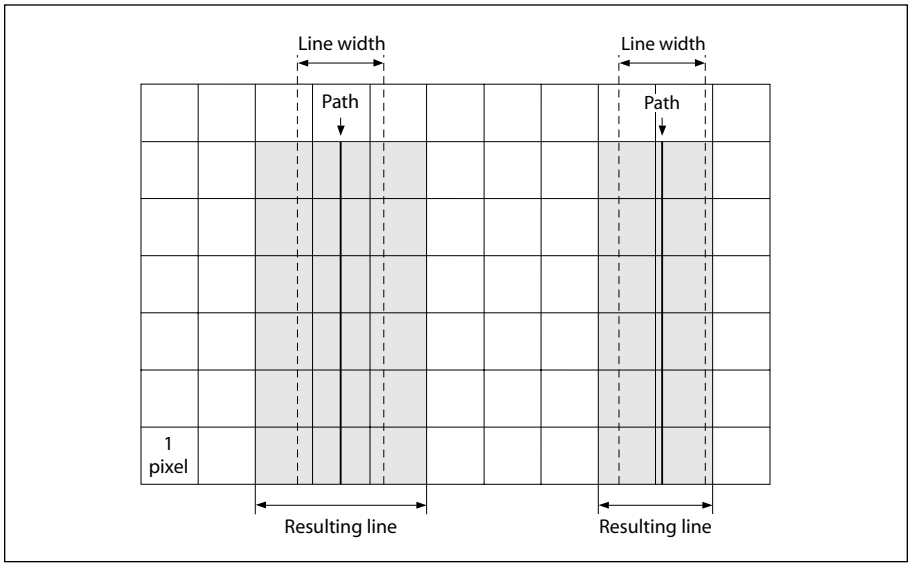


FIGURE 7.6 *Rasterization without stroke adjustment*

Note: *If stroke adjustment is enabled and the requested line width, transformed into device space, is less than half a pixel, the stroke is rendered as a single-pixel line. This is the thinnest line that can be rendered at device resolution. It is equivalent to the effect produced by setting the line width to 0 (see Section 7.5.1, "Scan Conversion Rules").*

Because automatic stroke adjustment can have a substantial effect on the appearance of lines, an application must be able to control whether the adjustment is to be performed. The **setstrokeadjust** operator alters a boolean flag in the graphics state that determines whether stroke adjustment will be performed during subsequent **stroke** and related operations.

CHAPTER 8

Operators

THIS CHAPTER CONTAINS detailed information about all the standard operators in the PostScript language. It is divided into two sections:

- Section 8.1 gives a summary of the operators, organized into groups of related functions. The summary is intended to help locate the operators needed to perform specific tasks. Less commonly used operators, such as those defined in procedure sets, are not included in this summary.
- Section 8.2 provides detailed descriptions of all operators, organized alphabetically by operator name.

Each operator description is presented in the following format:

operator *operand*$_1$ … *operand*$_n$ **operator** *result*$_1$ … *result*$_m$

A detailed explanation of the operator.

Example

An example of the use of this operator.
The symbol ⇒ designates the values left on the operand stack by the example.

Errors: **A list of errors that this operator might execute**
See Also: **A list of related operators**

At the head of an operator description, *operand*$_1$ through *operand*$_n$ are the operands that the operator requires, with *operand*$_n$ being the topmost element on the operand stack. The operator pops these objects from the operand stack and consumes them; then it executes. After executing, the operator leaves the objects *result*$_1$ through *result*$_m$ on the stack, with *result*$_m$ being the topmost element.

Normally, the operand and result names suggest either their types or their uses. Table 8.1 explains some commonly used names (other than basic type names).

TABLE 8.1 Operand and result types

NAME	DESCRIPTION
angle	Angle (number of degrees)
any	Object of any type
bool	Boolean value (*true* or *false*)
cidfont	CIDFont dictionary
dict	Dictionary object
filename	File name string
font	Font dictionary
form	Form dictionary
halftone	Halftone dictionary
int	Integer number
key	Object of any type except null, used as a key in a dictionary or a resource
matrix	Array of six numbers describing a transformation matrix
num	Number (integer or real)
numarray	Array of numbers
numstring	Encoded number string
pattern	Pattern dictionary
proc	Procedure (executable array or executable packed array)
real	Real (floating-point) number
userpath	Array of path construction operators and their operands

Some operators are polymorphic: their operands may be any of several types. For example, the notation *file* | *proc* | *string* indicates an operand that may be a file, a procedure, or a string.

The notation "⊢ " indicates the bottom of the stack. The notation "–" in the operand position indicates that the operator expects no operands, and a "–" in the result position indicates that the operator returns no results.

The documented effects on the operand stack and the possible errors are those produced directly by the operator itself. Many operators invoke arbitrary PostScript procedures. Such procedures can have arbitrary effects that are not mentioned in the operator descriptions.

In several descriptions of operators, the operator is described as being "equivalent to" a PostScript program using lower-LanguageLevel operators. Unless explicitly documented to the contrary, operator definitions are independent; redefining an operator name does not change the behavior of any other operator.

The PostScript language consists of three distinct groups of operators: LanguageLevel 1, LanguageLevel 2, and LanguageLevel 3 operators. This chapter clearly identifies LanguageLevel 2 and LanguageLevel 3 operators with the following icons:



LanguageLevel 2 operator



LanguageLevel 3 operator

LanguageLevel 1 operators are not identified with a specific icon. Some operators are available in LanguageLevel 1 but have alternate forms that are supported only in LanguageLevel 2 or higher; such forms are identified in the operator descriptions with the notation "*(LanguageLevel 2)*." Note that some LanguageLevel 2 operators are present in LanguageLevel 1 implementations that contain various language extensions; see Appendix A for details.

All operators are defined in **systemdict** except as noted otherwise in the operator description—with, for example, "*(**CIDInit** procedure set)*."

8.1 Operator Summary

Operand Stack Manipulation Operators

any	**pop**	–	Discard top element
any_1 any_2	**exch**	any_2 any_1	Exchange top two elements
any	**dup**	*any any*	Duplicate top element
any_1 … any_n *n*	**copy**	any_1 … any_n any_1 … any_n	Duplicate top *n* elements
any_n … any_0 *n*	**index**	any_n … any_0 any_n	Duplicate arbitrary element
any_{n-1} … any_0 *n j*	**roll**	$any_{(j-1) \bmod n}$ … any_0 any_{n-1} … $any_{j \bmod n}$	Roll *n* elements up *j* times
⊢ any_1 … any_n	**clear**	⊢	Discard all elements
⊢ any_1 … any_n	**count**	⊢ any_1 … any_n *n*	Count elements on stack
–	**mark**	*mark*	Push mark on stack
mark obj_1 … obj_n	**cleartomark**	–	Discard elements down through *mark*
mark obj_1 … obj_n	**counttomark**	*mark* obj_1 … obj_n *n*	Count elements down to *mark*

Arithmetic and Math Operators

num_1 num_2	**add**	*sum*	Return num_1 plus num_2
num_1 num_2	**div**	*quotient*	Return num_1 divided by num_2
int_1 int_2	**idiv**	*quotient*	Return int_1 divided by int_2
int_1 int_2	**mod**	*remainder*	Return remainder after dividing int_1 by int_2
num_1 num_2	**mul**	*product*	Return num_1 times num_2
num_1 num_2	**sub**	*difference*	Return num_1 minus num_2
num_1	**abs**	num_2	Return absolute value of num_1
num_1	**neg**	num_2	Return negative of num_1
num_1	**ceiling**	num_2	Return ceiling of num_1
num_1	**floor**	num_2	Return floor of num_1
num_1	**round**	num_2	Round num_1 to nearest integer
num_1	**truncate**	num_2	Remove fractional part of num_1
num	**sqrt**	*real*	Return square root of *num*
num den	**atan**	*angle*	Return arctangent of *num*/*den* in degrees
angle	**cos**	*real*	Return cosine of *angle* degrees
angle	**sin**	*real*	Return sine of *angle* degrees
base exponent	**exp**	*real*	Raise *base* to *exponent* power
num	**ln**	*real*	Return natural logarithm (base *e*)
num	**log**	*real*	Return common logarithm (base 10)

–	**rand** *int*	Generate pseudo-random integer
int	**srand** –	Set random number seed
–	**rrand** *int*	Return random number seed

Array Operators

int	**array** *array*	Create array of length *int*
–	**[** *mark*	Start array construction
mark obj_0 … obj_{n-1}	**]** *array*	End array construction
array	**length** *int*	Return number of elements in *array*
array index	**get** *any*	Return *array* element indexed by *index*
array index any	**put** –	Put *any* into *array* at *index*
array index count	**getinterval** *subarray*	Return subarray of *array* starting at *index* for *count* elements
$array_1$ *index* $array_2$|*packedarray*$_2$	**putinterval** –	Replace subarray of $array_1$ starting at *index* by $array_2$|*packedarray*$_2$
any_0 … any_{n-1} *array*	**astore** *array*	Pop elements from stack into *array*
array	**aload** any_0 … any_{n-1} *array*	Push all elements of *array* on stack
$array_1$ $array_2$	**copy** $subarray_2$	Copy elements of $array_1$ to initial subarray of $array_2$
array proc	**forall** –	Execute *proc* for each element of *array*

Packed Array Operators

any_0 … any_{n-1} *n*	**packedarray** *packedarray*	Create packed array consisting of *n* elements from stack
bool	**setpacking** –	Set array packing mode for { … } syntax (*true* = packed array)
–	**currentpacking** *bool*	Return array packing mode
packedarray	**length** *int*	Return number of elements in *packedarray*
packedarray index	**get** *any*	Return *packedarray* element indexed by *index*
packedarray index count	**getinterval** *subarray*	Return subarray of *packedarray* starting at *index* for *count* elements
packedarray	**aload** any_0 … any_{n-1} *packedarray*	Push all elements of *packedarray* on stack
$packedarray_1$ $array_2$	**copy** $subarray_2$	Copy elements of $packedarray_1$ to initial subarray of $array_2$
packedarray proc	**forall** –	Execute *proc* for each element of *packedarray*

Dictionary Operators

int	**dict** *dict*	Create dictionary with capacity for *int* elements
–	**<<** *mark*	Start dictionary construction
mark key_1 $value_1$ … key_n $value_n$	**>>** *dict*	End dictionary construction
dict	**length** *int*	Return number of entries in *dict*
dict	**maxlength** *int*	Return current capacity of *dict*
dict	**begin** –	Push *dict* on dictionary stack
–	**end** –	Pop current dictionary off dictionary stack
key value	**def** –	Associate *key* and *value* in current dictionary
key	**load** *value*	Search dictionary stack for *key* and return associated *value*
key value	**store** –	Replace topmost definition of *key*
dict key	**get** *any*	Return value associated with *key* in *dict*
dict key value	**put** –	Associate *key* with *value* in *dict*
dict key	**undef** –	Remove *key* and its value from *dict*
dict key	**known** *bool*	Test whether *key* is in *dict*
key	**where** *dict true* or *false*	Find dictionary in which *key* is defined
$dict_1$ $dict_2$	**copy** $dict_2$	Copy contents of $dict_1$ to $dict_2$
dict proc	**forall** –	Execute *proc* for each entry in *dict*
–	**currentdict** *dict*	Return current dictionary
–	**errordict** *dict*	Return error handler dictionary
–	**$error** *dict*	Return error control and status dictionary
–	**systemdict** *dict*	Return system dictionary
–	**userdict** *dict*	Return writeable dictionary in local VM
–	**globaldict** *dict*	Return writeable dictionary in global VM
–	**statusdict** *dict*	Return product-dependent dictionary
–	**countdictstack** *int*	Count elements on dictionary stack
array	**dictstack** *subarray*	Copy dictionary stack into *array*
–	**cleardictstack** –	Pop all nonpermanent dictionaries off dictionary stack

String Operators

int	**string** *string*	Create string of length *int*
string	**length** *int*	Return number of elements in *string*
string index	**get** *int*	Return string element indexed by *index*
string index int	**put** –	Put *int* into *string* at *index*
string index count	**getinterval** *substring*	Return substring of *string* starting at *index* for *count* elements
$string_1$ *index* $string_2$	**putinterval** –	Replace substring of $string_1$ starting at *index* by $string_2$
$string_1$ $string_2$	**copy** $substring_2$	Copy elements of $string_1$ to initial substring of $string_2$
string proc	**forall** –	Execute *proc* for each element of *string*
string seek	**anchorsearch** *post match true* or *string false*	Search for *seek* at start of *string*
string seek	**search** *post match pre true* or *string false*	Search for *seek* in *string*
string	**token** *post any true* or *false*	Read token from start of *string*

Relational, Boolean, and Bitwise Operators

any_1 any_2	**eq** *bool*	Test equal
any_1 any_2	**ne** *bool*	Test not equal
$num_1\|str_1$ $num_2\|str_2$	**ge** *bool*	Test greater than or equal
$num_1\|str_1$ $num_2\|str_2$	**gt** *bool*	Test greater than
$num_1\|str_1$ $num_2\|str_2$	**le** *bool*	Test less than or equal
$num_1\|str_1$ $num_2\|str_2$	**lt** *bool*	Test less than
$bool_1\|int_1$ $bool_2\|int_2$	**and** $bool_3\|int_3$	Perform logical|bitwise and
$bool_1\|int_1$	**not** $bool_2\|int_2$	Perform logical|bitwise not
$bool_1\|int_1$ $bool_2\|int_2$	**or** $bool_3\|int_3$	Perform logical|bitwise inclusive or
$bool_1\|int_1$ $bool_2\|int_2$	**xor** $bool_3\|int_3$	Perform logical|bitwise exclusive or
–	**true** *true*	Return boolean value *true*
–	**false** *false*	Return boolean value *false*
int_1 *shift*	**bitshift** int_2	Perform bitwise shift of int_1 (positive is left)

Control Operators

any	**exec** –	Execute arbitrary object
bool proc	**if** –	Execute *proc* if *bool* is *true*
bool $proc_1$ $proc_2$	**ifelse** –	Execute $proc_1$ if *bool* is *true*, $proc_2$ if *false*
initial increment limit proc	**for** –	Execute *proc* with values from *initial* by steps of *increment* to *limit*
int proc	**repeat** –	Execute *proc int* times
proc	**loop** –	Execute *proc* an indefinite number of times
–	**exit** –	Exit innermost active loop
–	**stop** –	Terminate **stopped** context
any	**stopped** *bool*	Establish context for catching **stop**
–	**countexecstack** *int*	Count elements on execution stack
array	**execstack** *subarray*	Copy execution stack into *array*
–	**quit** –	Terminate interpreter
–	**start** –	Executed at interpreter startup

Type, Attribute, and Conversion Operators

any	**type** *name*	Return type of *any*
any	**cvlit** *any*	Make object literal
any	**cvx** *any*	Make object executable
any	**xcheck** *bool*	Test executable attribute
array|*packedarray*|*file*|*string*	**executeonly** *array*|*packedarray*|*file*|*string*	Reduce access to execute-only
array|*packedarray*|*dict*|*file*|*string*	**noaccess** *array*|*packedarray*|*dict*|*file*|*string*	Disallow any access
array|*packedarray*|*dict*|*file*|*string*	**readonly** *array*|*packedarray*|*dict*|*file*|*string*	Reduce access to read-only
array|*packedarray*|*dict*|*file*|*string*	**rcheck** *bool*	Test read access
array|*packedarray*|*dict*|*file*|*string*	**wcheck** *bool*	Test write access
num|*string*	**cvi** *int*	Convert to integer
string	**cvn** *name*	Convert to name
num|*string*	**cvr** *real*	Convert to real
num radix string	**cvrs** *substring*	Convert with radix to string
any string	**cvs** *substring*	Convert to string

File Operators

filename access	**file** *file*	Open named file with specified access
datasrc|datatgt dict		
param$_1$ … *param*$_n$ *filtername*	**filter** *file*	Establish filtered file
file	**closefile** –	Close *file*
file	**read** *int true* or *false*	Read one character from *file*
file int	**write** –	Write one character to *file*
file string	**readhexstring** *substring bool*	Read hexadecimal numbers from *file* into *string*
file string	**writehexstring** –	Write *string* to *file* as hexadecimal
file string	**readstring** *substring bool*	Read string from *file*
file string	**writestring** –	Write *string* to *file*
file string	**readline** *substring bool*	Read line from *file* into *string*
file	**token** *any true* or *false*	Read token from *file*
file	**bytesavailable** *int*	Return number of bytes available to read
–	**flush** –	Send buffered data to standard output file
file	**flushfile** –	Send buffered data or read to EOF
file	**resetfile** –	Discard buffered characters
file	**status** *bool*	Return status of *file* (*true* = valid)
filename	**status** *pages bytes referenced created true* or *false*	Return information about named file
filename	**run** –	Execute contents of named file
–	**currentfile** *file*	Return file currently being executed
filename	**deletefile** –	Delete named file
filename$_1$ *filename*$_2$	**renamefile** –	Rename file *filename*$_1$ to *filename*$_2$
template proc scratch	**filenameforall** –	Execute *proc* for each file name matching *template*
file position	**setfileposition** –	Set *file* to specified position
file	**fileposition** *position*	Return current position in *file*
string	**print** –	Write *string* to standard output file
any	**=** –	Write text representation of *any* to standard output file
any	**==** –	Write syntactic representation of *any* to standard output file
⊢ *any*$_1$ … *any*$_n$	**stack** ⊢ *any*$_1$ … *any*$_n$	Print stack nondestructively using =
⊢ *any*$_1$ … *any*$_n$	**pstack** ⊢ *any*$_1$ … *any*$_n$	Print stack nondestructively using ==

obj tag	**printobject** –	Write binary object to standard output file, using *tag*
file obj tag	**writeobject** –	Write binary object to *file*, using *tag*
int	**setobjectformat** –	Set binary object format (0 = disable, 1 = IEEE high, 2 = IEEE low, 3 = native high, 4 = native low)
–	**currentobjectformat** *int*	Return binary object format

Resource Operators

key instance category	**defineresource** *instance*	Register named resource *instance* in *category*
key category	**undefineresource** –	Remove resource registration
key category	**findresource** *instance*	Return resource *instance* identified by *key* in *category*
renderingintent	**findcolorrendering** *name bool*	Select CIE-based color rendering dictionary by rendering intent
key category	**resourcestatus** *status size true* or *false*	Return status of resource instance
template proc scratch category	**resourceforall** –	Enumerate resource instances in *category*

Virtual Memory Operators

–	**save** *save*	Create VM snapshot
save	**restore** –	Restore VM snapshot
bool	**setglobal** –	Set VM allocation mode (*false* = local, *true* = global)
–	**currentglobal** *bool*	Return current VM allocation mode
any	**gcheck** *bool*	Return *true* if *any* is simple or in global VM, *false* if in local VM
$bool_1$ *password*	**startjob** $bool_2$	Start new job that will alter initial VM if $bool_1$ is *true*
index any	**defineuserobject** –	Define user object associated with *index*
index	**execuserobject** –	Execute user object associated with *index*
index	**undefineuserobject** –	Remove user object associated with *index*
–	**UserObjects** *array*	Return current **UserObjects** array defined in **userdict**

Miscellaneous Operators

proc **bind** *proc*		Replace operator names in *proc* with operators; perform idiom recognition
– **null** *null*		Push *null* on stack
– **version** *string*		Return interpreter version
– **realtime** *int*		Return real time in milliseconds
– **usertime** *int*		Return execution time in milliseconds
– **languagelevel** *int*		Return LanguageLevel
– **product** *string*		Return product name
– **revision** *int*		Return product revision level
– **serialnumber** *int*		Return machine serial number
– **executive** –		Invoke interactive executive
bool **echo** –		Turn echoing on or off
– **prompt** –		Executed when ready for interactive input

Graphics State Operators (Device-Independent)

– **gsave** –		Push graphics state
– **grestore** –		Pop graphics state
– **clipsave** –		Push clipping path
– **cliprestore** –		Pop clipping path
– **grestoreall** –		Pop to bottommost graphics state
– **initgraphics** –		Reset graphics state parameters
– **gstate** *gstate*		Create graphics state object
gstate **setgstate** –		Set graphics state from *gstate*
gstate **currentgstate** *gstate*		Copy current graphics state into *gstate*
num **setlinewidth** –		Set line width
– **currentlinewidth** *num*		Return current line width
int **setlinecap** –		Set shape of line ends for stroke (0 = butt, 1 = round, 2 = square)
– **currentlinecap** *int*		Return current line cap
int **setlinejoin** –		Set shape of corners for stroke (0 = miter, 1 = round, 2 = bevel)
– **currentlinejoin** *int*		Return current line join
num **setmiterlimit** –		Set miter length limit
– **currentmiterlimit** *num*		Return current miter limit
bool **setstrokeadjust** –		Set stroke adjustment (*false* = disable, *true* = enable)

– **currentstrokeadjust** *bool*	Return current stroke adjustment
array offset **setdash** –	Set dash pattern for stroking
– **currentdash** *array offset*	Return current dash pattern
array|*name* **setcolorspace** –	Set color space
– **currentcolorspace** *array*	Return current color space
$comp_1$ … $comp_n$ **setcolor** –	Set color components
pattern **setcolor** –	Set colored tiling pattern as current color
$comp_1$ … $comp_n$ *pattern* **setcolor** –	Set uncolored tiling pattern as current color
– **currentcolor** $comp_1$ … $comp_n$	Return current color components
num **setgray** –	Set color space to **DeviceGray** and color to specified gray value (0 = black, 1 = white)
– **currentgray** *num*	Return current color as gray value
hue saturation brightness **sethsbcolor** –	Set color space to **DeviceRGB** and color to specified hue, saturation, brightness
– **currenthsbcolor** *hue saturation brightness*	Return current color as hue, saturation, brightness
red green blue **setrgbcolor** –	Set color space to **DeviceRGB** and color to specified red, green, blue
– **currentrgbcolor** *red green blue*	Return current color as red, green, blue
cyan magenta yellow black **setcmykcolor** –	Set color space to **DeviceCMYK** and color to specified cyan, magenta, yellow, black
– **currentcmykcolor** *cyan magenta yellow black*	Return current color as cyan, magenta, yellow, black

Graphics State Operators (Device-Dependent)

halftone **sethalftone** –	Set halftone dictionary
– **currenthalftone** *halftone*	Return current halftone dictionary
frequency angle proc **setscreen** –	Set gray halftone screen by frequency, angle, and spot function
frequency angle halftone **setscreen** –	Set gray halftone screen from halftone dictionary
– **currentscreen** *frequency angle proc*|*halftone*	Return current gray halftone screen
redfreq redang redproc|*redhalftone* *greenfreq greenang greenproc*|*greenhalftone* *bluefreq blueang blueproc*|*bluehalftone* *grayfreq grayang grayproc*|*grayhalftone* **setcolorscreen** –	Set all four halftone screens

–	**currentcolorscreen**	*redfreq redang redproc|redhalftone greenfreq greenang greenproc|greenhalftone bluefreq blueang blueproc|bluehalftone grayfreq grayang grayproc|grayhalftone* Return all four halftone screens
proc	**settransfer** –	Set gray transfer function
–	**currenttransfer** *proc*	Return current gray transfer function
redproc greenproc blueproc grayproc	**setcolortransfer** –	Set all four transfer functions
–	**currentcolortransfer** *redproc greenproc blueproc grayproc*	Return current transfer functions
proc	**setblackgeneration** –	Set black-generation function
–	**currentblackgeneration** *proc*	Return current black-generation function
proc	**setundercolorremoval** –	Set undercolor-removal function
–	**currentundercolorremoval** *proc*	Return current undercolor-removal function
dict	**setcolorrendering** –	Set CIE-based color rendering dictionary
–	**currentcolorrendering** *dict*	Return current CIE-based color rendering dictionary
num	**setflat** –	Set flatness tolerance
–	**currentflat** *num*	Return current flatness
bool	**setoverprint** –	Set overprint parameter
–	**currentoverprint** *bool*	Return current overprint parameter
num	**setsmoothness** –	Set smoothness parameter
–	**currentsmoothness** *num*	Return current smoothness parameter

Coordinate System and Matrix Operators

–	**matrix** *matrix*	Create identity matrix
–	**initmatrix** –	Set CTM to device default
matrix	**identmatrix** *matrix*	Fill *matrix* with identity transform
matrix	**defaultmatrix** *matrix*	Fill *matrix* with device default matrix
matrix	**currentmatrix** *matrix*	Fill *matrix* with CTM
matrix	**setmatrix** –	Replace CTM by *matrix*
t_x t_y	**translate** –	Translate user space by (t_x, t_y)
t_x t_y *matrix*	**translate** *matrix*	Define translation by (t_x, t_y)

$s_x\ s_y$	**scale** –	Scale user space by s_x and s_y
$s_x\ s_y\ matrix$	**scale** *matrix*	Define scaling by s_x and s_y
angle	**rotate** –	Rotate user space by *angle* degrees
angle matrix	**rotate** *matrix*	Define rotation by *angle* degrees
matrix	**concat** –	Replace CTM by *matrix* × CTM
$matrix_1\ matrix_2\ matrix_3$	**concatmatrix** $matrix_3$	Fill $matrix_3$ with $matrix_1 \times matrix_2$
x y	**transform** *x′ y′*	Transform (*x*, *y*) by CTM
x y matrix	**transform** *x′ y′*	Transform (*x*, *y*) by *matrix*
dx dy	**dtransform** *dx′ dy′*	Transform distance (*dx*, *dy*) by CTM
dx dy matrix	**dtransform** *dx′ dy′*	Transform distance (*dx*, *dy*) by *matrix*
x′ y′	**itransform** *x y*	Perform inverse transform of (*x′*, *y′*) by CTM
x′ y′ matrix	**itransform** *x y*	Perform inverse transform of (*x′*, *y′*) by *matrix*
dx′ dy′	**idtransform** *dx dy*	Perform inverse transform of distance (*dx′*, *dy′*) by CTM
dx′ dy′ matrix	**idtransform** *dx dy*	Perform inverse transform of distance (*dx′*, *dy′*) by *matrix*
$matrix_1\ matrix_2$	**invertmatrix** $matrix_2$	Fill $matrix_2$ with inverse of $matrix_1$

Path Construction Operators

–	**newpath** –	Initialize current path to be empty
–	**currentpoint** *x y*	Return current point coordinates
x y	**moveto** –	Set current point to (*x*, *y*)
dx dy	**rmoveto** –	Perform relative **moveto**
x y	**lineto** –	Append straight line to (*x*, *y*)
dx dy	**rlineto** –	Perform relative **lineto**
$x\ y\ r\ angle_1\ angle_2$	**arc** –	Append counterclockwise arc
$x\ y\ r\ angle_1\ angle_2$	**arcn** –	Append clockwise arc
$x_1\ y_1\ x_2\ y_2\ r$	**arct** –	Append tangent arc
$x_1\ y_1\ x_2\ y_2\ r$	**arcto** $xt_1\ yt_1\ xt_2\ yt_2$	Append tangent arc
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	**curveto** –	Append Bézier cubic section
$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$	**rcurveto** –	Perform relative **curveto**
–	**closepath** –	Connect subpath back to its starting point
–	**flattenpath** –	Convert curves to sequences of straight lines
–	**reversepath** –	Reverse direction of current path
–	**strokepath** –	Compute outline of stroked path

userpath	**ustrokepath** –	Compute outline of stroked *userpath*
userpath matrix	**ustrokepath** –	Compute outline of stroked *userpath*
string bool	**charpath** –	Append glyph outline to current path
userpath	**uappend** –	Interpret *userpath* and append to current path
–	**clippath** –	Set current path to clipping path
$ll_x\ ll_y\ ur_x\ ur_y$	**setbbox** –	Set bounding box for current path
–	**pathbbox** $ll_x\ ll_y\ ur_x\ ur_y$	Return bounding box of current path
move line curve close	**pathforall** –	Enumerate current path
bool	**upath** *userpath*	Create *userpath* for current path; include **ucache** if *bool* is *true*
–	**initclip** –	Set clipping path to device default
–	**clip** –	Clip using nonzero winding number rule
–	**eoclip** –	Clip using even-odd rule
x y width height	**rectclip** –	Clip with rectangular path
numarray|numstring	**rectclip** –	Clip with rectangular paths
–	**ucache** –	Declare that user path is to be cached

Painting Operators

–	**erasepage** –	Paint current page white
–	**stroke** –	Draw line along current path
–	**fill** –	Fill current path with current color
–	**eofill** –	Fill using even-odd rule
x y width height	**rectstroke** –	Define rectangular path and stroke
x y width height matrix	**rectstroke** –	Define rectangular path, concatenate *matrix*, and stroke
numarray|numstring	**rectstroke** –	Define rectangular paths and stroke
numarray|numstring matrix	**rectstroke** –	Define rectangular paths, concatenate *matrix*, and stroke
x y width height	**rectfill** –	Fill rectangular path
numarray|numstring	**rectfill** –	Fill rectangular paths
userpath	**ustroke** –	Interpret and stroke *userpath*
userpath matrix	**ustroke** –	Interpret *userpath*, concatenate *matrix*, and stroke
userpath	**ufill** –	Interpret and fill *userpath*
userpath	**ueofill** –	Fill *userpath* using even-odd rule
dict	**shfill** –	Fill area defined by shading pattern

dict	**image**	–	Paint any sampled image
width height bits/sample matrix datasrc	**image**	–	Paint monochrome sampled image
width height bits/comp matrix $datasrc_0$ … $datasrc_{ncomp-1}$ *multi ncomp*	**colorimage**	–	Paint color sampled image
dict	**imagemask**	–	Paint current color through mask
width height polarity matrix datasrc	**imagemask**	–	Paint current color through mask

Insideness-Testing Operators

x y	**infill**	*bool*	Test whether (*x*, *y*) would be painted by **fill**
userpath	**infill**	*bool*	Test whether pixels in *userpath* would be painted by **fill**
x y	**ineofill**	*bool*	Test whether (*x*, *y*) would be painted by **eofill**
userpath	**ineofill**	*bool*	Test whether pixels in *userpath* would be painted by **eofill**
x y userpath	**inufill**	*bool*	Test whether (*x*, *y*) would be painted by **ufill** of *userpath*
$userpath_1$ $userpath_2$	**inufill**	*bool*	Test whether pixels in $userpath_1$ would be painted by **ufill** of $userpath_2$
x y userpath	**inueofill**	*bool*	Test whether (*x*, *y*) would be painted by **ueofill** of *userpath*
$userpath_1$ $userpath_2$	**inueofill**	*bool*	Test whether pixels in $userpath_1$ would be painted by **ueofill** of $userpath_2$
x y	**instroke**	*bool*	Test whether (*x*, *y*) would be painted by **stroke**
x y userpath	**inustroke**	*bool*	Test whether (*x*, *y*) would be painted by **ustroke** of *userpath*
x y userpath matrix	**inustroke**	*bool*	Test whether (*x*, *y*) would be painted by **ustroke** of *userpath*
$userpath_1$ $userpath_2$	**inustroke**	*bool*	Test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$
$userpath_1$ $userpath_2$ *matrix*	**inustroke**	*bool*	Test whether pixels in $userpath_1$ would be painted by **ustroke** of $userpath_2$

Form and Pattern Operators

pattern matrix	**makepattern**	*pattern′*	Create pattern instance from prototype
pattern	**setpattern**	–	Install *pattern* as current color
$comp_1$ … $comp_n$ *pattern*	**setpattern**	–	Install *pattern* as current color
form	**execform**	–	Paint *form*

Device Setup and Output Operators

– **showpage** –		Transmit and reset current page
– **copypage** –		Transmit current page
dict **setpagedevice** –		Install page-oriented output device
– **currentpagedevice** *dict*		Return current page device parameters
– **nulldevice** –		Install no-output device

Glyph and Font Operators

key font|cidfont **definefont** *font|cidfont*		Register *font|cidfont* in **Font** resource category
key name|string|dict array **composefont** *font*		Register composite font dictionary created from CMap and array of CIDFonts or fonts
key **undefinefont** –		Remove **Font** resource registration
key **findfont** *font|cidfont*		Return **Font** resource instance identified by *key*
font|cidfont scale **scalefont** *font′|cidfont′*		Scale *font|cidfont* by *scale* to produce *font′|cidfont′*
font|cidfont matrix **makefont** *font′|cidfont′*		Transform *font|cidfont* by *matrix* to produce *font′|cidfont′*
font|cidfont **setfont** –		Set font or CIDFont in graphics state
– **rootfont** *font|cidfont*		Return last set font or CIDFont
– **currentfont** *font|cidfont*		Return current font or CIDFont, possibly a descendant of **rootfont**
key scale|matrix **selectfont** –		Set font or CIDFont given name and transform
string **show** –		Paint glyphs for *string* in current font
a_x a_y *string* **ashow** –		Add (a_x, a_y) to width of each glyph while showing *string*
c_x c_y *char string* **widthshow** –		Add (c_x, c_y) to width of glyph for *char* while showing *string*
c_x c_y *char* a_x a_y *string* **awidthshow** –		Combine effects of **ashow** and **widthshow**
string numarray|numstring **xshow** –		Paint glyphs for *string* using *x* widths in *numarray|numstring*
string numarray|numstring **xyshow** –		Paint glyphs for *string* using *x* and *y* widths in *numarray|numstring*
string numarray|numstring **yshow** –		Paint glyphs for *string* using *y* widths in *numarray|numstring*

name|cid	**glyphshow** –	Paint glyph for character identified by *name|cid*
string	**stringwidth** w_x w_y	Return width of glyphs for *string* in current font
proc string	**cshow** –	Invoke character mapping algorithm and call *proc*
proc string	**kshow** –	Execute *proc* between characters shown from *string*
–	**FontDirectory** *dict*	Return dictionary of **Font** resource instances
–	**GlobalFontDirectory** *dict*	Return dictionary of **Font** resource instances in global VM
–	**StandardEncoding** *array*	Return Adobe standard font encoding vector
–	**ISOLatin1Encoding** *array*	Return ISO Latin-1 font encoding vector
key	**findencoding** *array*	Find encoding vector
w_x w_y ll_x ll_y ur_x ur_y	**setcachedevice** –	Declare cached glyph metrics
$w0_x$ $w0_y$ ll_x ll_y ur_x ur_y $w1_x$ $w1_y$ v_x v_y	**setcachedevice2** –	Declare cached glyph metrics
w_x w_y	**setcharwidth** –	Declare uncached glyph metrics

Interpreter Parameter Operators

dict	**setsystemparams** –	Set systemwide interpreter parameters
–	**currentsystemparams** *dict*	Return systemwide interpreter parameters
dict	**setuserparams** –	Set per-context interpreter parameters
–	**currentuserparams** *dict*	Return per-context interpreter parameters
string dict	**setdevparams** –	Set parameters for input/output device
string	**currentdevparams** *dict*	Return device parameters
int	**vmreclaim** –	Control garbage collector
int	**setvmthreshold** –	Control garbage collector
–	**vmstatus** *level used maximum*	Report VM status
–	**cachestatus** *bsize bmax msize mmax csize cmax blimit*	Return font cache status and parameters
int	**setcachelimit** –	Set maximum bytes in cached glyph
mark size lower upper	**setcacheparams** –	Set font cache parameters
–	**currentcacheparams** *mark size lower upper*	Return current font cache parameters

mark blimit	**setucacheparams**	–	Set user path cache parameters
–	**ucachestatus**	*mark bsize bmax rsize rmax blimit*	Return user path cache status and parameters

Errors

configurationerror	**setpagedevice** or **setdevparams** request cannot be satisfied
dictfull	No more room in dictionary
dictstackoverflow	Too many **begin** operators
dictstackunderflow	Too many **end** operators
execstackoverflow	Executive stack nesting too deep
handleerror	Called to report error information
interrupt	External interrupt request (for example, Control-C)
invalidaccess	Attempt to violate access attribute
invalidexit	**exit** not in loop
invalidfileaccess	Unacceptable access string
invalidfont	Invalid **Font** resource name or font or CIDFont dictionary
invalidrestore	Improper **restore**
ioerror	Input/output error
limitcheck	Implementation limit exceeded
nocurrentpoint	Current point undefined
rangecheck	Operand out of bounds
stackoverflow	Operand stack overflow
stackunderflow	Operand stack underflow
syntaxerror	PostScript language syntax error
timeout	Time limit exceeded
typecheck	Operand of wrong type
undefined	Name not known
undefinedfilename	File not found
undefinedresource	Resource instance not found
undefinedresult	Overflow, underflow, or meaningless result
unmatchedmark	Expected mark not on stack
unregistered	Internal error
VMerror	Virtual memory exhausted

8.2 Operator Details

[– **[** *mark*

pushes a mark object on the operand stack (the same as the **mark** and **<<** operators). The customary use of the **[** operator is to mark the beginning of an indefinitely long sequence of objects that will eventually be formed into a new array object by a matching **]** operator. See the discussion of array syntax in Section 3.2, "Syntax," and of array construction in Section 3.6, "Overview of Basic Operators."

Errors: **stackoverflow**
See Also: **], <<, mark, array, astore**

] *mark* obj_0 … obj_{n-1} **]** *array*

creates a new array of *n* elements (where *n* is the number of elements above the topmost mark on the operand stack), stores those elements into the array, and returns the array on the operand stack. The **]** operator stores the topmost object from the stack into element $n-1$ of *array* and the bottommost one (the one immediately above the mark) into element 0 of *array*. It removes all the array elements from the stack, as well as the mark object.

The array is allocated in local or global VM according to the current VM allocation mode. An **invalidaccess** error occurs if the array is in global VM and any of the objects $obj_0 \ldots obj_{n-1}$ are in local VM. See Section 3.7.2, "Local and Global VM."

Examples

```
[5 4 3]      ⇒  % A three-element array, with elements 5, 4, 3
mark 5 4 3 counttomark array astore exch pop   ⇒  % Same as above

[1 2 add]    ⇒  % A one-element array, with element 3
```

The first two lines of code above have the same effect, but the second line uses lower-level array and stack manipulation primitives instead of **[** and **]**.

In the last example, note that the PostScript interpreter acts on all of the array elements as it encounters them (unlike its behavior with the { … } syntax for executable array construction), so the **add** operator is executed before the array is constructed.

Errors: **stackoverflow, unmatchedmark, VMerror**
See Also: **[, mark, array, astore**

<< – **<<** *mark*



pushes a mark object on the operand stack (the same as the **mark** and **[** operators).

Errors: **stackoverflow**
See Also: **>>, mark**

>> *mark key*$_1$ *value*$_1$ … *key*$_n$ *value*$_n$ **>>** *dict*



creates and returns a dictionary containing the specified key-value pairs. The operands are a mark followed by an even number of objects, which the operator uses alternately as keys and values to be inserted into the dictionary. The dictionary is allocated space for precisely the number of key-value pairs supplied.

The dictionary is allocated in local or global VM according to the current VM allocation mode. An **invalidaccess** error occurs if the dictionary is in global VM and any keys or values are in local VM (see Section 3.7.2, "Local and Global VM"). A **rangecheck** error occurs if there is an odd number of objects above the topmost mark on the stack.

The **>>** operator is equivalent to the following code:

```
counttomark 2 idiv
dup dict
begin
    {def} repeat
    pop
    currentdict
end
```

Example

```
<< /Duplex true
    /PageSize [612 792]
    /Collate false
>> setpagedevice
```

This example constructs a dictionary containing three key-value pairs, which it immediately passes to the **setpagedevice** operator.

Errors: **invalidaccess, rangecheck, typecheck, unmatchedmark, VMerror**
See Also: **<<, mark, dict**

= *any* **=** –

pops an object from the operand stack, produces a text representation of that object's value, and writes the result to the standard output file, followed by a newline character. The text is that produced by the **cvs** operator; thus, **=** prints the value of a number, boolean, string, name, or operator object and prints --nostringval-- for an object of any other type.

The name **=** is not special. In PostScript programs it must be delimited by white-space or special characters the same as names composed of alphabetical characters. The value of **=** is not an operator, but rather a built-in procedure.

Errors: **stackunderflow**
See Also: **==, stack, cvs, print, flush**

== *any* **==** –

pops an object from the operand stack, produces a text representation of that object, and writes the result to the standard output file, followed by a newline character. This operator attempts to produce a result that resembles the PostScript syntax for creating the object. It precedes literal names with /, brackets strings with (…), and expands the values of arrays and packed arrays and brackets them with […] or { … }. For an object with no printable representation, **==** produces the name of its type in the form -*type*-, such as -mark- or -dict-. For an operator object, it produces the operator's name in the form --*operator*--, such as --add--.

The name **==** is not special. In PostScript programs it must be delimited by white-space or special characters the same as names composed of alphabetical characters. The value of **==** is not an operator, but rather a built-in procedure.

The **==** operator is intended for convenience in debugging. The details of how this operator formats its output are intentionally unspecified. A program requiring detailed control over output format should do its own formatting explicitly, using lower-level operators such as **cvs**. Also, the LanguageLevel 2 operators **printobject** and **writeobject** may be more suitable for generating machine-readable output.

Errors: **stackunderflow**
See Also: **=, print, pstack, flush**

$error – **error** *dict*

pushes the dictionary object **$error** on the operand stack (see Section 3.11.2, "Error Handling"). **$error** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **errordict**

abs num_1 **abs** num_2

returns the absolute value of num_1. The type of the result is the same as the type of num_1 unless num_1 is the smallest (most negative) integer, in which case the result is a real number.

Examples

```
 4.5 abs  ⇒  4.5
–3 abs    ⇒  3
 0 abs    ⇒  0
```

Errors: **stackunderflow, typecheck**
See Also: **neg**

add num_1 num_2 **add** *sum*

returns the sum of num_1 and num_2. If both operands are integers and the result is within integer range, the result is an integer; otherwise, the result is a real number.

Examples

```
3 4 add      ⇒  7
9.9 1.1 add  ⇒  11.0
```

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **div, mul, sub, idiv, mod**

addglyph *metrics bitmap cid cidfont* **addglyph** – (***BitmapFontInit*** *procedure set)*



loads into the font cache the glyph bitmap (and metrics) for the character identified by *cid* in *cidfont*, a Type 4 CIDFont. It replaces the existing bitmap for that character, if any. See "Type 4 CIDFonts" on page 379.

The *metrics* operand is a 6- or 10-number array, $[w_x\, w_y\, ll_x\, ll_y\, ur_x\, ur_y]$ or $[w0_x\, w0_y\, ll_x\, ll_y\, ur_x\, ur_y\, w1_x\, w1_y\, v_x\, v_y]$, whose elements represent the glyph metrics and have the same meaning as the operands to **setcachedevice** and **setcachedevice2**, respectively.

The *bitmap* operand is a string object that contains the bitmap data. The bitmap representation is the normal PostScript representation for a 1-bit-per-pixel image. Logically, this image is painted in glyph space with its (0, 0) corner coinciding with (ll_x, ll_y).

When the CTM is the transformation matrix for which the font was designed, the transformation from glyph space to device space is the identity transformation. Thus, the image is treated as a device-resolution bitmap, positioned with the image origin at (ll_x, ll_y) relative to the current point.

A **rangecheck** error occurs if ur_x is less than ll_x or ur_y is less than ll_y, if the image dimensions implied by these values are inconsistent with the length of the bitmap string, or if *cid* is outside the valid range of CIDs (see Appendix B). A **limitcheck** error occurs if the glyph cannot be placed in the font cache, either because it is too large or because the cache is full.

Errors: **invalidfont, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **removeall, removeglyphs**

aload *array* **aload** any_0 … any_{n-1} *array*
packedarray **aload** any_0 … any_{n-1} *packedarray*

successively pushes all *n* elements of *array* or *packedarray* on the operand stack (where *n* is the length of the operand), and then pushes the operand itself.

Example

```
[23 (ab) –6] aload   ⇒   23 (ab) –6 [23 (ab) –6]
```

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **astore, get, getinterval**

anchorsearch *string seek* **anchorsearch** *post match true* *(if found)*
string false *(if not found)*

determines whether the string *seek* matches the initial substring of *string* (that is, whether *string* is at least as long as *seek* and the corresponding characters are equal). If it matches, **anchorsearch** splits *string* into two segments—*match*, the portion of *string* that matches *seek*, and *post*, the remainder of *string*—and returns the string objects *post* and *match* followed by the boolean value *true*. Otherwise, it returns the original *string* followed by *false*. **anchorsearch** is a special case of the **search** operator.

Examples

```
(abbc) (ab) anchorsearch  ⇒  (bc) (ab) true
(abbc) (bb) anchorsearch  ⇒  (abbc) false
(abbc) (bc) anchorsearch  ⇒  (abbc) false
(abbc) (B) anchorsearch   ⇒  (abbc) false
```

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **search, token**

and $bool_1$ $bool_2$ **and** $bool_3$
int_1 int_2 **and** int_3

returns the logical conjunction of the operands if they are boolean. If the operands are integers, **and** returns the bitwise "and" of their binary representations.

Examples

```
true true and     ⇒  true        % A complete truth table
true false and    ⇒  false
false true and    ⇒  false
false false and   ⇒  false

99 1 and          ⇒  1
52 7 and          ⇒  4
```

Errors: **stackunderflow, typecheck**
See Also: **or, xor, not, true, false**

arc *x y r angle*$_1$ *angle*$_2$ **arc** –

appends an arc of a circle to the current path, possibly preceded by a straight line segment. The arc is centered at coordinates (*x*, *y*) in user space, with radius *r*. The operands *angle*$_1$ and *angle*$_2$ define the endpoints of the arc by specifying the angles of the vectors joining them to the center of the arc. The angles are measured in degrees counterclockwise from the positive *x* axis of the current user coordinate system (see Figure 8.1).

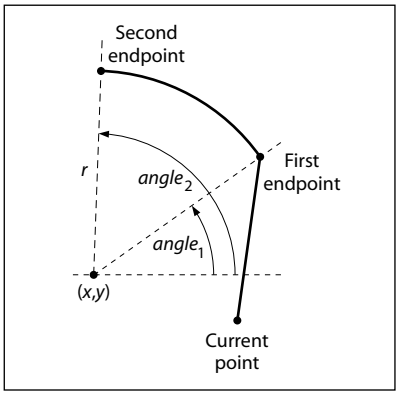


FIGURE 8.1 ***arc*** *operator*

The arc produced is circular in user space. If user space is scaled nonuniformly (that is, differently in the *x* and *y* dimensions), the resulting curve will be elliptical in device space.

If there is a current point, a straight line segment from the current point to the first endpoint of the arc is added to the current path preceding the arc itself. If the current path is empty, this initial line segment is omitted. In either case, the second endpoint of the arc becomes the new current point.

If *angle*$_2$ is less than *angle*$_1$, it is increased by multiples of 360 until it becomes greater than or equal to *angle*$_1$. No other adjustments are made to the two angles. In particular, the angle subtended by the arc is not reduced modulo 360; if the difference *angle*$_2$ − *angle*$_1$ exceeds 360, the resulting path will trace portions of the circle more than once.

The arc is represented internally by one or more cubic Bézier curves (see **curveto**) approximating the required shape. This is done with sufficient accuracy to produce a faithful rendition of the required arc. However, a program that reads the

constructed path using **pathforall** will encounter **curveto** segments where arcs were specified originally.

Example

```
newpath
    0 0 moveto
    0 0 1 0 45 arc
closepath
```

This example constructs a 45-degree "pie slice" with a 1-unit radius, centered at the coordinate origin (see Figure 8.2).

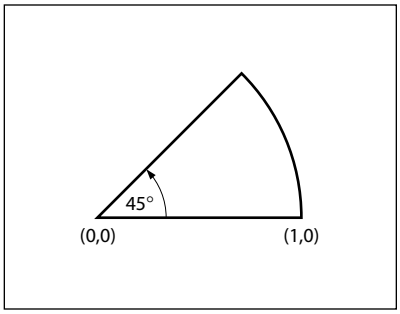


FIGURE 8.2 ***arc*** *operator example*

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **arcn, arct, arcto, curveto**

arcn *x y r angle*$_1$ *angle*$_2$ **arcn** –

(arc negative) appends an arc of a circle to the current path, possibly preceded by a straight line segment. Its behavior is identical to that of **arc**, except that the angles defining the endpoints of the arc are measured clockwise from the positive *x* axis of the user coordinate system, rather than counterclockwise. If *angle*$_2$ is greater than *angle*$_1$, it is decreased by multiples of 360 until it becomes less than or equal to *angle*$_1$.

Example

```
newpath
    0 0 2 0 90 arc
    0 0 1 90 0 arcn
closepath
```

This example constructs a 90-degree "windshield wiper swath" 1 unit wide, with an outer radius of 2 units, centered at the coordinate origin (see Figure 8.3).

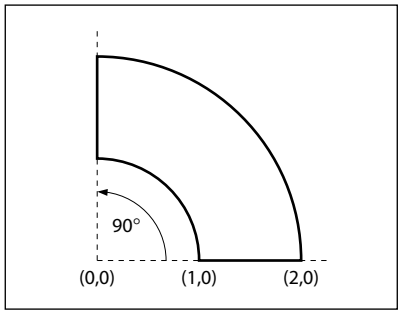


FIGURE 8.3 ***arcn*** *operator example*

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **arc, arct, arcto, curveto**

arct $x_1\ y_1\ x_2\ y_2\ r$ **arct** –



appends an arc of a circle to the current path, possibly preceded by a straight line segment. The arc is defined by a radius r and two tangent lines, drawn from the current point (x_0, y_0) to (x_1, y_1) and from (x_1, y_1) to (x_2, y_2). The center of the arc is located within the inner angle formed by the two tangent lines (see Figure 8.4), and is the only point located at a perpendicular distance r from both lines. The arc begins at the tangent point (xt_1, yt_1) on the first tangent line, passes between the center and the point (x_1, y_1), and ends at the tangent point (xt_2, yt_2) on the second tangent line. If the current point is undefined, a **nocurrentpoint** error occurs.

The arc produced is circular in user space. If user space is scaled nonuniformly (that is, differently in the x and y dimensions), the resulting curve will be elliptical in device space.

If the first tangent point (xt_1, yt_1) is different from the current point (x_0, y_0), a straight line segment joining those two points is added to the current path preceding the arc. In any event, the second tangent point (xt_2, yt_2) becomes the new current point.

If the two tangent lines are collinear, the points (xt_1, yt_1) and (xt_2, yt_2) are identical. In this case, the joining arc has length 0 and **arct** merely appends to the current path a straight line segment from (x_0, y_0) to (x_1, y_1).

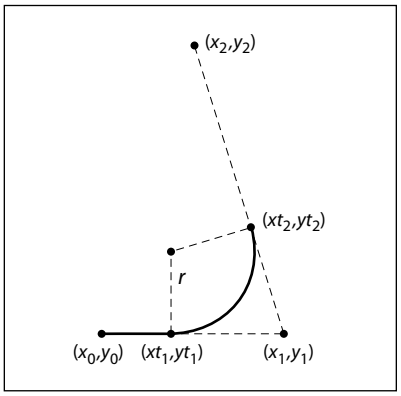


FIGURE 8.4 ***arct*** *operator*

Example

```
newpath
0 0 moveto
0 4 4 4 1 arct
4 4 lineto
```

This example constructs a right angle 4 units wide and 4 units high, with a rounded corner of radius 1 unit (see Figure 8.5).

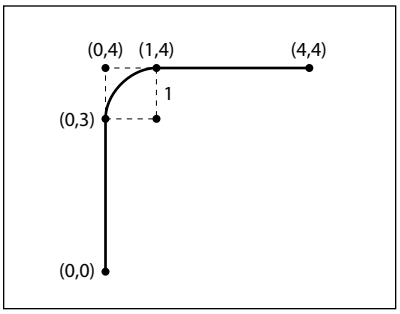


FIGURE 8.5 ***arct*** *operator example*

Errors: **limitcheck, nocurrentpoint, rangecheck, stackunderflow, typecheck, undefinedresult**
See Also: **arc, arcn, arcto, curveto**

arcto $x_1\ y_1\ x_2\ y_2\ r$ **arcto** $xt_1\ yt_1\ xt_2\ yt_2$

appends an arc of a circle to the current path, possibly preceded by a straight line segment. Its behavior is identical to that of **arct**, except that it also returns the user space coordinates of the two tangent points (xt_1, yt_1) and (xt_2, yt_2) on the operand stack.

arcto is not allowed as an element of a user path (see Section 4.6, "User Paths"), whereas **arct** is allowed.

Errors: **limitcheck, nocurrentpoint, stackunderflow, typecheck, undefinedresult**
See Also: **arc, arcn, arct, curveto**

array *int* **array** *array*

creates an array of length *int*, each of whose elements is initialized with a null object, and pushes this array on the operand stack. The *int* operand must be a nonnegative integer not greater than the maximum allowable array length (see Appendix B). The array is allocated in local or global VM according to the current VM allocation mode (see Section 3.7.2, "Local and Global VM").

Example

```
3 array  ⇒  [null null null]
```

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **[,], aload, astore, packedarray**

ashow $a_x\ a_y$ *string* **ashow** –

paints glyphs for the characters of *string* in a manner similar to **show**; however, while doing so, **ashow** adjusts the width of each glyph shown by adding a_x to the glyph's x width and a_y to its y width, thus modifying the spacing between glyphs. The numbers a_x and a_y are x and y displacements in the user coordinate system, not in the glyph coordinate system.

This operator enables fitting a string of text to a specific width by adjusting all the spacing between glyphs by a uniform amount. For a discussion of glyph widths, see Section 5.4, "Glyph Metric Information."

Example

```
/Helvetica findfont 12 scalefont setfont
14 61 moveto (Normal spacing) show
14 47 moveto 4 0 (Wide spacing) ashow
```

Normal spacing
Wide spacing

Errors: **invalidaccess, invalidfont, nocurrentpoint, stackunderflow, typecheck**
See Also: **show, awidthshow, cshow, kshow, widthshow, xshow, xyshow, yshow**

astore *any*$_0$ … *any*$_{n-1}$ *array* **astore** *array*

stores the objects *any*$_0$ to *any*$_{n-1}$ from the operand stack into *array*, where *n* is the length of *array*. The **astore** operator first removes the *array* operand from the stack and determines its length. It then removes that number of objects from the stack, storing the topmost one into element $n - 1$ of *array* and the bottommost one into element 0. Finally, it pushes *array* back on the stack. Note that an **astore** operation cannot be performed on packed arrays.

If the value of *array* is in global VM and any of the objects *any*$_0$ through *any*$_{n-1}$ are composite objects whose values are in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Example

```
(a) (bcd) (ef) 3 array astore  ⇒  [(a) (bcd) (ef)]
```

This example creates a three-element array, stores the strings a, bcd, and ef into it as elements 0, 1, and 2, and leaves the array object on the operand stack.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **aload, put, putinterval**

atan *num den* **atan** *angle*

returns the angle (in degrees between 0 and 360) whose tangent is *num* divided by *den.* Either *num* or *den* may be 0, but not both. The signs of *num* and *den* determine the quadrant in which the result will lie: a positive *num* yields a result in the positive *y* plane, while a positive *den* yields a result in the positive *x* plane. The result is a real number.

Examples

```
0 1 atan      ⇒   0.0
1 0 atan      ⇒   90.0
–100 0 atan   ⇒   270.0
4 4 atan      ⇒   45.0
```

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **cos, sin**

awidthshow *c_x c_y char a_x a_y string* **awidthshow** –

paints glyphs for the characters of *string* in a manner similar to **show**, but combines the special effects of **ashow** and **widthshow**. **awidthshow** adjusts the width of each glyph shown by adding a_x to its *x* width and a_y to its *y* width, thus modifying the spacing between glyphs. Furthermore, **awidthshow** modifies the width of each occurrence of the glyph for the character *char* by an additional amount (c_x, c_y). The interpretation of *char* is as described for the **widthshow** operator.

This operator enables fitting a string of text to a specific width by adjusting the spacing between all glyphs by a uniform amount, while independently controlling the width of the glyph for a specific character, such as the space character. For a discussion of glyph widths, see Section 5.4, "Glyph Metric Information."

Example

```
/Helvetica findfont 12 scalefont setfont
30 60 moveto (Normal spacing) show
30 46 moveto 6 0 8#040 .5 0 (Wide spacing) awidthshow
```

Normal spacing
Wide spacing

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **ashow, cshow, kshow, show, widthshow, xshow, xyshow, yshow**

begin *dict* **begin** –

pushes *dict* on the dictionary stack, making it the current dictionary and installing it as the first of the dictionaries consulted during implicit name lookup and by **def**, **load**, **store**, and **where**.

Errors: **dictstackoverflow, invalidaccess, stackunderflow, typecheck**
See Also: **end, countdictstack, dictstack**

beginbfchar *n* **beginbfchar** – *(**CIDInit** procedure set)*



starts a mapping (completed by **endbfchar**) from *n* individual character codes to character codes or names (which usually select characters in base fonts); see Section 5.11.4, "CMap Dictionaries."

beginbfrange *n* **beginbfrange** – *(**CIDInit** procedure set)*



starts a mapping (completed by **endbfrange**) from *n* character code ranges to character codes or names (which usually select characters in base fonts); see Section 5.11.4, "CMap Dictionaries."

begincidchar *n* **begincidchar** – *(**CIDInit** procedure set)*



starts a mapping (completed by **endcidchar**) from *n* individual character codes to CIDs; see Section 5.11.4, "CMap Dictionaries."

begincidrange *n* **begincidrange** – *(**CIDInit** procedure set)*



starts a mapping (completed by **endcidrange**) from *n* character code ranges to CIDs; see Section 5.11.4, "CMap Dictionaries."

begincmap – **begincmap** – *(**CIDInit** procedure set)*



denotes the start of a CMap definition (ended by **endcmap**); see Section 5.11.4, "CMap Dictionaries."

begincodespacerange *n* **begincodespacerange** – *(**CIDInit** procedure set)*



begins the definition of *n* codespace ranges (completed by **endcodespacerange**); see Section 5.11.4, "CMap Dictionaries."

beginnotdefchar *n* **beginnotdefchar** – *(**CIDInit** procedure set)*



starts a notdef mapping (completed by **endnotdefchar**) from *n* individual character codes to CIDs; see Section 5.11.4, "CMap Dictionaries."

beginnotdefrange *n* **beginnotdefrange** – *(**CIDInit** procedure set)*



starts a notdef mapping (completed by **endnotdefrange**) from *n* character code ranges to CIDs; see Section 5.11.4, "CMap Dictionaries."

beginrearrangedfont *key array* **beginrearrangedfont** – *(**CIDInit** procedure set)*



begins the definition (ended by **endrearrangedfont**) of a rearranged font identified by *key*, using the component fonts listed in *array*; see Section 5.11.4, "CMap Dictionaries."

beginusematrix *fontnum* **beginusematrix** – *(**CIDInit** procedure set)*



denotes the start of a transformation matrix (completed by **endusematrix**) to use with font *fontnum*; see Section 5.11.4, "CMap Dictionaries."

bind *proc* **bind** *proc*

replaces executable operator names in *proc* by their values. For each element of *proc* that is an executable name, **bind** looks up the name in the context of the current dictionary stack as if by the **load** operator. If the name is found and its value is an operator object, **bind** replaces the name with the operator in *proc*. If the name is not found or its value is not an operator, **bind** does not make a change.

For each procedure object contained within *proc*, **bind** applies itself recursively to that procedure, makes the procedure read-only, and stores it back into *proc*. **bind** applies to both arrays and packed arrays, but it treats their access attributes differently. It will ignore a read-only array; that is, it will neither bind elements of the array nor examine nested procedures. On the other hand, **bind** will operate on a packed array (which always has read-only or even more restricted access), disregarding its access attribute. No error occurs in either case.

The effect of **bind** is that all operator names in *proc* and in procedures nested within *proc* to any depth become tightly bound to the operators themselves. During subsequent execution of *proc*, the interpreter encounters the operators themselves rather than their names. See Section 3.12, "Early Name Binding."

In LanguageLevel 3, if the user parameter **IdiomRecognition** is *true*, then after replacing executable names with operators, **bind** compares *proc* with every template procedure defined in instances of the **IdiomSet** resource category. If it finds a match, it returns the associated substitute procedure. See Section 3.12.1, "bind Operator."

Errors: **typecheck**
See Also: **load**

bitshift

int$_1$ *shift* **bitshift** *int*$_2$

shifts the binary representation of *int*$_1$ left by *shift* bits and returns the result. Bits shifted out are lost; bits shifted in are 0. If *shift* is negative, a right shift by *–shift* bits is performed. This operation produces an arithmetically correct result only for positive values of *int*$_1$. Both *int*$_1$ and *shift* must be integers.

Examples

```
7 3 bitshift      ⇒  56
142 –3 bitshift   ⇒  17
```

Errors: **stackunderflow, typecheck**
See Also: **and, or, xor, not**

bytesavailable

file **bytesavailable** *int*

returns the number of bytes immediately available for reading from *file* without waiting. The result is −1 if end-of-file has been encountered or if the number of bytes available cannot be determined for other reasons.

Errors: **invalidaccess, ioerror, stackunderflow, typecheck**
See Also: **read, readhexstring, readline, readstring**

cachestatus – **cachestatus** *bsize bmax msize mmax csize cmax blimit*

returns measurements of several aspects of the font cache (see Section 5.5, "Font Cache"). This operator reports the current consumption and limit for each of three font cache resources: bytes of bitmap storage (*bsize* and *bmax*), font/matrix combinations (*msize* and *mmax*), and total number of cached glyphs (*csize* and *cmax*). It also reports the limit on the number of bytes occupied by a single cached glyph (*blimit*); glyph bitmaps larger than this are not cached.

Except for *blimit*, which corresponds to the **MaxFontItem** user parameter, the values returned by **cachestatus** cannot be controlled directly by a PostScript program. They will vary as a function of the **MaxFontCache** system parameter, but the behavior is implementation-dependent.

Errors: **stackoverflow**
See Also: **setcachelimit, setsystemparams**

ceiling num_1 **ceiling** num_2

returns the least integer value greater than or equal to num_1. The type of the result is the same as the type of the operand.

Examples

```
3.2 ceiling   ⇒   4.0
–4.8 ceiling  ⇒  –4.0
99 ceiling    ⇒   99
```

Errors: **stackunderflow, typecheck**
See Also: **floor, round, truncate, cvi**

charpath *string bool* **charpath** –

obtains the path for the glyph outlines that would result if *string* were shown at the current point using **show**. Instead of painting the path, however, **charpath** appends it to the current path. This yields a result suitable for general filling, stroking, or clipping (see Sections 4.4, "Path Construction"; 4.5, "Painting"; and 5.1, "Organization and Use of Fonts").

The *bool* operand determines what happens if the glyph path is designed to be stroked rather than filled or outlined. If *bool* is *false*, **charpath** simply appends the glyph path to the current path; the result is suitable only for stroking. If *bool* is *true*, **charpath** applies the **strokepath** operator to the glyph path; the result is suit-

able for filling or clipping, but not for stroking. **charpath** does not produce results for portions of a glyph defined as images or masks rather than as paths.

The outlines of some fonts are protected. (In LanguageLevel 1, this applies to all fonts; in LanguageLevels 2 and 3, it applies only to certain fonts that are explicitly marked as protected.) If the current font is outline-protected, using **charpath** to obtain its outlines causes the **pathforall** and **upath** operators to be disabled for as long as those outlines remain in the current path.

Errors: **invalidfont, limitcheck, nocurrentpoint, stackunderflow, typecheck**
See Also: **show, flattenpath, pathbbox, clip**

clear

⊢ *any*$_1$ … *any*$_n$ **clear** ⊢

pops all objects from the operand stack and discards them.

Errors: none
See Also: **count, cleartomark, pop**

cleardictstack

– **cleardictstack** –

pops all dictionaries off the dictionary stack except for the permanent entries. In LanguageLevel 1, the permanent entries are **systemdict** and **userdict**; in LanguageLevels 2 and 3, they are **systemdict**, **globaldict**, and **userdict**. (In LanguageLevel 1, **cleardictstack** is a procedure defined in **userdict** instead of an operator defined in **systemdict**.)

Errors: none
See Also: **begin, end**

cleartomark

mark obj$_1$ … *obj*$_n$ **cleartomark** –

pops entries from the operand stack repeatedly until it encounters a mark, which it also pops from the stack. *obj*$_1$ through *obj*$_n$ are any objects other than marks.

Errors: **unmatchedmark**
See Also: **clear, mark, counttomark, pop**

clip – **clip** –

intersects the area inside the current clipping path with the area inside the current path to produce a new, smaller clipping path. The nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195) is used to determine what points lie inside the current path, while the inside of the current clipping path is determined by whatever rule was used at the time the path was created.

In general, **clip** produces a new path whose inside (according to the nonzero winding number rule) consists of all areas that are inside both of the original paths. The way this new path is constructed (the order of its segments, whether it self-intersects, and so forth) is not specified. **clip** treats an open subpath of the current path as though it were closed; it does not actually alter the path itself. It is permissible for the current path to be empty. The result of executing **clip** is always a nonempty clipping path, though it may enclose zero area.

There is no way to enlarge the current clipping path (other than by **initclip** or **initgraphics**) or to set a new clipping path without reference to the current one. The recommended way of using **clip** is to bracket the **clip** operation and the sequence of graphics operations to be clipped between **gsave** and **grestore** or between **clipsave** and **cliprestore**. The **grestore** will restore the clipping path that was in effect before the **gsave**. The **setgstate** operator can also be used to reset the clipping path to an earlier state.

Unlike **fill** and **stroke**, **clip** does not implicitly perform a **newpath** operation after it has finished using the current path. Any subsequent path construction operators will append to the current path unless **newpath** is invoked explicitly; this can cause unexpected behavior.

Errors: **limitcheck**
See Also: **eoclip, clippath, initclip, rectclip**

clippath – **clippath** –

sets the current path to the current clipping path. This operator is useful for determining the exact extent of the imaging area on the current output device.

If the current clipping path was set with **clip** or **eoclip**, the path set by **clippath** is generally suitable only for filling or clipping. It is not suitable for stroking, because it may contain interior segments or disconnected subpaths produced by the clipping process.

Example

```
clippath
1 setgray
fill
```

This example erases (fills with white) the area inside the current clipping path.

Errors: none
See Also: **clip, eoclip, initclip, rectclip**

cliprestore – cliprestore –



resets the current clipping path (see Section 4.4.2, "Clipping Path") from the one on the top of the clipping path stack in the current graphics state and pops the clipping path stack, restoring the clipping path in effect at the time of the matching **clipsave** operation. This operator provides a way to restore only the clipping path without affecting any of the other graphics state parameters associated with a **grestore** operation.

If there has been no **clipsave** operation since the most recent unmatched **gsave**, the **cliprestore** operator replaces the clipping path with the one that was in effect at the time of the **gsave** operation. (This also applies to a **gsave** that was performed implicitly by a **save** operation.) Note that the restored clipping path is *not* taken from the top of the clipping path stack associated with the previously saved graphics state, but rather from the clipping path parameter of the saved graphics state itself. If both the current clipping path stack and the graphics state stack are empty (which can occur only during an unencapsulated job), the **cliprestore** operation has no effect.

Errors: none
See Also: **clipsave, gsave, grestore**

clipsave – clipsave –



pushes a copy of the current clipping path (see Section 4.4.2, "Clipping Path") on the clipping path stack in the current graphics state. The saved clipping path can later be restored by a matching **cliprestore** operation.

The **gsave** operator also saves the clipping path as part of the total graphics state; **clipsave** saves only the clipping path, without any of the other graphics state parameters.

Errors: **limitcheck**
See Also: **cliprestore, gsave, grestore**

closefile

file **closefile** –

closes *file*, breaking the association between the file object and the underlying file (see Section 3.8, "File Input and Output"). For an output file, **closefile** first performs a **flushfile** operation. It may also take device-dependent actions, such as truncating a disk file to the current position or transmitting an end-of-file indication. Executing **closefile** on a file that has already been closed has no effect; it does not cause an error.

Errors: **ioerror, stackunderflow, typecheck**
See Also: **file, filter, status**

closepath

– **closepath** –

closes the current subpath by appending a straight line segment connecting the current point to the subpath's starting point, which is generally the point most recently specified by **moveto** (see Section 4.4, "Path Construction").

closepath terminates the current subpath; appending another segment to the current path will begin a new subpath, even if the new segment begins at the endpoint reached by the **closepath** operation. If the current subpath is already closed or the current path is empty, **closepath** does nothing.

Errors: **limitcheck**
See Also: **newpath, moveto, lineto**

colorimage

width height bits/comp matrix
datasrc$_0$ … *datasrc*$_{ncomp-1}$ *multi ncomp* **colorimage** –



paints a sampled color image onto the current page. This description only summarizes the general behavior of the **colorimage** operator; see Section 4.10, "Images," for full details.

The image is a rectangular array of *width* × *height* sample values, each consisting of 1, 3, or 4 color components, as specified by *ncomp*. Each component consists of

bits/comp bits of data; valid values of *bits/comp* are 1, 2, 4, 8, or 12. All components are the same size.

The image is considered to exist in its own source coordinate system, or *image space*. The rectangular boundary of the image has its lower-left corner at coordinates (0, 0) and its upper-right corner at (*width*, *height*). The *matrix* operand defines a transformation from user space to image space.

If *ncomp* is 1, image samples have only one (gray) component; the behavior of **colorimage** is equivalent to that of **image** using the first five operands. If *ncomp* is 3, samples consist of red, green, and blue components; if *ncomp* is 4, they consist of cyan, magenta, yellow, and black components. The 1-, 3-, and 4-component sample values are interpreted according to the **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces, respectively (see Section 4.8.2, "Device Color Spaces"), regardless of the current color space.

The *multi* operand is a boolean value that determines how **colorimage** obtains sample data from its data sources. If *multi* is *false*, there is a single data source, $datasrc_0$; **colorimage** obtains all components from that source, interleaved on a per-sample basis. If *multi* is *true*, there are multiple data sources, $datasrc_0$ through $datasrc_{ncomp-1}$—one for each color component. The data sources may be procedures, strings, or files (including filtered files), but must all be of the same type (see Section 4.10.2, "Sample Representation").

Unlike **image** and **imagemask**, **colorimage** does not have an alternate form in which the parameters are bundled into a single image dictionary operand. In LanguageLevels 2 and 3, given the appropriate image dictionary, the **image** operator can do anything that **colorimage** can do, and much more. For example, **image** can interpret color samples in any color space, whereas **colorimage** is limited to the **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined, undefinedresult**
See Also: **image, imagemask**

composefont *key name array* **composefont** *font*
key string array **composefont** *font*
key dict array **composefont** *font*



creates a composite font dictionary—a CID-keyed font—from the CMap specified by the second operand and the CIDFonts or fonts in *array*. It then performs

the equivalent of a **definefont** operation, associating this dictionary with *key* in the **Font** resource category.

The CMap can be specified by a name or string operand to be looked up (by **findresource**) as a key in the **CMap** resource category, or the operand can be an actual CMap dictionary. Similarly, each element of *array* can be a name or a string to be looked up as a key in the **CIDFont** or **Font** resource category (first in **CIDFont** and then, if not found, in **Font**), or the array element can be an actual CIDFont or font dictionary.

composefont defines entries in the resulting composite font dictionary as follows:

CMap	The specified CMap
Encoding	Identity mapping whose length is the same as **FDepVector** (consecutive integers starting from 0)
FDepVector	CIDFont or font dictionaries specified by *array*
FMapType	9
FontMatrix	Identity transformation
FontName	*key*
FontType	0
WMode	Value of **WMode** in the specified CMap if present, otherwise 0

If the CMap specifies that the glyph space of any of the descendant fonts or CIDFonts is to be transformed, an appropriate **makefont** operation is performed on each such font during the process of building **FDepVector**. The transformation applied is the one specified by **beginusematrix** and **endusematrix** in the CMap.

composefont always creates a new font dictionary, regardless of whether there already exists one made from the same CMap and array of CIDFonts or fonts. A PostScript program should invoke **composefont** before the first use of a CID-keyed font, and then **findfont** each time it needs to access the resulting font.

Errors: **dictfull, invalidaccess, invalidfont, limitcheck, rangecheck, stackunderflow, typecheck, undefinedresource, VMerror**
See Also: **definefont, findfont, makefont, beginusematrix, endusematrix**

concat *matrix* **concat** –

applies the transformation represented by *matrix* to the user coordinate space. **concat** accomplishes this by concatenating *matrix* with the current transformation matrix (CTM); that is, it replaces the CTM with the matrix product *matrix* × CTM (see Section 4.3, "Coordinate Systems and Transformations").

Example

```
[72 0 0 72 0 0] concat
72 72 scale
```

Both lines of code above have the same effect on the user coordinate space.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **concatmatrix, setmatrix, currentmatrix, translate, scale, rotate**

concatmatrix $matrix_1$ $matrix_2$ $matrix_3$ **concatmatrix** $matrix_3$

replaces the value of $matrix_3$ with the matrix product $matrix_1 \times matrix_2$ and pushes the result back on the operand stack. The current transformation matrix is not affected.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **concat, setmatrix, currentmatrix, translate, scale, rotate**

configurationerror *(error)*



A **setpagedevice** or **setdevparams** operator has been executed with a request for a feature that either is not available in the interpreter implementation or is not currently available because of the state of the hardware. For **setpagedevice**, this error is generated only if the **Policies** entry in a page device dictionary specifies that an error should be generated.

When a **configurationerror** is generated, a two-element array called **errorinfo** is placed in **$error**. This array contains the key and value of the request that could not be met. See Section 3.11, "Errors."

See Also: **setpagedevice, setdevparams**

copy *any*$_1$ … *any*$_n$ *n* **copy** *any*$_1$ … *any*$_n$ *any*$_1$ … *any*$_n$

array$_1$ *array*$_2$ **copy** *subarray*$_2$
dict$_1$ *dict*$_2$ **copy** *dict*$_2$
string$_1$ *string*$_2$ **copy** *substring*$_2$
packedarray$_1$ *array*$_2$ **copy** *subarray*$_2$
gstate$_1$ *gstate*$_2$ **copy** *gstate*$_2$

performs two entirely different functions, depending on the type of the topmost operand.

In the first form, where the top element on the operand stack is a nonnegative integer *n*, **copy** pops *n* from the stack and duplicates the top *n* elements on the stack as shown above. This form of **copy** operates only on the objects themselves, not on the values of composite objects.

Examples

```
(a) (b) (c) 2 copy   ⇒   (a) (b) (c) (b) (c)
(a) (b) (c) 0 copy   ⇒   (a) (b) (c)
```

In the other forms, **copy** copies all the elements of the first composite object into the second. The composite object operands must be of the same type, except that a packed array can be copied into an array (and only into an array—**copy** cannot copy into packed arrays, because they are read-only). This form of **copy** copies the *value* of a composite object. This is quite different from **dup** and other operators that copy only the objects themselves (see Section 3.3.1, "Simple and Composite Objects"). However, **copy** performs only one level of copying. It does not apply recursively to elements that are themselves composite objects; instead, the values of those elements become shared.

In the case of arrays or strings, the length of the second object must be at least as great as the first; **copy** returns the initial *subarray* or *substring* of the second operand into which the elements were copied. Any remaining elements of *array*$_2$ or *string*$_2$ are unaffected.

In the case of dictionaries, LanguageLevel 1 requires that *dict*$_2$ have a length (as returned by the **length** operator) of 0 and a maximum capacity (as returned by the **maxlength** operator) at least as great as the length of *dict*$_1$. LanguageLevels 2 and 3 do not impose this restriction, since dictionaries can expand when necessary.

The literal/executable and access attributes of the result are normally the same as those of the second operand. However, in LanguageLevel 1 the access attribute of *dict*$_2$ is copied from that of *dict*$_1$.

If the value of the destination object is in global VM and any of the elements copied from the source object are composite objects whose values are in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Example

```
/a1 [1 2 3] def
a1 dup length array copy  ⇒  [1 2 3]
```

Errors: **invalidaccess, rangecheck, stackoverflow, stackunderflow, typecheck**
See Also: **dup, get, put, putinterval**

copypage – copypage –

transmits the contents of the current page to the current output device, but without performing the additional reinitialization actions that **showpage** performs. Specifically, its behavior differs from that of **showpage** in the following ways:

- **showpage** usually performs the equivalent of an **erasepage** operation after transmitting the page, clearing the contents of raster memory in preparation for the next page. **copypage** performs this step only in LanguageLevel 3; in LanguageLevels 1 and 2, it does not erase the page after transmission.
- **showpage** then always performs the equivalent of an **initgraphics** operation, reinitializing the graphics state for the next page. **copypage** never does this.
- If an **EndPage** procedure is defined in the page device dictionary, **showpage** passes it a reason code of 0 on the operand stack, indicating that it is being called from **showpage**. In LanguageLevels 1 and 2, **copypage** passes a reason code of 1 to inform the **EndPage** procedure that it is being called from **copypage** rather than **showpage**; in LanguageLevel 3, **copypage** passes a reason code of 0, as if the call were coming from **showpage** instead. See Section 6.2.6, "Device Initialization and Page Setup," for more information on **EndPage** procedures and reason codes.

If a device's **BeginPage** or **EndPage** procedure invokes **copypage**, an **undefined** error occurs.

Note that because **copypage** behaves differently in LanguageLevel 3 than in LanguageLevels 1 and 2, some uses of this operator will produce different results depending on LanguageLevel. For example, some old applications used the code

```
n {copypage} repeat
erasepage
```

to produce *n* copies of the current page; in LanguageLevel 3, this will instead produce one copy followed by $n-1$ blank pages. Similarly, applications that used **copypage** to implement forms in LanguageLevels 1 and 2 can no longer do so in LanguageLevel 3; the first page will be printed correctly, showing both fixed and

variable contents, but subsequent pages will show only the variable contents, the fixed contents having been erased.

The use of this operator is discouraged. It is intended primarily as a debugging aid. Routine use of **copypage** as a substitute for **showpage** may *severely* degrade the page throughput of some PostScript devices. To print multiple copies of the same page, use the **NumCopies** page device parameter *(LanguageLevel 2)* or set the value of **#copies** in the current dictionary, as discussed in the description of the **showpage** operator.

Errors: **limitcheck, undefined**
See Also: **showpage, erasepage**

cos *angle* **cos** *real*

returns the cosine of *angle,* which is interpreted as an angle in degrees. The result is a real number.

Examples

```
0 cos    ⇒  1.0
90 cos   ⇒  0.0
```

Errors: **stackunderflow, typecheck**
See Also: **atan, sin**

count ⊢ any_1 … any_n **count** ⊢ any_1 … any_n n

counts the number of items on the operand stack and pushes this count on the operand stack.

Examples

```
clear count          ⇒  0
clear 1 2 3 count    ⇒  1 2 3 3
```

Errors: **stackoverflow**
See Also: **counttomark**

countdictstack – **countdictstack** *int*

counts the number of dictionaries currently on the dictionary stack and pushes this count on the operand stack.

Errors: **stackoverflow**
See Also: **dictstack, begin, end**

countexecstack – **countexecstack** *int*

counts the number of objects on the execution stack and pushes this count on the operand stack.

Errors: **stackoverflow**
See Also: **execstack**

counttomark *mark obj*$_1$ … *obj*$_n$ **counttomark** *mark obj*$_1$ … *obj*$_n$ *n*

counts the number of objects on the operand stack, starting with the top element and continuing down to but not including the first mark encountered. *obj*$_1$ through *obj*$_n$ are any objects other than marks.

Examples

```
1 mark 2 3 counttomark   ⇒  1 mark 2 3 2
1 mark counttomark       ⇒  1 mark 0
```

Errors: **stackoverflow, unmatchedmark**
See Also: **mark, count**

cshow *proc string* **cshow** –



invokes *proc* once for each operation of the character mapping algorithm. **cshow** is intended primarily for use with composite fonts (see Section 5.10, “Composite Fonts”—in particular, Section 5.10.1, “Character Mapping”).

The value of **currentfont** during the execution of *proc* is the base font or CIDFont that the character mapping algorithm selects (unless *proc* calls **setfont** or **selectfont**). When *proc* is invoked, the stack contains three values: the selected character’s code (an integer) and the *x* and *y* components of the width vector for the character’s glyph in the user coordinate system. **cshow** does not paint the

glyph and does not change the current point, although *proc* may do so. When *proc* completes execution, the value of **currentfont** is restored.

cshow takes special measures so that *proc* can execute **show** (or any of its variants) even when the mapping algorithm selects a CIDFont—that is, when the CMap yields a CID. In this case, the character code from *string* (or its last byte, if the code is more than 1 byte long) is put on the stack for *proc*. If *proc* executes a **show** operator (without changing the current font), the string passed to **show** must consist of a single byte equal to the code given to *proc*, or a **rangecheck** error occurs; however, the code is otherwise ignored. The glyph that is actually shown is the one identified by the originally selected CID.

cshow can be used to provide careful positioning of individual glyphs while taking advantage of the composite font mapping machinery of the interpreter. However, it can also be used with a base font; the mapping algorithm for a base font simply selects consecutive characters from the string.

Errors: **invalidaccess, invalidfont, rangecheck, stackunderflow, typecheck**
See Also: **show, ashow, awidthshow, kshow, widthshow, xshow, xyshow, yshow**

currentblackgeneration

– **currentblackgeneration** *proc*



returns the current black-generation function in the graphics state.

Errors: **stackoverflow**
See Also: **setblackgeneration, setundercolorremoval**

currentcacheparams

– **currentcacheparams** *mark size lower upper*



pushes a mark object followed by the current font cache parameters on the operand stack. The number of cache parameters returned is variable (see **setcacheparams**).

Errors: **stackoverflow**
See Also: **setcacheparams, setsystemparams, setuserparams**

currentcmykcolor – **currentcmykcolor** *cyan magenta yellow black*



returns the four components of the current color in the graphics state according to the CMYK (cyan-magenta-yellow-black) color model. The result is determined in various ways, depending on the current color space:

- If the current color space is **DeviceCMYK**, **currentcmykcolor** returns the value of the current color directly.
- If the current color space is **DeviceGray** or **DeviceRGB**, **currentcmykcolor** converts the current color to equivalent CMYK components by the formulas discussed in Section 7.2, "Conversions among Device Color Spaces."
- If the current color space is an **Indexed** space or is a **Separation** or **DeviceN** space with its alternative color space selected, **currentcmykcolor** applies the methods above to the underlying color space.
- For any other color space, **currentcmykcolor** returns 0.0 for all four CMYK components.

Errors: **stackoverflow**
See Also: **setcmykcolor, currentcolorspace, currentcolor, currentgray, currentrgbcolor, currenthsbcolor**

currentcolor – **currentcolor** $comp_1$ … $comp_n$



returns the components of the current color in the graphics state, expressed in the current color space. The number of components returned, *n*, is determined by the color space.

Errors: **stackoverflow**
See Also: **setcolor, setcolorspace**

currentcolorrendering – **currentcolorrendering** *dict*



returns the current CIE-based color rendering dictionary parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setcolorrendering, findcolorrendering**

currentcolorscreen – **currentcolorscreen** *redfreq redang redproc*
greenfreq greenang greenproc
bluefreq blueang blueproc
grayfreq grayang grayproc

– **currentcolorscreen** *redfreq redang redhalftone*
greenfreq greenang greenhalftone
bluefreq blueang bluehalftone
grayfreq grayang grayhalftone



returns the frequency, angle, and spot function for all four color components of the current halftone screen parameter in the graphics state (see Section 7.4, "Halftones"). If the current halftone was established via the **setcolorscreen** operator, the four screens are described independently; if **setscreen** was used instead, the same parameter values are repeated for all four.

If the current halftone was defined via the **sethalftone** operator, **currentcolorscreen** returns a frequency of 60 and an angle of 0 for each of the four screens, along with a halftone dictionary describing the properties of the halftone screen, substituted in place of the spot function.

Errors: **stackoverflow**
See Also: **setcolorscreen, setscreen, sethalftone, currentscreen, currenthalftone**

currentcolorspace – **currentcolorspace** *array*



returns an array containing the family name and parameters of the current color space in the graphics state (see **setcolorspace**). The results are always returned in an array, even if the color space has no parameters and was specified to **setcolorspace** by name.

Errors: **stackoverflow**
See Also: **setcolorspace, setcolor**

currentcolortransfer – **currentcolortransfer** *redproc greenproc blueproc grayproc*



returns the current transfer functions in the graphics state (see Section 7.3, "Transfer Functions") for all four primary color components of the output device (cyan, magenta, yellow, and black). If the current transfer functions were established via the **setcolortransfer** operator, the four transfer functions are returned independently; if **settransfer** was used, the same transfer function is repeated for all four.

Errors: **stackoverflow**
See Also: **setcolortransfer, settransfer, currenttransfer**

currentdash – **currentdash** *array offset*

returns an array and offset defining the current value of the dash pattern parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setdash, stroke**

currentdevparams *string* **currentdevparams** *dict*



returns a dictionary containing the keys and current values of all parameters for the device or other named parameter set identified by *string* (see Section C.4, "Device Parameters"). The returned dictionary is merely a container for key-value pairs. Each execution of **currentdevparams** allocates and returns a new dictionary.

Errors: **stackoverflow, undefined, VMerror**
See Also: **setdevparams**

currentdict – **currentdict** *dict*

pushes the current dictionary (the dictionary on the top of the dictionary stack) on the operand stack. **currentdict** does not pop the dictionary stack; it just pushes a duplicate of its top element on the operand stack.

Errors: **stackoverflow**
See Also: **begin, dictstack**

currentfile – **currentfile** *file*

returns the file object from which the PostScript interpreter is currently or was most recently reading program input—that is, the topmost file object on the execution stack. The returned file has the literal attribute.

If there is no file object on the execution stack, **currentfile** returns an invalid file object that does not correspond to any file. This never occurs during execution of ordinary user programs.

The file returned by **currentfile** is usually but not always the standard input file. An important exception occurs during interactive mode operation (see Section 3.8.3, "Special Files"). In this case, the interpreter does not read directly from the standard input file; instead, it reads from a file representing an edited statement (each statement is represented by a different file).

The **currentfile** operator is useful for obtaining images or other data residing in the program file itself (see the example below). At any given time, this file is positioned at the end of the last PostScript token read from the file by the interpreter. If that token was a number or a name immediately followed by a white-space character, the file is positioned after the white-space character (the first, if there are several); otherwise, it is positioned after the last character of the token.

Example

```
/str 100 string def
currentfile str readline
here is a line of text
pop /textline exch def
```

After execution of this example, the name textline is associated with the string here is a line of text.

Errors: **stackoverflow**
See Also: **exec, run**

currentflat – **currentflat** *num*

returns the current value of the flatness parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setflat, flattenpath**

currentfont – **currentfont** *font*
– **currentfont** *cidfont*

returns the current font or CIDFont dictionary, based on the font parameter in the graphics state. Normally, **currentfont** returns the value of the font parameter, as set by **setfont** or **selectfont** (and also returned by **rootfont**). However, when the font parameter denotes a composite font, and **currentfont** is executed inside the **BuildGlyph**, **BuildChar**, or **CharStrings** procedure of a descendant base font or CIDFont (or inside a procedure invoked by **cshow**), **currentfont** returns the current descendant base font or CIDFont. (Of course, if the procedure calls **setfont** or **selectfont** first, **rootfont** and **currentfont** both return the newly selected font.)

Errors: **stackoverflow**
See Also: **rootfont, selectfont, setfont**

currentglobal – **currentglobal** *bool*



returns the VM allocation mode currently in effect.

Errors: **stackoverflow**
See Also: **setglobal**

currentgray – **currentgray** *num*

returns the gray level equivalent to the current color in the graphics state. The result is determined in various ways, depending on the current color space:

- If the current color space is **DeviceGray**, **currentgray** returns the value of the current color directly.
- If the current color space is **DeviceRGB** or **DeviceCMYK**, **currentgray** converts the current color to an equivalent gray level by the formulas discussed in Section 7.2, "Conversions among Device Color Spaces."
- If the current color space is an **Indexed** space or is a **Separation** space with its alternative color space selected, **currentgray** applies the methods above to the underlying color space.
- For any other color space, **currentgray** returns 0.0.

Errors: **stackoverflow**
See Also: **setgray, currentcolorspace, currentcolor, currenthsbcolor, currentrgbcolor, currentcmykcolor**

currentgstate *gstate* **currentgstate** *gstate*



copies the current graphics state to a gstate (graphics state) object and pushes the result back on the operand stack. An **invalidaccess** error will occur if *gstate* is in global VM and any of the composite objects in the current graphics state are in local VM (see Section 3.7.2, "Local and Global VM"). Such objects might include the current halftone screen, transfer function, or dash pattern. In general, allocating gstate objects in global VM is risky and should be avoided.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **gstate, setgstate**

currenthalftone – **currenthalftone** *halftone*



returns a halftone dictionary describing the current halftone screen parameter in the graphics state. If the current halftone was established via the **setscreen** or **setcolorscreen** operator instead of **sethalftone**, **currenthalftone** fabricates and returns a halftone dictionary of type 1 or 2.

Errors: **stackoverflow, VMerror**
See Also: **sethalftone, setscreen, setcolorscreen, sethalftone, currentscreen, currentcolorscreen**

currenthsbcolor – **currenthsbcolor** *hue saturation brightness*

returns the three components of the current color in the graphics state according to the HSB (hue-saturation-brightness) color model. If the current color space is not **DeviceRGB**, the current color is first converted to RGB as described for the **currentrgbcolor** operator. The resulting RGB color is then converted to HSB form; see the Bibliography for further sources of information on this conversion.

Errors: **stackoverflow**
See Also: **sethsbcolor, currentcolorspace currentcolor, currentgray, currentrgbcolor, currentcmykcolor**

currentlinecap – **currentlinecap** *int*

returns the current value of the line cap parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setlinecap, stroke, currentlinejoin**

currentlinejoin – **currentlinejoin** *int*

returns the current value of the line join parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setlinejoin, stroke, currentlinecap**

currentlinewidth – **currentlinewidth** *num*

returns the current value of the line width parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setlinewidth, stroke**

currentmatrix *matrix* **currentmatrix** *matrix*

replaces the value of *matrix* with the current transformation matrix (CTM) in the graphics state and pushes this modified matrix back on the operand stack (see Section 4.3.2, "Transformations").

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **setmatrix, initmatrix, defaultmatrix, translate, scale, rotate**

currentmiterlimit – **currentmiterlimit** *num*

returns the current value of the miter limit parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setmiterlimit, stroke**

currentobjectformat – **currentobjectformat** *int*



returns the object format parameter currently in effect.

Errors: **stackoverflow**
See Also: **setobjectformat**

currentoverprint – **currentoverprint** *bool*



returns the current value of the overprint parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setoverprint**

currentpacking – **currentpacking** *bool*



returns the array packing mode currently in effect.

Errors: **stackoverflow**
See Also: **setpacking, packedarray**

currentpagedevice – **currentpagedevice** *dict*



returns a read-only dictionary reflecting the current contents of the page device dictionary in the graphics state (see Section 6.1.1, "Page Device Dictionary"). If the current output device is not a page device, the dictionary returned will be empty. **currentpagedevice** creates a new dictionary if necessary. It is unspecified whether parameters that are composite objects are copied from or shared with the original parameters that were given to **setpagedevice**.

Changes made to the hardware state of the output device since the last execution of **setpagedevice**, such as changing paper trays or switch settings, are not immediately reflected in the dictionary returned by **currentpagedevice**. If the execution environment is under the control of a job server (see Section 3.7.7, "Job Execution Environment"), the server sets up a page device dictionary that matches the hardware state before starting each job. At the beginning of each job, therefore, the dictionary returned by **currentpagedevice** correctly matches the current hardware state of the device.

Errors: **stackoverflow, VMerror**
See Also: **setpagedevice**

currentpoint

– **currentpoint** *x y*

returns the *x* and *y* coordinates, in the user coordinate system, of the current point in the graphics state (the trailing endpoint of the current path).

As discussed in Section 4.4.1, "Current Path," points entered into a path are immediately converted to device coordinates by the current transformation matrix (CTM); subsequent modifications to the CTM do not affect existing points. **currentpoint** computes the user space coordinates corresponding to the current point according to the current value of the CTM. Thus, if a current point is set and then the CTM is changed, the coordinates returned by **currentpoint** will be different from those that were originally specified for the point.

If the current point is undefined because the current path is empty, a **nocurrentpoint** error occurs.

Errors: **nocurrentpoint, stackoverflow, undefinedresult**
See Also: **moveto, lineto, curveto, arc**

currentrgbcolor

– **currentrgbcolor** *red green blue*

returns the three components of the current color in the graphics state according to the RGB (red-green-blue) color model. The result is determined in various ways, depending on the current color space:

- If the current color space is **DeviceRGB**, **currentrgbcolor** returns the value of the current color directly.

- If the current color space is **DeviceGray** or **DeviceCMYK**, **currentrgbcolor** converts the current color to equivalent RGB components by the formulas discussed in Section 7.2, "Conversions among Device Color Spaces."
- If the current color space is an **Indexed** space or is a **Separation** or **DeviceN** space with its alternative color space selected, **currentrgbcolor** applies the methods above to the underlying color space.
- For any other color space, **currentrgbcolor** returns 0.0 for all three RGB components.

Errors: **stackoverflow**
See Also: **setrgbcolor, currentcolorspace, currentcolor, currentgray, currenthsbcolor, currentcmykcolor**

currentscreen

– **currentscreen** *frequency angle proc*
– **currentscreen** *frequency angle halftone* *(LanguageLevel 2)*

returns the frequency, angle, and spot function of the current halftone screen parameter in the graphics state (see Section 7.4, "Halftones"), assuming that the halftone was established via the **setscreen** operator. If **setcolorscreen** was used instead, the values returned describe the screen for the gray color component only.

If the current halftone was defined via the **sethalftone** operator, **currentscreen** returns a halftone dictionary describing its properties in place of the spot function. For type 1 halftone dictionaries, the values returned for *frequency* and *angle* are taken from the dictionary's **Frequency** and **Angle** entries; for all other halftone types, **currentscreen** returns a frequency of 60 and an angle of 0.

Errors: **stackoverflow**
See Also: **setscreen, setcolorscreen, sethalftone, currentcolorscreen, currenthalftone**

currentshared

– **currentshared** *bool*



performs the same operation as **currentglobal**. This operator is defined for compatibility with earlier PostScript interpreter implementations.

Errors: **stackoverflow**
See Also: **setglobal, setshared**

currentsmoothness – **currentsmoothness** *num*



returns the current value of the smoothness parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setsmoothness**

currentstrokeadjust – **currentstrokeadjust** *bool*



returns the current value of the stroke adjustment parameter in the graphics state.

Errors: **stackoverflow**
See Also: **setstrokeadjust, setlinewidth**

currentsystemparams – **currentsystemparams** *dict*



returns a dictionary containing the keys and current values of all system parameters that are defined in the implementation. The returned dictionary is merely a container for key-value pairs. Each execution of **currentsystemparams** allocates and returns a new dictionary. See Appendix C for information about specific system parameters.

Errors: **stackoverflow, VMerror**
See Also: **setsystemparams**

currenttransfer – **currenttransfer** *proc*

returns the current transfer function in the graphics state (see Section 7.3, "Transfer Functions"), assuming that the transfer function was established via the **settransfer** operator. If **setcolortransfer** was used instead, only the transfer function for the gray color component is returned.

Errors: **stackoverflow**
See Also: **settransfer, setcolortransfer, currentcolortransfer**

currenttrapparams – **currenttrapparams** *dict* (***Trapping*** *procedure set)*



returns a copy of the current trapping parameter dictionary.

For trapping parameters whose values are dictionaries or arrays, the value returned in the result dictionary is a copy of the original; for parameters whose values are strings, it is a shared reference to the original rather than a copy. The dictionary returned by **currenttrapparams** can be modified and used as an operand to **settrapparams**.

Errors: **stackoverflow**
See Also: **settrapparams, settrapzone**

currentundercolorremoval
– **currentundercolorremoval** *proc*



returns the current undercolor-removal function in the graphics state.

Errors: **stackoverflow**
See Also: **setundercolorremoval, setblackgeneration**

currentuserparams – **currentuserparams** *dict*



returns a dictionary containing the keys and current values of all user parameters that are defined in the implementation. The returned dictionary is a container for key-value pairs. Each execution of **currentuserparams** allocates and returns a new dictionary. See Appendix C for information about specific user parameters.

Errors: **stackoverflow, VMerror**
See Also: **setuserparams**

curveto $x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ **curveto** –

appends a section of a cubic Bézier curve to the current path between the current point (x_0, y_0) and the endpoint (x_3, y_3), using (x_1, y_1) and (x_2, y_2) as the Bézier control points. The endpoint (x_3, y_3) becomes the new current point. If the current point is undefined because the current path is empty, a **nocurrentpoint** error occurs.

The four points (x_0, y_0), (x_1, y_1), (x_2, y_2), and (x_3, y_3) define the shape of the curve geometrically (see Figure 8.6). The curve is always entirely enclosed by the convex quadrilateral defined by the four points. It starts at (x_0, y_0), is tangent to the line from (x_0, y_0) to (x_1, y_1) at that point, and leaves the starting point in that direction. It ends at (x_3, y_3), is tangent to the line from (x_2, y_2) to (x_3, y_3) at that point, and approaches the endpoint from that direction. The lengths of the lines from (x_0, y_0) to (x_1, y_1) and from (x_2, y_2) to (x_3, y_3) represent, in a sense, the "velocity" of the path at the endpoints.

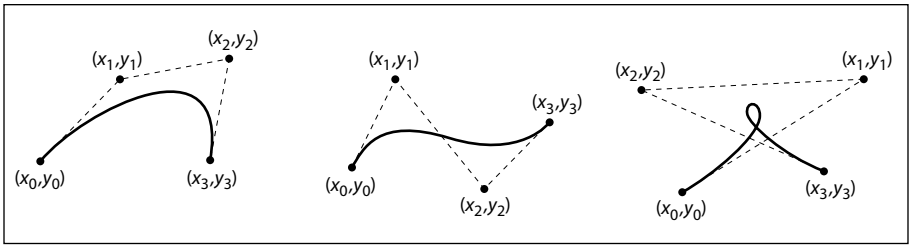


FIGURE 8.6 ***curveto*** *operator*

Mathematically, a cubic Bézier curve is derived from a pair of parametric cubic equations:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + x_0$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + y_0$$

The cubic section produced by **curveto** is the path traced by $x(t)$ and $y(t)$ as t ranges from 0 to 1. The Bézier control points corresponding to this curve are:

$$x_1 = x_0 + c_x/3 \qquad y_1 = y_0 + c_y/3$$

$$x_2 = x_1 + (b_x + c_x)/3 \qquad y_2 = y_1 + (b_y + c_y)/3$$

$$x_3 = x_0 + a_x + b_x + c_x \qquad y_3 = y_0 + a_y + b_y + c_y$$

Errors: **limitcheck, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **moveto, lineto, arcto, arc, arcn, arct**

cvi *num* **cvi** *int*
string **cvi** *int*

(convert to integer) takes an integer, real, or string object from the stack and produces an integer result. If the operand is an integer, **cvi** simply returns it. If the operand is a real number, it truncates any fractional part (that is, rounds it toward 0) and converts it to an integer. If the operand is a string, **cvi** invokes the equivalent of the **token** operator to interpret the characters of the string as a number according to the PostScript syntax rules. If that number is a real number, **cvi** converts it to an integer. A **rangecheck** error occurs if a real number is too large to convert to an integer. (See the **round**, **truncate**, **floor**, and **ceiling** operators, which remove fractional parts without performing type conversion.)

Examples

```
(3.3E1) cvi  ⇒   33
–47.8 cvi    ⇒  –47
520.9 cvi    ⇒  520
```

Errors: **invalidaccess, rangecheck, stackunderflow, syntaxerror, typecheck, undefinedresult**
See Also: **cvr, ceiling, floor, round, truncate**

cvlit *any* **cvlit** *any*

(convert to literal) makes the object on the top of the operand stack have the literal instead of the executable attribute.

Errors: **stackunderflow**
See Also: **cvx, xcheck**

cvn *string* **cvn** *name*

(convert to name) converts the string operand to a name object that is lexically the same as the string. The name object is executable if the string was executable.

Examples

```
(abc) cvn       ⇒  /abc
(abc) cvx cvn   ⇒  abc
```

Errors: **invalidaccess, limitcheck, stackunderflow, typecheck**
See Also: **cvs, type**

cvr *num* **cvi** *real*
string **cvi** *real*

(convert to real) takes an integer, real, or string object and produces a real result. If the operand is an integer, **cvr** converts it to a real number. If the operand is a real number, **cvr** simply returns it. If the operand is a string, **cvr** invokes the equivalent of the **token** operator to interpret the characters of the string as a number according to the PostScript syntax rules. If that number is an integer, **cvr** converts it to a real number.

Errors: **invalidaccess, limitcheck, stackunderflow, syntaxerror, typecheck, undefinedresult**
See Also: **cvi**

cvrs *num radix string* **cvrs** *substring*

(convert with radix to string) produces a text representation of the number *num* in the specified radix, stores the text into *string* (overwriting some initial portion of its value), and returns a string object designating the substring actually used. If *string* is too small to hold the result of the conversion, a **rangecheck** error occurs.

If *radix* is 10, **cvrs** produces the same result as the **cvs** operator when applied to either an integer or a real number. That is, it produces a signed integer or real token that conforms to the PostScript language syntax for that number.

If *radix* is not 10, **cvrs** converts *num* to an integer, as if by the **cvi** operator. Then it treats the machine representation of that integer as an unsigned positive integer and converts it to text form according to the specific radix. The resulting text is not necessarily a valid number. However, if it is immediately preceded by the same radix and #, the combination is a valid PostScript token that represents the same number.

Examples

```
/temp 12 string def
  123 10 temp cvrs    ⇒  (123)
 –123 10 temp cvrs    ⇒  (–123)
  123.4 10 temp cvrs  ⇒  (123.4)
  123 16 temp cvrs    ⇒  (7B)
 –123 16 temp cvrs    ⇒  (FFFFFF85)
  123.4 16 temp cvrs  ⇒  (7B)
```

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **cvs**

cvs *any string* **cvs** *substring*

(convert to string) produces a text representation of an arbitrary object *any*, stores the text into *string* (overwriting some initial portion of its value), and returns a string object designating the substring actually used. If *string* is too small to hold the result of the conversion, a **rangecheck** error occurs.

If *any* is a number, **cvs** produces a string representation of that number. If *any* is a boolean value, **cvs** produces either the string true or the string false. If *any* is a string, **cvs** copies its contents into *string*. If *any* is a name or an operator, **cvs** produces the text representation of that name or the operator's name. If *any* is any other type, **cvs** produces the text --nostringval--.

If *any* is a real number, the precise format of the result string is implementation-dependent and not under program control. For example, the value 0.001 might be represented as 0.001 or as 1.0E–3.

Examples

```
/str 20 string def
123  456  add str cvs   ⇒   (579)
mark str cvs            ⇒   (--nostringval--)
```

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **cvi, cvr, string, type**

cvx *any* **cvx** *any*

(convert to executable) makes the object on the top of the operand stack have the executable instead of the literal attribute.

Errors: **stackunderflow**
See Also: **cvlit, xcheck**

def *key value* **def** –

associates *key* with *value* in the current dictionary—the one on the top of the dictionary stack (see Section 3.4, "Stacks"). If *key* is already present in the current dictionary, **def** simply replaces its value; otherwise, **def** creates a new entry for *key* and stores *value* with it.

If the current dictionary is in global VM and *value* is a composite object whose value is in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Examples

```
/ncnt 1 def                 % Define ncnt to be 1 in current dict
/ncnt ncnt 1 add def        % ncnt now has value 2
```

Errors: **dictfull, invalidaccess, limitcheck, stackunderflow, typecheck, VMerror**
See Also: **store, put**

defaultmatrix

matrix **defaultmatrix** *matrix*

replaces the value of *matrix* with the default transformation matrix for the current output device and pushes this modified matrix back on the operand stack.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **initmatrix, setmatrix, currentmatrix**

definefont

key font **definefont** *font*
key cidfont **definefont** *cidfont*

registers *font* or *cidfont* in the **Font** resource category as an instance associated with *key* (usually a name). **definefont** first checks that *font* or *cidfont* is a well-formed dictionary—in other words, that it contains all entries required in that type of dictionary. It inserts an additional entry whose key is **FID** and whose value is an object of type fontID. It makes the dictionary's access read-only. Finally, it associates *key* with *font* or *cidfont* in the font directory.

definefont distinguishes between a CIDFont and a font by the presence or absence of a **CIDFontType** entry. If the operand is a CIDFont, **definefont** also inserts a **FontType** entry with an appropriate value (see Table 5.11 on page 370).

If the operand is a composite font (see Section 5.10, "Composite Fonts"), **definefont** inserts the entries **EscChar**, **ShiftIn**, and **ShiftOut** if they are not present but are required; it may also insert the implementation-dependent entries **PrefEnc**, **MIDVector**, and **CurMID**. All the descendant fonts must have been registered by **definefont** previously; descendant CIDFonts must have been either registered by **definefont** or defined as CIDFont resource instances (with **defineresource**).

In LanguageLevel 1, the dictionary must be large enough to accommodate all of the additional entries inserted by **definefont**. The font must not have been registered previously, and an **FID** entry must not be present.

In LanguageLevels 2 and 3, a **Font** resource instance can be associated with more than one key. If *font* or *cidfont* has already been registered, **definefont** does not alter it in any way.

Subsequent invocation of **findfont** with *key* will return the same resource instance. Font registration is subject to the normal semantics of virtual memory (see Section 3.7, "Memory Management"). In particular, the lifetime of the definition depends on the VM allocation mode at the time **definefont** is executed. A local definition can be undone by a subsequent **restore** operation.

definefont is actually a special case of **defineresource** operating on the **Font** category. For details, see **defineresource** and Section 3.9, "Named Resources."

Errors: **dictfull, invalidaccess, invalidfont, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **makefont, scalefont, setfont, defineresource, FontDirectory, GlobalFontDirectory, setglobal**

defineresource

key instance category **defineresource** *instance*



associates a resource instance with a resource name in a specified category. *category* is a name object that identifies a resource category, such as **Font** (see Section 3.9.2, "Resource Categories"). *key* is a name or string object that will be used to identify the resource instance. (Names and strings are interchangeable; other types of keys are permitted but are not recommended.) *instance* is the resource instance itself; its type must be appropriate to the resource category.

Before defining the resource instance, **defineresource** verifies that the instance object is the correct type. Depending on the resource category, it may also perform additional validation of the object and may have other side effects (see Section 3.9.2); these side effects are determined by the **DefineResource** procedure in the category implementation dictionary. Finally, **defineresource** makes the object read-only if its access is not already restricted.

The lifetime of the definition depends on the VM allocation mode at the time **defineresource** is executed. If the current VM allocation mode is local (**currentglobal** returns *false*), the effect of **defineresource** is undone by the next nonnested **restore** operation. If the current VM allocation mode is global (**currentglobal** returns *true*), the effect of **defineresource** persists until global VM is restored at the end of the job. If the current job is not encapsulated, the effect of

a global **defineresource** operation persists indefinitely, and may be visible to other execution contexts.

Local and global definitions are maintained separately. If a new resource instance is defined with the same category and key as an existing one, the new definition overrides the old one. The precise effect depends on whether the old definition is local or global and whether the new definition (current VM allocation mode) is local or global. There are two main cases:

- *The new definition is local.* **defineresource** installs the new local definition, replacing an existing local definition if there is one. If there is an existing global definition, **defineresource** does not disturb it. However, the global definition is obscured by the local one. If the local definition is later removed, the global definition reappears.
- *The new definition is global.* **defineresource** first removes an existing local definition if there is one. It then installs the new global definition, replacing an existing global definition if there is one.

defineresource can be used multiple times to associate a given resource instance with more than one key.

If the category name is unknown, an **undefined** error occurs. If the instance is of the wrong type for the specified category, a **typecheck** error occurs. If the instance is in local VM but the current VM allocation mode is global, an **invalidaccess** error occurs; this is analogous to storing a local object into a global dictionary. Other errors can occur for specific categories; for example, when dealing with the **Font** or **CIDFont** category, **defineresource** may execute an **invalidfont** error.

Errors: **invalidaccess, stackunderflow, typecheck, undefined**
See Also: **undefineresource, findresource, resourcestatus, resourceforall**

defineuserobject

index any **defineuserobject** –



establishes an association between the nonnegative integer *index* and the object *any* in the **UserObjects** array (see Section 3.7.6, "User Objects"). First, it creates a **UserObjects** array in **userdict** if one is not already present, or extends an existing **UserObjects** array if necessary. It then executes the equivalent of

```
userdict /UserObjects get
3 1 roll put
```

In other words, it simply stores *any* into the array at the position specified by *index*.

If **defineuserobject** creates or extends the **UserObjects** array, it allocates the array in local VM, regardless of the current VM allocation mode.

The behavior of **defineuserobject** obeys normal PostScript language semantics in all respects. In particular, the modification to the **UserObjects** array and to **userdict**, if any, is immediately visible to all contexts that share the same local VM. It can be undone by a subsequent **restore** operation according to the usual VM rules.

index values must be within the range permitted for arrays; a large *index* value may cause allocation of an array that would exhaust VM resources. Assigning *index* values sequentially starting at 0 is strongly recommended.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **execuserobject, undefineuserobject, UserObjects**

deletefile

filename **deletefile** –



removes the specified file from its storage device (see Section 3.8.2, "Named Files"). If no such file exists, an **undefinedfilename** error occurs. If the device does not allow this operation, an **invalidfileaccess** error occurs. If an environment-dependent error is detected, an **ioerror** occurs.

Errors: **invalidfileaccess, ioerror, stackunderflow, typecheck, undefinedfilename**
See Also: **file, renamefile, status**

dict

int **dict** *dict*

creates an empty dictionary with an initial capacity of *int* elements and pushes the created dictionary object on the operand stack. *int* is expected to be a nonnegative integer. The dictionary is allocated in local or global VM according to the VM allocation mode (see Section 3.7.2, "Local and Global VM").

In LanguageLevel 1, the resulting dictionary has a maximum capacity of *int* elements. Attempting to exceed that limit causes a **dictfull** error.

In LanguageLevels 2 and 3, the *int* operand specifies only the initial capacity; the dictionary can grow beyond that capacity if necessary. The **dict** operator immediately consumes sufficient VM to hold *int* entries. If more than that number of entries are subsequently stored in the dictionary, additional VM is consumed at that time.

There is a cost associated with expanding a dictionary beyond its initial allocation. For efficiency reasons, a dictionary is expanded in chunks rather than one element at a time, so it may contain a substantial amount of unused space. If a program knows how large a dictionary it needs, it should create one of that size initially. On the other hand, if a program cannot predict how large the dictionary will eventually grow, it should choose a small initial allocation sufficient for its immediate needs. The built-in writeable dictionaries (for example, **userdict**) follow the latter convention.

Errors: **limitcheck, stackunderflow, typecheck, VMerror**
See Also: **begin, end, length, maxlength**

dictfull *(error)*

A **def**, **put**, or **store** operation has attempted to define a new entry in a dictionary that is already full—in other words, whose length and maximum capacity (as returned by **length** and **maxlength**) are already equal. This can occur only in LanguageLevel 1, where a dictionary has a fixed limit on the number of entries with distinct keys that it can hold. This limit is established by the operand to the **dict** operator that creates the dictionary.

See Also: **def, put, store, dict**

dictstack *array* **dictstack** *subarray*

stores all elements of the dictionary stack into *array* and returns an object describing the initial *n*-element subarray of *array*, where *n* is the current depth of the dictionary stack. **dictstack** copies the topmost dictionary into element $n - 1$ of *array* and the bottommost one into element 0. The dictionary stack itself is unchanged. If the length of *array* is less than the depth of the dictionary stack, a **rangecheck** error occurs.

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **countdictstack**

dictstackoverflow *(error)*

The dictionary stack has grown too large; too many **begin** operators without corresponding **end** operators have pushed dictionaries on the dictionary stack. See Appendix B for the limit on the size of the dictionary stack.

Before invoking this error, the interpreter creates an array containing all elements of the dictionary stack stored as if by **dictstack**, pushes this array on the operand stack, and resets the dictionary stack to contain only the permanent entries.

See Also: begin, countdictstack, cleardictstack

dictstackunderflow *(error)*

An attempt has been made to remove (with **end**) the bottommost instance of **userdict** from the dictionary stack. This occurs if an **end** operator is executed for which there was no corresponding **begin**.

See Also: end

div num_1 num_2 **div** *quotient*

divides num_1 by num_2, producing a result that is always a real number even if both operands are integers. Use **idiv** instead if the operands are integers and an integer result is desired.

Examples

3 2 div ⇒ 1.5
4 2 div ⇒ 2.0

Errors: stackunderflow, typecheck, undefinedresult
See Also: idiv, add, mul, sub, mod

dtransform *dx dy* **dtransform** *dx′ dy′*
dx dy matrix **dtransform** *dx′ dy′*

(delta transform) applies a transformation matrix to the distance vector (*dx*, *dy*), returning the transformed distance vector (*dx′*, *dy′*). The first form of the operator uses the current transformation matrix in the graphics state to transform the

distance vector from user space to device space coordinates. The second form applies the transformation specified by the *matrix* operand rather than the CTM.

A delta transformation is similar to an ordinary transformation (see Section 4.3, "Coordinate Systems and Transformations"), but does not use the translation components t_x and t_y of the transformation matrix. The distance vectors are thus positionless in both the original and target coordinate spaces, making this operator useful for determining how distances map from user space to device space.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **transform, idtransform**

dup

any **dup** *any any*

duplicates the top element on the operand stack. **dup** copies only the object; the value of a composite object is not copied but is shared. See Section 3.3, "Data Types and Objects."

Errors: **stackoverflow, stackunderflow**
See Also: **copy, index**

echo

bool **echo** –

specifies whether the special files %lineedit and %statementedit are to copy characters from the standard input file to the standard output file. This affects only the behavior of **executive**; it does not apply to normal communication with the PostScript interpreter. **echo** is not defined in products that do not support **executive**. See Sections 2.4.4, "Using the Interpreter Interactively," and 3.8.3, "Special Files."

Errors: **stackunderflow, typecheck**
See Also: **executive, file**

eexec *file* **eexec** –
string **eexec** –

causes the contents of *file* (open for reading) or *string* to be decrypted and then executed in a manner similar to the **exec** operator. The decryption operation does not cause *file* or *string* to be modified.

eexec creates a new file object that serves as a decryption filter on *file* or *string*. It pushes the new file object on the execution stack, making it the current file for the PostScript interpreter. Subsequently, each time the interpreter reads a character from this file, or a program reads explicitly from the file returned by **currentfile**, the decryption filter reads one character from the original *file* or *string* and decrypts it.

The decryption filter file is closed automatically when the end of the original *file* or *string* is encountered, or it can be closed explicitly by **closefile**. If the file passed to **eexec** was the current file, this resumes direct execution of that file with the decryption filter removed. The file may consist of encrypted text followed by unencrypted text if the last thing executed in the encrypted text is currentfile closefile.

Before beginning execution, **eexec** pushes **systemdict** on the dictionary stack, thus ensuring that the operators executed by the encrypted program have their standard meanings. When the decryption filter file is closed either explicitly or implicitly, the dictionary stack is popped. The program must be aware that it is being executed with **systemdict** as the current dictionary; in particular, any definitions that it makes must be into a specific dictionary rather than the current one, since **systemdict** is read-only.

The encrypted file may be represented in either binary or hexadecimal; **eexec** can decrypt it without being told which type it is. The recommended representation is hexadecimal, because hexadecimal data can be transmitted through communication channels that are not completely transparent. Regardless of the representation of the encrypted file, the encryption and decryption processes are transparent—that is, an arbitrary binary file can be encrypted, transmitted as either binary or hexadecimal, and decrypted to yield the original information.

The encryption employed by **eexec** is intended primarily for use in Type 1 font programs. The book *Adobe Type 1 Font Format* contains a complete description of the encryption algorithm and recommended uses of **eexec**.

Errors: **dictstackoverflow, invalidaccess, invalidfileaccess, limitcheck, stackunderflow, typecheck**
See Also: **exec, filter**

end – **end** –

pops the current dictionary off the dictionary stack, making the dictionary below it the current dictionary. If **end** tries to pop the bottommost instance of **userdict**, a **dictstackunderflow** error occurs.

Errors: **dictstackunderflow**
See Also: **begin, dictstack, countdictstack**

endbfchar *srccode*$_1$ *dstcode*$_1$|*dstname*$_1$ … *(**CIDInit** procedure set)*
srccode$_n$ *dstcode*$_n$|*dstname*$_n$ **endbfchar** –



completes the mapping started by **beginbfchar** from individual character codes to character codes or names. The single code specified by the string *srccode*$_1$ is mapped either to a code, *dstcode*$_1$, or to a name, *dstname*$_1$; likewise for the remaining *srccode* values. See Section 5.11.4, "CMap Dictionaries."

endbfrange *srccodelo*$_1$ *srccodehi*$_1$ *dstcodelo*$_1$|*dstnamearray*$_1$ … *(**CIDInit** procedure set)*
srccodelo$_n$ *srccodehi*$_n$ *dstcodelo*$_n$|*dstnamearray*$_n$ **endbfrange** –



completes the mapping started by **beginbfrange** from character code ranges to character codes or names. The codes in the interval specified by the strings *srccodelo*$_1$ through *srccodehi*$_1$ are mapped either to a range of consecutive codes beginning with *dstcodelo*$_1$ or to a sequence of names listed in an array, *dstnamearray*$_1$; likewise for the remaining intervals. See Section 5.11.4, "CMap Dictionaries."

endcidchar *srccode*$_1$ *dstcid*$_1$ … *srccode*$_n$ *dstcid*$_n$ **endcidchar** – *(**CIDInit** procedure set)*



completes the mapping started by **begincidchar** from individual character codes to CIDs. The single code specified by the string *srccode*$_1$ is mapped to an integer CID, *dstcid*$_1$; likewise for the remaining *srccode* values. See Section 5.11.4, "CMap Dictionaries."

endcidrange *srccodelo*$_1$ *srccodehi*$_1$ *dstcidlo*$_1$ … *srccodelo*$_n$ *srccodehi*$_n$ *dstcidlo*$_n$ **endcidrange** – *(**CIDInit** procedure set)*



completes the mapping started by **begincidrange** from character code ranges to CIDs. The codes in the interval specified by the strings *srccodelo*$_1$ through *srccodehi*$_1$ are mapped to a range of consecutive CIDs beginning with the integer *dstcidlo*$_1$; likewise for the remaining intervals. See Section 5.11.4, "CMap Dictionaries."

endcmap – **endcmap** – *(**CIDInit** procedure set)*



denotes the end of the CMap definition started by **begincmap**; see Section 5.11.4, "CMap Dictionaries."

endcodespacerange *srccodelo*$_1$ *srccodehi*$_1$ … *srccodelo*$_n$ *srccodehi*$_n$ **endcodespacerange** – *(**CIDInit** procedure set)*



completes the definition of codespace ranges started by **begincodespacerange**. The codes in the interval specified by the strings *srccodelo*$_1$ through *srccodehi*$_1$ are declared to be valid codes; likewise for the remaining intervals. See Section 5.11.4, "CMap Dictionaries."

endnotdefchar *srccode*$_1$ *dstcid*$_1$ … *srccode*$_n$ *dstcid*$_n$ **endnotdefchar** – *(**CIDInit** procedure set)*



completes the notdef mapping started by **beginnotdefchar** from individual character codes to CIDs. The single code specified by the string *srccode*$_1$ is mapped to an integer CID, *dstcid*$_1$; likewise for the remaining *srccode* values. See Section 5.11.4, "CMap Dictionaries."

endnotdefrange *srccodelo*$_1$ *srccodehi*$_1$ *dstcid*$_1$ … *srccodelo*$_n$ *srccodehi*$_n$ *dstcid*$_n$ **endnotdefrange** – *(**CIDInit** procedure set)*



completes the notdef mapping started by **beginnotdefrange** from character code ranges to CIDs. The codes in the interval specified by the strings *srccodelo*$_1$ through *srccodehi*$_1$ are mapped to a single integer CID, *dstcid*$_1$; likewise for the remaining intervals. See Section 5.11.4, "CMap Dictionaries."

endrearrangedfont – **endrearrangedfont** – *(**CIDInit** procedure set)*



ends the definition of a rearranged font started by **beginrearrangedfont**; see Section 5.11.4, "CMap Dictionaries."

endusematrix *matrix* **endusematrix** – *(**CIDInit** procedure set)*



completes the specification of the transformation matrix started by **beginusematrix**; see Section 5.11.4, "CMap Dictionaries."

eoclip – **eoclip** –

intersects the area inside the current clipping path with the area inside the current path to produce a new, smaller clipping path. The even-odd rule (see "Even-Odd Rule" on page 196) is used to determine what points lie inside the current path, while the inside of the current clipping path is determined by whatever rule was used at the time the path was created. In all other respects, the behavior of **eoclip** is identical to that of **clip**.

Errors: **limitcheck**
See Also: **clip, clippath, initclip, rectclip**

eofill – **eofill** –

paints the area inside the current path with the current color. The even-odd rule is used to determine what points lie inside the path (see "Even-Odd Rule" on page 196). In all other respects, the behavior of **eofill** is identical to that of **fill**.

Errors: **limitcheck**
See Also: **fill, ineofill, ueofill**

eq *any*$_1$ *any*$_2$ **eq** *bool*

pops two objects from the operand stack and pushes *true* if they are equal, or *false* if not. The definition of equality depends on the types of the objects being compared. Simple objects are equal if their types and values are the same. Strings are equal if their lengths and individual elements are equal. Other composite objects (arrays and dictionaries) are equal only if they share the same value. Separate values are considered unequal, even if all the components of those values are the same.

This operator performs some type conversions. Integers and real numbers can be compared freely: an integer and a real number representing the same mathematical value are considered equal by **eq**. Strings and names can likewise be compared freely: a name defined by some sequence of characters is equal to a string whose elements are the same sequence of characters.

The literal/executable and access attributes of objects are not considered in comparisons between objects.

Examples

```
4.0 4 eq              ⇒ true   % A real number and an integer may be equal
(abc) (abc) eq        ⇒ true   % Strings with equal elements are equal
(abc) /abc eq         ⇒ true   % A string and a name may be equal
[1 2 3] dup eq        ⇒ true   % An array is equal to itself
[1 2 3] [1 2 3] eq    ⇒ false  % Distinct array objects are not equal
```

Errors: **invalidaccess, stackunderflow**
See Also: **ne, le, lt, ge, gt**

erasepage – **erasepage** –

erases the current page by painting it with gray level 1.0 (which is ordinarily white, but may be some other color if an atypical transfer function has been defined). The entire page is erased, without reference to the clipping path currently in force. **erasepage** affects only the contents of raster memory; it does not modify the graphics state, nor does it cause a page to be transmitted to the output device.

The **showpage** operator automatically invokes **erasepage** after imaging a page. There are few situations in which a PostScript page description should invoke **erasepage** explicitly, since it affects portions of the page outside the current clipping path. It is usually more appropriate to erase just the area inside the current clipping path (see **clippath**). This allows the page description to be embedded within another, composite page without undesirable effects.

Errors: none
See Also: **showpage, fill, clippath**

errordict – **errordict** *dict*

pushes the dictionary object **errordict** on the operand stack (see Section 3.11, "Errors"). **errordict** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **$error**

exch any_1 any_2 **exch** any_2 any_1

exchanges the top two elements on the operand stack.

Example

```
1 2 exch  ⇒  2 1
```

Errors: **stackunderflow**
See Also: **dup, roll, index, pop**

exec *any* **exec** –

pushes the operand on the execution stack, executing it immediately. The effect of executing an object depends on the object's type and literal/executable attribute; see Section 3.5, "Execution." In particular, executing a literal object will cause it only to be pushed back on the operand stack. Executing a procedure, however, will cause the procedure to be called.

Examples

```
(3 2 add) cvx exec   ⇒  5
3 2 /add exec        ⇒  3 2 /add
3 2 /add cvx exec    ⇒  5
```

In the first example, the string 3 2 add is made executable and then executed. Executing a string causes its characters to be scanned and interpreted according to the PostScript language syntax rules.

In the second example, the literal objects 3, 2, and /add are pushed on the operand stack, then **exec** is applied to /add. Since /add is a literal name, executing it simply causes it to be pushed back on the operand stack. The **exec** operator in this case has no useful effect.

In the third example, the literal name /add on the top of the operand stack is made executable by **cvx**. Applying **exec** to this executable name causes it to be looked up and the **add** operation to be performed.

Errors: **stackunderflow**
See Also: **xcheck, cvx, run**

execform *form* **execform** –



paints a form defined by a form dictionary constructed as described in Section 4.7, "Forms." The graphical output produced by **execform** is defined by the form dictionary's **PaintProc** procedure.

If this is the first invocation of **execform** for *form*, **execform** first verifies that the dictionary contains the required entries. Then it adds an entry to the dictionary with the key **Implementation**, whose value is private to the PostScript interpreter. Finally, it makes the dictionary read-only. (**execform** performs these alterations directly to the operand dictionary; it does not copy the dictionary. These actions succeed even if the dictionary is already read-only.)

When **execform** needs to call the **PaintProc** procedure, it pushes the form dictionary on the operand stack, then executes the equivalent of the following code:

```
gsave                                % Operand stack: dict
   dup /Matrix get concat
   dup /BBox get aload pop           % Stack: dict llx lly urx ury
   exch 3 index sub
   exch 2 index sub                  % Stack: dict llx lly width height
   rectclip                          % Also does a newpath
   dup /PaintProc get                % Stack: dict proc
   exec                              % Execute procedure with dict on stack
grestore
```

The **PaintProc** procedure is expected to consume the dictionary operand and to execute a sequence of graphics operators to paint the form. The **PaintProc** procedure must always produce the same output given the same graphics state parameters, independently of the number of times it is called and independently, for example, of the contents of **userdict**. The PostScript program should not expect any particular invocation of **execform** to cause the specified **PaintProc** procedure to be executed.

The errors listed below are those produced directly by **execform**. The **PaintProc** procedure can, of course, cause other errors.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, undefined, VMerror**
See Also: **findresource**

execstack

array **execstack** *subarray*

stores all elements of the execution stack into *array* and returns an object describing the initial *n*-element subarray of *array*, where *n* is the current depth of the execution stack. **execstack** copies the topmost object into element $n - 1$ of *array* and the bottommost one into element 0 of *array*. The execution stack itself is unchanged. If the length of *array* is less than the depth of the execution stack, a **rangecheck** error occurs.

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **countexecstack, exec**

execstackoverflow *(error)*

The execution stack has grown too large; procedure invocation is nested deeper than the PostScript interpreter permits. See Appendix B for the limit on the size of the execution stack.

See Also: **exec**

execuserobject *index* **execuserobject** –



executes the object associated with the nonnegative integer *index* in the **UserObjects** array (see Section 3.7.6, "User Objects"). **execuserobject** is equivalent to

```
userdict /UserObjects get
exch get exec
```

execuserobject's behavior is similar to that of **exec** or other explicit execution operators. That is, if the object is executable, it is executed; otherwise, it is pushed on the operand stack.

If **UserObjects** is not defined in **userdict** because **defineuserobject** has never been executed, an **undefined** error occurs. If *index* is not a valid index for the existing **UserObjects** array, a **rangecheck** error occurs. If *index* is a valid index but **defineuserobject** has not been executed previously for that index, a null object is returned.

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck, undefined**
See Also: **defineuserobject, undefineuserobject, UserObjects**

executeonly *array* **executeonly** *array*
packedarray **executeonly** *packedarray*
file **executeonly** *file*
string **executeonly** *string*

reduces the access attribute of an array, packed array, file, or string object to execute-only (see Section 3.3.2, "Attributes of Objects"). Access can only be reduced by this operator, never increased. When an object is execute-only, its value cannot be read or modified explicitly by PostScript operators (an **invalidaccess** error will result), but it can still be executed by the PostScript interpreter—for example, by invoking it with the **exec** operator.

executeonly affects the access attribute only of the object that it returns. If there are other composite objects that share the same value, their access attributes are unaffected.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **rcheck, wcheck, xcheck, readonly, noaccess**

executive – executive –

invokes the interactive executive, which facilitates direct user interaction with the PostScript interpreter; see Section 2.4.4, "Using the Interpreter Interactively."

executive uses the special %statementedit file to obtain commands from the user (see Section 3.8.3, "Special Files"). The **echo** operator and the value of **prompt** also affect the behavior of **executive**.

executive is not necessarily defined in all products. It should not be considered a standard part of the PostScript language.

Errors: **undefined**
See Also: **prompt, echo, file**

exit – exit –

terminates execution of the innermost, dynamically enclosing instance of a looping context without regard to lexical relationship. A *looping context* is a procedure invoked repeatedly by one of the following control operators:

cshow	**forall**	**pathforall**
filenameforall	**kshow**	**repeat**
for	**loop**	**resourceforall**

exit pops the execution stack down to the level of that operator. The interpreter then resumes execution at the next object in normal sequence after that operator.

exit does not affect the operand stack or dictionary stack. Any objects pushed on these stacks during execution of the looping context remain after the context is exited.

If **exit** would escape from the context of a **run** or **stopped** operator, an **invalidexit** error occurs (still in the context of the **run** or **stopped**). If there is no enclosing looping context, the interpreter prints an error message and executes the built-in

operator **quit**. This never occurs during execution of ordinary user programs, because they are enclosed by a **stopped** context.

Errors: **invalidexit**
See Also: **stop, stopped**

exitserver *password* **exitserver** – (***serverdict***)

initiates an unencapsulated job, similarly to the operation

true *password* startjob

in LanguageLevel 2 or 3, except that an **invalidaccess** error occurs if *password* is incorrect. See Section 3.7.7, "Job Execution Environment."

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **startjob**

exp *base exponent* **exp** *real*

raises *base* to the *exponent* power. The operands may be either integers or real numbers. If the exponent has a fractional part, the result is meaningful only if the base is nonnegative. The result is always a real number.

Examples

```
 9 0.5 exp  ⇒   3.0
−9 −1 exp   ⇒  −0.111111
```

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **sqrt, ln, log, mul**

false – **false** *false*

pushes a boolean object whose value is *false* on the operand stack. **false** is not an operator; it is a name in **systemdict** associated with the boolean value *false*.

Errors: **stackoverflow**
See Also: **true, and, or, not, xor**

file *filename access* **file** *file*

creates a file object for the file identified by *filename*, accessing it as specified by *access*. Both operands are strings. Conventions for file names and access specifications depend on the operating system environment in which the PostScript interpreter is running. See Section 3.8.2, "Named Files."

Once created and opened, the returned file object remains valid until the file is closed either explicitly (by invoking **closefile**) or implicitly (by encountering end-of-file while reading or executing the file). A file is also closed by **restore** if the file object was created more recently than the **save** snapshot being restored, or is closed by garbage collection if the file object is no longer accessible. There is a limit on the number of files that can be open simultaneously; see Appendix B.

If *filename* is malformed, or if the file does not exist and *access* does not permit creating a new file, an **undefinedfilename** error occurs. If *access* is malformed or the requested access is not permitted by the device, an **invalidfileaccess** error occurs. If the number of files opened by the current context exceeds an implementation limit, a **limitcheck** error occurs. If an environment-dependent error is detected, an **ioerror** occurs.

Examples

```
(%stdin) (r) file   ⇒   % Standard input file object
(myfile) (w) file   ⇒   % Output file object, writing to named file
```

Errors: **invalidfileaccess, ioerror, limitcheck, stackunderflow, typecheck, undefinedfilename**
See Also: **closefile, currentfile, filter, status**

filenameforall *template proc scratch* **filenameforall** –



enumerates all files whose names match the specified *template* string. For each matching file, **filenameforall** copies the file's name into the supplied *scratch* string, pushes a string object designating the substring of *scratch* actually used, and calls *proc*. **filenameforall** does not return any results of its own, but *proc* may do so.

The details of template matching are device-dependent, but the typical convention is that all characters in *template* are case-sensitive and are treated literally, with the exception of the following:

* Matches zero or more consecutive characters.

? Matches exactly one character.

\ Causes the next character of the template to be treated literally, even if it is *, ?, or \. Note that \ is treated as an escape character in a string literal. Thus, if *template* is a string literal, \\ must be used to represent \ in the resulting string.

If *template* does not begin with %, it is matched against device-relative file names of all devices in the search order (see Section 3.8.2, "Named Files"). When a match occurs, the file name passed to *proc* is likewise device-relative—in other words, it does not have a %*device*% prefix.

If *template* does begin with %, it is matched against complete file names in the form %*device*%*file*. Template matching can be performed on *device*, *file*, or both parts of the name. When a match occurs, the file name passed to *proc* is likewise in the complete form %*device*%*file*.

The order of enumeration is unspecified and device-dependent. There are no restrictions on what *proc* can do. However, if *proc* causes new files to be created, it is unspecified whether those files will be encountered later in the same enumeration. Likewise, the set of file names considered for template matching is device-dependent.

Errors: **invalidaccess, ioerror, rangecheck, stackoverflow, stackunderflow, typecheck**
See also: **file, status**

fileposition *file* **fileposition** *position*



returns the current position in an existing open file. The result is a nonnegative integer interpreted as number of bytes from the beginning of the file. If the file object is not valid or the underlying file is not positionable, an **ioerror** occurs.

Errors: **ioerror, stackunderflow, typecheck**
See also: **setfileposition, file**

fill – **fill** –

paints the area inside the current path with the current color. The nonzero winding number rule is used to determine what points lie inside the path (see "Nonzero Winding Number Rule" on page 195).

fill implicitly closes any open subpaths of the current path before painting. Any previous contents of the filled area are obscured, so an area can be erased by filling it with the current color set to white.

After filling the current path, **fill** clears it with an implicit **newpath** operation. To preserve the current path across a **fill** operation, use the sequence

```
gsave
   fill
grestore
```

Errors: **limitcheck**
See Also: **stroke, eofill, ufill, shfill**

filter *datasrc dict* $param_1$ … $param_n$ *filtername* **filter** *file*
datatgt dict $param_1$ … $param_n$ *filtername* **filter** *file*



creates and returns a filtered file (see Sections 3.8.4, "Filters," and 3.13, "Filtered Files Details").

The first operand specifies the underlying data source or data target that the filter is to read or write. It can be a file, a procedure, or a string.

The *dict* operand contains additional parameters that control how the filter is to operate. It can be omitted whenever all dictionary-supplied parameters have their default values for the given filter. The operands $param_1$ through $param_n$ are additional parameters that some filters require as operands rather than in *dict*; most

filters do not require these operands. The number and types of parameters specified in *dict* or as operands depends on the filter name.

The *filtername* operand identifies the data transformation that the filter is to perform. The standard filter names are as follows:

ASCIIHexEncode	**ASCIIHexDecode**
ASCII85Encode	**ASCII85Decode**
LZWEncode	**LZWDecode**
FlateEncode *(LanguageLevel 3)*	**FlateDecode** *(LanguageLevel 3)*
RunLengthEncode	**RunLengthDecode**
CCITTFaxEncode	**CCITTFaxDecode**
DCTEncode	**DCTDecode**
NullEncode	**SubFileDecode**
	ReusableStreamDecode *(LanguageLevel 3)*

An encoding filter is an output (writeable) file; a decoding filter is an input (readable) file. The file object returned by the filter can be used as an operand to normal file input and output operators, such as **read** and **write**. Reading from an input filtered file causes the filter to read from the underlying data source and transform the data. Similarly, writing to an output filtered file causes the filter to transform the data and write it to the underlying data target.

The **filter** operator disregards the current VM allocation mode. Instead, the returned file object is created in global VM if and only if all the composite objects it retains after filter creation are in global VM. These objects include the underlying source or target object, the end-of-data string for **SubFileDecode**, and the *dict* operand of **DCTDecode** or **DCTEncode** (which can contain strings and arrays used during operation of the filter).

Note that the *dict* operand is not retained by filters other than **DCTDecode** and **DCTEncode**. The dictionary is used only as a container for parameters, which are completely processed by the **filter** operator. Therefore, the VM allocation mode of this dictionary is irrelevant.

Errors: **invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **file, closefile, resourceforall**

findcolorrendering *renderingintent* **findcolorrendering** *name bool*



selects a CIE-based color rendering dictionary (CRD) for rendering colors on the current output device according to a specified rendering intent (see Section 7.1.3, "Rendering Intents"). *renderingintent* is a name or string specifying the desired rendering intent. *name* is a name representing a CRD present in the **ColorRendering** resource category. The CRD can then be established as current in the graphics state with the **findresource** and **setcolorrendering** operators.

If *bool* is *true*, the CRD designated by *name* exactly satisfies the requested rendering intent for the current device configuration and halftone. If *bool* is *false*, a CRD exactly satisfying the request is not available; in this case, *name* proposes an alternate CRD instead.

findcolorrendering combines the requested rendering intent with the current device configuration and halftone to form a CRD name of the form

renderingintent.deviceconfig.halftone

where *renderingintent* is taken verbatim from the *renderingintent* operand, and *deviceconfig* and *halftone* are the names returned by calls to the **GetPageDeviceName** and **GetHalftoneName** procedures, respectively, in the **ColorRendering** procedure set.

To ensure that all parameters that may influence the selection of a color rendering dictionary are correctly accounted for, **findcolorrendering** should not be called until after any other operators that can affect either the device configuration or the current halftone.

Errors: **stackoverflow, stackunderflow, typecheck**
See Also: **setcolorrendering, currentcolorrendering, GetPageDeviceName, GetHalftoneName, GetSubstituteCRD**

findencoding *key* **findencoding** *array*



obtains an encoding vector identified by *key* and pushes it on the operand stack. Encoding vectors are described in Section 5.3, "Character Encoding."

findencoding is a special case of **findresource** applied to the **Encoding** category (see Section 3.9, "Named Resources"). If the encoding vector specified by *key* does not exist or cannot be found, an **undefinedresource** error occurs.

Errors: **stackunderflow, typecheck, undefinedresource**
See Also: **findresource, StandardEncoding, ISOLatin1Encoding**

findfont *key* **findfont** *font*
key **findfont** *cidfont*

obtains a **Font** resource instance whose name is *key* and pushes the instance (which may be a font or CIDFont dictionary) on the operand stack (see Section 5.1, "Organization and Use of Fonts"). *key* may be a key previously passed to **definefont**, in which case the **Font** resource instance associated with *key* (in the font directory) is returned.

If the **Font** resource identified by *key* is not defined in virtual memory, **findfont** takes an action that varies according to the environment in which the PostScript interpreter is operating. In some environments, **findfont** may attempt to read a font definition from an external source, such as a file. In other environments, **findfont** substitutes a default font or executes the **invalidfont** error. **findfont** is a special case of **findresource** applied to the **Font** category. See Section 3.9, "Named Resources."

findfont, like **findresource**, normally looks first for **Font** resources defined in local VM, then for those defined in global VM. However, if the current VM allocation mode is global, **findfont** considers only **Font** resources defined in global VM. If **findfont** needs to load a font or CIDFont into VM, it may use either local or global VM; see Section 3.9.2, "Resource Categories," for more information.

findfont is not an operator, but rather a built-in procedure. It may be redefined by a PostScript program that requires different strategies for finding fonts.

Errors: **invalidfont, stackunderflow, typecheck**
See Also: **scalefont, makefont, setfont, selectfont, definefont, findresource, FontDirectory, GlobalFontDirectory**

findresource *key category* **findresource** *instance*



attempts to obtain a named resource instance in a specified category. *category* is a name object that identifies a resource category, such as **Font** (see Section 3.9.2, "Resource Categories"). *key* is a name or string object that identifies the resource instance. (Names and strings are interchangeable; other types of keys are permitted but are not recommended.) If it succeeds, **findresource** pushes the resource instance on the operand stack; this is an object whose type depends on the resource category.

findresource first attempts to obtain a resource instance that has previously been defined in virtual memory by **defineresource**. If the current VM allocation mode is local, **findresource** considers local resource definitions first, then global definitions (see **defineresource**). However, if the current VM allocation mode is global, **findresource** considers only global resource definitions.

If the requested resource instance is not currently defined in VM, **findresource** attempts to obtain it from an external source. The way this is done is not specified by the PostScript language; it varies among different implementations and different resource categories. The effect of this action is to create an object in VM and execute **defineresource**. **findresource** then returns the newly created object. If *key* is not a name or a string, **findresource** will not attempt to obtain an external resource.

When **findresource** loads an object into VM, it may use global VM even if the current VM allocation mode is local. In other words, it may set the VM allocation mode to global (true setglobal) while loading the resource instance and executing **defineresource**. The policy for whether to use global or local VM resides in the **FindResource** procedure for the specific resource category; see Section 3.9.2, "Resource Categories."

During its execution, **findresource** may remove the definitions of resource instances that were previously loaded into VM by **findresource**. The mechanisms and policies for this depend on the category and the implementation; reclamation of resources may occur at times other than during execution of **findresource**. However, resource definitions that were made by explicit execution of **defineresource** are never disturbed by automatic reclamation.

If the specified resource category does not exist, an **undefined** error occurs. If the category exists but there is no instance whose name is *key*, an **undefinedresource** error occurs.

Errors: **stackunderflow, typecheck, undefined, undefinedresource**
See Also: **defineresource, resourcestatus, resourceforall, undefineresource**

flattenpath – flattenpath –

replaces the current path with an equivalent path in which all curved segments are approximated by sequences of straight lines. The precision of the approximation is controlled by the current flatness parameter in the graphics state (see **setflat**). Straight line segments in the original path are not affected. If the current path does not contain any curved segments, **flattenpath** leaves it unchanged.

The flattening of curves to straight line segments is done automatically when a path is used to control painting (for example, by **stroke**, **fill**, **eofill**, or **clip**). Only rarely does a program need to flatten a path explicitly (see **pathbbox**).

Errors: **limitcheck**
See Also: **setflat, curveto, arc, arcn, arct, arcto, pathbbox**

floor

num$_1$ **floor** *num*$_2$

returns the greatest integer value less than or equal to *num*$_1$. The type of the result is the same as the type of the operand.

Examples

```
  3.2 floor  ⇒   3.0
 –4.8 floor  ⇒  –5.0
  99 floor   ⇒   99
```

Errors: **stackunderflow, typecheck**
See Also: **ceiling, round, truncate, cvi**

flush

– **flush** –

causes any buffered characters for the standard output file to be delivered immediately. In general, a program requiring output to be sent immediately, such as during real-time, two-way interactions, should call **flush** after generating that output.

Errors: **ioerror**
See Also: **flushfile, print**

flushfile

file **flushfile** –

If *file* is an output file, **flushfile** causes any buffered characters for that file to be delivered immediately. In general, a program requiring output to be sent immediately, such as during real-time, two-way interactions, should call **flushfile** after generating that output.

If *file* is an input file, **flushfile** reads and discards data from that file until the end-of-file indication is encountered. This is useful during error recovery, and the PostScript job server uses it for that purpose. **flushfile** does not close the file, unless it is a decoding filter file.

Errors: **ioerror, stackunderflow, typecheck**
See Also: **flush, read, write**

FontDirectory – **FontDirectory** *dict*

pushes a dictionary of **Font** resource instances on the operand stack. **FontDirectory** is not an operator; it is a name in **systemdict** associated with the dictionary object.

The **FontDirectory** dictionary associates **Font** resource names with font or CIDFont dictionaries. **definefont** places entries in **FontDirectory**, and **findfont** looks there first. The dictionary is read-only; only **definefont** and **undefinefont** can change it.

Although **FontDirectory** contains all **Font** resources that are currently defined in virtual memory, it does not necessarily describe all the **Font** resources available to a PostScript program. This is because the **findfont** operator can sometimes obtain fonts from an external source and load them into VM dynamically. Consequently, examining **FontDirectory** is not a reliable method of inquiring about available **Font** resources. The preferred method is to use the LanguageLevel 2 operators **resourcestatus** and **resourceforall** to inquire about the **Font** resource category. See Section 3.9, "Named Resources."

In LanguageLevel 2 or 3, when the VM allocation mode is global, the name **FontDirectory** is temporarily rebound to the value of **GlobalFontDirectory**, which contains only those **Font** resources that have been defined in global VM (see Section 3.7.2, "Local and Global VM"). This ensures the correct behavior of fonts that are defined in terms of other fonts.

Errors: **stackoverflow**
See Also: **definefont, undefinefont, findfont, findresource,GlobalFontDirectory**

for *initial increment limit proc* **for** –

executes the procedure *proc* repeatedly, passing it a sequence of values from *initial* by steps of *increment* to *limit*. The **for** operator expects *initial*, *increment*, and *limit* to be numbers. It maintains a temporary internal variable, known as the control variable, which it first sets to *initial*. Then, before each repetition, it compares the control variable to the termination value *limit*. If *limit* has not been exceeded, **for** pushes the control variable on the operand stack, executes *proc*, and adds *increment* to the control variable.

The termination condition depends on whether *increment* is positive or negative. If *increment* is positive, **for** terminates when the control variable becomes greater than *limit*. If *increment* is negative, **for** terminates when the control variable becomes less than *limit*. If *initial* meets the termination condition, **for** does not execute *proc* at all. If *proc* executes the **exit** operator, **for** terminates prematurely.

Usually, *proc* will use the value on the operand stack for some purpose. However, if *proc* does not remove the value, it will remain there. Successive executions of *proc* will cause successive values of the control variable to accumulate on the operand stack.

Examples

```
0 1 1 4 {add} for   ⇒  10
1 2 6 {} for        ⇒  1 3 5
3 –.5 1 {} for      ⇒  3.0 2.5 2.0 1.5 1.0
```

In the first example above, the value of the control variable is added to whatever is on the stack, so 1, 2, 3, and 4 are added in turn to a running sum whose initial value is 0. The second example has an empty procedure, so the successive values of the control variable are left on the stack. The last example counts backward from 3 to 1 by halves, leaving the successive values on the stack.

Beware of using real numbers instead of integers for any of the first three operands. Most real numbers are not represented exactly. This can cause an error to accumulate in the value of the control variable, with possibly surprising results. In particular, if the difference between *initial* and *limit* is a multiple of *increment*, as in the last example, the control variable may not achieve the *limit* value.

Errors: **stackoverflow, stackunderflow, typecheck**
See Also: **repeat, loop, forall, exit**

forall *array proc* **forall** –
packedarray proc **forall** –
dict proc **forall** –
string proc **forall** –

enumerates the elements of the first operand, executing the procedure *proc* for each element. If the first operand is an array, packed array, or string object, **forall** pushes an element on the operand stack and executes *proc* for each element in the object, beginning with the element whose index is 0 and continuing sequentially. In the case of a string, the elements pushed on the operand stack are integers in the range 0 to 255, *not* 1-character strings.

If the first operand is a dictionary, **forall** pushes a key and a value on the operand stack and executes *proc* for each key-value pair in the dictionary. The order in which **forall** enumerates the entries in the dictionary is arbitrary. New entries put in the dictionary during the execution of *proc* may or may not be included in the enumeration. Existing entries removed from the dictionary by *proc* will not be encountered later in the enumeration.

If the first operand is empty (that is, has length 0), **forall** does not execute *proc* at all. If *proc* executes the **exit** operator, **forall** terminates prematurely.

Although **forall** does not leave any results on the operand stack when it is finished, the execution of *proc* may leave arbitrary results there. If *proc* does not remove each enumerated element from the operand stack, the elements will accumulate there.

Examples

```
0 [ 13 29 3 –8 21 ] { add } forall   ⇒   58

/d 2 dict def
d /abc 123 put
d /xyz (test) put
d {} forall                          ⇒   /xyz (test) /abc 123
```

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **for, repeat, loop, exit**

gcheck *any* **gcheck** *bool*



returns *true* if the operand is a simple object, or if it is composite and its value resides in global VM. It returns *false* if the operand is composite and its value resides in local VM. In other words, **gcheck** returns *true* if its operand could legally be stored as an element of another object in global VM. See Section 3.7.2, "Local and Global VM."

Errors: **stackunderflow**

ge num_1 num_2 **ge** *bool*
$string_1$ $string_2$ **ge** *bool*

pops two objects from the operand stack and pushes *true* if the first operand is greater than or equal to the second, or *false* otherwise. If both operands are numbers, **ge** compares their mathematical values. If both operands are strings, **ge** compares them element by element, treating the elements as integers in the range 0 to 255, to determine whether the first string is lexically greater than or equal to the second. If the operands are of other types or one is a string and the other is a number, a **typecheck** error occurs.

Examples

```
4.2 4 ge          ⇒ true
(abc) (d) ge      ⇒ false
(aba) (ab) ge     ⇒ true
(aba) (aba) ge    ⇒ true
```

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **gt, eq, ne, le, lt**

get *array index* **get** *any*
packedarray index **get** *any*
dict key **get** *any*
string index **get** *int*

returns a single element from the value of the first operand. If the first operand is an array, a packed array, or a string, **get** treats the second operand as an index and returns the element identified by the index, counting from 0. *index* must be in the range 0 to $n - 1$, where n is the length of the array, packed array, or string. If it is outside this range, a **rangecheck** error occurs.

If the first operand is a dictionary, **get** looks up the second operand as a key in the dictionary and returns the associated value. If the key is not present in the dictionary, an **undefined** error occurs.

Examples

```
[31 41 59] 0 get              ⇒ 31
[0 (string1) [ ] {add 2 div}]          % A mixed-type array
   2 get                      ⇒ [ ]     % An empty array

/mykey (myvalue) def
currentdict /mykey get        ⇒ (myvalue)

(abc) 1 get                   ⇒ 98      % Character code for b
(a) 0 get                     ⇒ 97
```

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck, undefined**
See Also: **put, getinterval**

GetHalftoneName

– **GetHalftoneName** *name* *(**ColorRendering** procedure set)*



returns the name of the current halftone. This is not an operator but a procedure used by the **findcolorrendering** operator in constructing the name of a color rendering dictionary.

The name is typically taken from the optional **HalftoneName** entry in the current halftone dictionary. If no such entry is present, **GetHalftoneName** may simply return the default name none, or it may make more elaborate efforts to return a meaningful name, such as by constructing one based on the angle and frequency of the current halftone.

Errors: **stackoverflow**
See Also: **findcolorrendering, GetPageDeviceName, GetSubstituteCRD**

getinterval

array index count **getinterval** *subarray*
packedarray index count **getinterval** *subarray*
string index count **getinterval** *substring*

creates a new array, packed array, or string object whose value consists of some subsequence of the original array, packed array, or string. The subsequence consists of *count* elements starting at the specified *index* in the original object. The elements in the subsequence are shared between the original and new objects (see Section 3.3.1, “Simple and Composite Objects”).

The returned subarray or substring is an ordinary array, packed array, or string object whose length is *count* and whose elements are indexed starting at 0. The element at index 0 in the result is the same as the element at *index* in the original object.

getinterval requires *index* to be a valid index in the original object and *count* to be a nonnegative integer such that *index* + *count* is not greater than the length of the original object.

Examples

```
[9 8 7 6 5] 1 3 getinterval  ⇒  [8 7 6]
(abcde) 1 3 getinterval      ⇒  (bcd)
(abcde) 0 0 getinterval      ⇒  ()        % An empty string
```

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **get, putinterval**

GetPageDeviceName

– **GetPageDeviceName** *name* (***ColorRendering*** *procedure set)*



returns a name representing the current configuration of the page device. This is not an operator but a procedure used by the **findcolorrendering** operator in constructing the name of a color rendering dictionary.

The name is typically taken from the optional **PageDeviceName** entry in the page device dictionary. If no such entry is present, **GetPageDeviceName** may simply return the default name none, or it may make more elaborate efforts to return a meaningful name, such as by constructing one based on the current page device parameters.

Errors: **stackoverflow**
See Also: **findcolorrendering, GetHalftoneName, GetSubstituteCRD**

GetSubstituteCRD

renderingintent **GetSubstituteCRD** *name* (***ColorRendering*** *procedure set)*



returns the name of an alternate color rendering dictionary (CRD) when no suitable CRD can be found to satisfy a requested rendering intent, given the current device configuration and halftone. This is not an operator but a procedure used by the **findcolorrendering** operator to propose an alternate CRD when the requested rendering intent cannot be satisfied.

renderingintent is the rendering intent passed to the **findcolorrendering** operator. *name* is the name of an alternate CRD that exists in the **ColorRendering** resource category. After calling **GetSubstituteCRD**, **findcolorrendering** always returns *false* (since the requested CRD could not be found), along with the alternate CRD name received from **GetSubstituteCRD**. (If a suitable CRD is found, **findcolorrendering** returns *true* without calling **GetSubstituteCRD**.) In the event it cannot generate a meaningful CRD substitution, **GetSubstituteCRD** simply returns the name of some built-in CRD, such as DefaultColorRendering.

Errors: **stackoverflow, stackunderflow, typecheck**
See Also: **findcolorrendering, GetHalftoneName, GetPageDeviceName**

globaldict

– **globaldict** *dict*



pushes the dictionary object **globaldict** on the operand stack (see Section 3.7.5, "Standard and User-Defined Dictionaries"). **globaldict** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **systemdict, userdict**

GlobalFontDirectory

– **GlobalFontDirectory** *dict*



pushes a dictionary of **Font** resource instances on the operand stack. Its contents are limited to those **Font** resources that have been defined in global VM. See **FontDirectory** for a complete explanation. **GlobalFontDirectory** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **FontDirectory**

glyphshow *name* **glyphshow** –
cid **glyphshow** –



shows the glyph for a single character from the current font or CIDFont; the character is identified by name if a base font or by CID if a CIDFont. If the current font is a composite (Type 0) font, an **invalidfont** error occurs.

Unlike all other **show** variants, **glyphshow** bypasses the current font's **Encoding** array; it can access any character in the font, whether or not that character's name is present in the font's encoding vector. **glyphshow** is the only **show** variant that works directly with a CIDFont.

For a base font, the behavior of **glyphshow** depends on the current font's **FontType** value. For fonts that contain a **CharStrings** dictionary, such as Type 1 fonts, **glyphshow** looks up *name* there to obtain a glyph description to execute. If *name* is not present in the **CharStrings** dictionary, **glyphshow** substitutes the .notdef entry, which must be present.

For Type 3 fonts, if the font dictionary contains a **BuildGlyph** procedure, **glyphshow** pushes the current font dictionary and *name* on the operand stack and then invokes **BuildGlyph** in the usual way (see Section 5.7, "Type 3 Fonts"). If there is no **BuildGlyph** procedure, but only a **BuildChar** procedure, **glyphshow** searches the font's **Encoding** array for an occurrence of *name*. If it finds one, it pushes the font dictionary and the array index on the operand stack, then invokes **BuildChar** in the usual way. If *name* is not present in the encoding, **glyphshow** substitutes the name .notdef and repeats the search. If .notdef is not present either, an **invalidfont** error occurs.

For a CIDFont, **glyphshow** proceeds as **show** would for a CID-keyed font whose mapping algorithm yields this CIDFont with *cid* as the character selector. A **rangecheck** error occurs if *cid* is outside the valid range of CIDs (see Appendix B).

Except for the means of selecting the character to be shown, **glyphshow** behaves the same as **show**. Like **show**, **glyphshow** can access glyphs that are already in the font cache; **glyphshow** does not always need to execute the character's glyph description.

Errors: **invalidaccess, invalidfont, nocurrentpoint, stackunderflow, typecheck**
See Also: **show**

grestore – **grestore** –

resets the current graphics state from the one on the top of the graphics state stack and pops the graphics state stack, restoring the graphics state in effect at the time of the matching **gsave** operation. This operator provides a simple way to undo complicated transformations and other graphics state modifications without having to reestablish all graphics state parameters individually.

If the topmost graphics state on the stack was saved with **save** rather than **gsave** (that is, if there has been no **gsave** operation since the most recent unmatched **save**), **grestore** restores that topmost graphics state without popping it from the stack. If there is no unmatched **save** (which can happen only during an unencapsulated job) and the graphics state stack is empty, **grestore** has no effect.

Errors: none
See Also: **gsave, grestoreall, save, setgstate**

grestoreall – **grestoreall** –

repeatedly performs **grestore** operations until it encounters a graphics state that was saved by a **save** operation (as opposed to **gsave**), leaving that state on the top of the graphics state stack and resetting the current graphics state from it. If no such graphics state is encountered (which can happen only during an unencapsulated job), the current graphics state is reset to the bottommost state on the stack and the stack is cleared to empty. If the graphics state stack is empty, **grestoreall** has no effect.

Errors: none
See Also: **gsave, grestore, save, setgstate**

gsave – **gsave** –

pushes a copy of the current graphics state on the graphics state stack (see Section 4.2, "Graphics State"). All elements of the graphics state are saved, including the current transformation matrix, current path, clipping path, and identity of the raster output device, but not the contents of raster memory. The saved state can later be restored by a matching **grestore**. After saving the graphics state, **gsave** resets the clipping path stack in the current graphics state to empty.

The **save** operator also implicitly performs a **gsave** operation, but restoring a graphics state saved by **save** is slightly different from restoring one saved by **gsave**; see the descriptions of **grestore** and **grestoreall**.

Note that, unlike **save**, **gsave** does not return a save object on the operand stack to represent the saved state. **gsave** and **grestore** work strictly in a stacklike fashion, except for the wholesale restoration performed by **restore** and **grestoreall**.

Errors: **limitcheck**
See Also: **grestore, grestoreall, save, restore, gstate, currentgstate, clipsave, cliprestore**

gstate

– **gstate** *gstate*



returns a copy of the current graphics state on the operand stack.

The result is returned as a new gstate (graphics state) object, allocated in either local or global VM according to the current VM allocation mode (see Section 3.7.2, "Local and Global VM"). **gstate** is thus the only graphics state operator that consumes VM.

If *gstate* is allocated in global VM, an **invalidaccess** error will occur if any of the composite objects in the current graphics state are in local VM. Such objects might include the current halftone screen, transfer function, or dash pattern. In general, allocating gstate objects in global VM is risky and should be avoided.

Errors: **invalidaccess, stackoverflow, VMerror**
See Also: **setgstate, currentgstate**

gt

num$_1$ *num*$_2$ **gt** *bool*
string$_1$ *string*$_2$ **gt** *bool*

pops two objects from the operand stack and pushes *true* if the first operand is greater than the second, or *false* otherwise. If both operands are numbers, **gt** compares their mathematical values. If both operands are strings, **gt** compares them element by element, treating the elements as integers in the range 0 to 255, to determine whether the first string is lexically greater than the second. If the operands are of other types or one is a string and the other is a number, a **typecheck** error occurs.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **ge, eq, ne, le, lt**

handleerror *(error)*

is looked up in **errordict** and executed to report error information saved by the default error handlers (see Section 3.11, "Errors"). There is also a procedure named **handleerror** in **systemdict**; it merely calls the procedure in **errordict**.

identmatrix *matrix* **identmatrix** *matrix*

replaces the value of *matrix* with the identity matrix

```
[1 0 0 1 0 0]
```

and pushes the result back on the operand stack. This matrix represents the identity transformation, which leaves all coordinates unchanged.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **matrix, initmatrix, defaultmatrix, setmatrix, currentmatrix**

idiv int_1 int_2 **idiv** *quotient*

divides int_1 by int_2 and returns the integer part of the quotient, with any fractional part discarded. Both operands of **idiv** must be integers and the result is an integer.

Examples

```
 3  2 idiv  ⇒   1
 4  2 idiv  ⇒   2
−5  2 idiv  ⇒  −2
```

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **div, add, mul, sub, mod, cvi**

idtransform *dx′ dy′* **idtransform** *dx dy*
dx′ dy′ matrix **idtransform** *dx dy*

(inverse delta transform) applies the inverse of a transformation matrix to the distance vector (*dx′*, *dy′*), returning the transformed distance vector (*dx*, *dy*). The first form of the operator uses the inverse of the current transformation matrix in the graphics state to transform the distance vector from device space to user space coordinates. The second form applies the inverse of the transformation specified by the *matrix* operand rather than that of the CTM.

A delta transformation is similar to an ordinary transformation (see Section 4.3, "Coordinate Systems and Transformations"), but does not use the translation components t_x and t_y of the transformation matrix. The distance vectors are thus positionless in both the original and target coordinate spaces, making this operator useful for determining how distances map from device space to user space.

Errors: **rangecheck, stackunderflow, typecheck, undefinedresult**
See Also: **dtransform, itransform, invertmatrix**

if *bool proc* **if** –

removes both operands from the stack, then executes *proc* if *bool* is *true*. The **if** operator pushes no results of its own on the operand stack, but *proc* may do so (see Section 3.5, "Execution").

Example

```
3 4 lt {(3 is less than 4)} if   ⇒   (3 is less than 4)
```

Errors: **stackunderflow, typecheck**
See Also: **ifelse**

ifelse *bool proc*$_1$ *proc*$_2$ **ifelse** –

removes all three operands from the stack, then executes *proc*$_1$ if *bool* is *true* or *proc*$_2$ if *bool* is *false*. The **ifelse** operator pushes no results of its own on the operand stack, but the procedure it executes may do so (see Section 3.5, "Execution").

Example

```
4 3 lt
      {(TruePart)}
      {(FalsePart)}
   ifelse              ⇒  (FalsePart)      % Since 4 is not less than 3
```

Errors: **stackunderflow, typecheck**
See Also: **if**

image *width height bits/sample matrix datasrc* **image** –
dict **image** – *(LanguageLevel 2)*

paints a sampled image onto the current page. This description only summarizes the general behavior of the **image** operator; see Section 4.10, "Images," for full details.

The image is a rectangular array of *width* × *height* sample values, each consisting of *bits/sample* bits of data. Valid values of *bits/sample* are 1, 2, 4, 8, or 12. The data is received as a sequence of characters—that is, 8-bit integers in the range 0 to 255. If *bits/sample* is less than 8, sample values are packed from left to right within a character (see Section 4.10.2, "Sample Representation").

The image is considered to exist in its own coordinate system, or *image space*. The rectangular boundary of the image has its lower-left corner at coordinates (0, 0) and its upper-right corner at (*width*, *height*). The *matrix* operand defines a transformation from user space to image space.

In the first form of the operator, the parameters are specified as separate operands. This form always renders a monochrome image according to the **DeviceGray** color space, regardless of the current color space in the graphics state. This is the only form supported in LanguageLevel 1.

In the second form *(LanguageLevel 2)*, the parameters are contained as entries in an image dictionary *dict*, which is supplied as the single operand. This form renders either a monochrome or a color image, according to the current color space. The number of component values per source sample and the interpretation of those values depend on the color space.

In LanguageLevel 1, *datasrc* must be a procedure. In LanguageLevel 2 or 3, it may be any data source—a procedure, a string, or a readable file, including a filtered file (see Section 3.13, "Filtered Files Details").

If *datasrc* is a procedure, it is executed repeatedly to obtain the actual image data. *datasrc* must return a string on the operand stack containing any number of additional characters of sample data. The sample values are assumed to be received in a fixed order: (0, 0) to (*width* – 1, 0), then (0, 1) to (*width* – 1, 1), and so on. If *datasrc* returns a string of length 0, **image** will terminate execution prematurely.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined, undefinedresult**
See Also: **imagemask, colorimage**

imagemask *width height polarity matrix datasrc* **imagemask** –
dict **imagemask** – *(LanguageLevel 2)*

uses a monochrome sampled image as a *stencil mask* of 1-bit samples to control where to apply paint to the current page in the current color (see "Stencil Masking" on page 302.)

In the first form of the operator *(LanguageLevel 1)*, the parameters are specified as separate operands. In the second form *(LanguageLevel 2)*, the parameters are contained as entries in an image dictionary *dict*, which is supplied as the single operand. The behavior of the operator is the same in both forms.

imagemask uses the *width*, *height*, *matrix*, and *datasrc* operands in precisely the same way the **image** operator uses them. The *polarity* operand is a boolean value that determines the polarity of the mask. If *polarity* is *true*, portions of the page corresponding to mask values of 1 are painted in the current color, while those corresponding to mask values of 0 are left unchanged; if *polarity* is *false*, these polarities are reversed. The polarity controls the interpretation of mask samples only; it has no effect on the color of the pixels that are painted.

In the second form of **imagemask**, the polarity is specified by means of the **Decode** entry in the image dictionary. **Decode** values of `[1 0]` and `[0 1]` denote polarity values of *true* and *false*, respectively.

In LanguageLevel 1, *datasrc* must be a procedure. In LanguageLevel 2 or 3, it may be any data source—a procedure, a string, or a readable file, including a filtered file (see Section 3.13, "Filtered Files Details").

imagemask is most useful for painting character glyphs represented as bitmaps. Such bitmaps represent masks through which a color is to be transferred; the bitmaps themselves do not have a color.

Example

```
54 112 translate                    % Locate lower-left corner of square
120 120 scale                       % Scale 1 unit to 120 points
0 0 moveto                          % Define square
0 1 lineto
1 1 lineto
1 0 lineto
closepath
.9 setgray                          % Set current color to gray
fill                                % Fill with gray background
0 setgray                           % Set current color to black
24 23                               % Specify dimensions of source mask
true                                % Set polarity to paint the 1 bits
[24 0 0 –23 0 23]                   % Map unit square to mask
{< 003B00 002700 002480 0E4940      % Define mask data
   114920 14B220 3CB650 75FE88
   17FF8C 175F14 1C07E2 3803C4
   703182 F8EDFC B2BBC2 BB6F84
   31BFC2 18EA3C 0E3E00 07FC00
   03F800 1E1800 1FF800 >}
imagemask
```

This example paints the image shown in Figure 8.7.

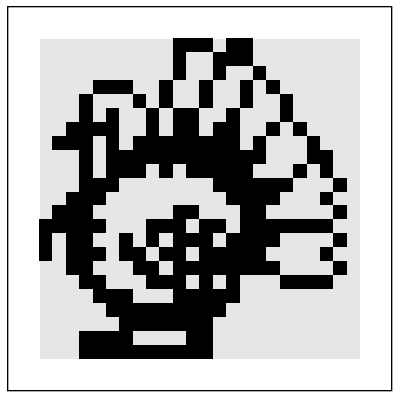


FIGURE 8.7 ***imagemask*** *example*

Errors: **invalidaccess, ioerror, limitcheck, stackunderflow, typecheck, undefinedresult**
See Also: **image, colorimage**

index *any*$_n$ … *any*$_0$ *n* **index** *any*$_n$ … *any*$_0$ *any*$_n$

removes the nonnegative integer *n* from the operand stack, counts down to the *n*th element from the top of the stack, and pushes a copy of that element on the stack.

Examples

```
(a) (b) (c) (d) 0 index   ⇒   (a) (b) (c) (d) (d)
(a) (b) (c) (d) 3 index   ⇒   (a) (b) (c) (d) (a)
```

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **copy, dup, roll**

ineofill *x y* **ineofill** *bool*
userpath **ineofill** *bool*



tests whether the area that would be painted by filling the current path with the **eofill** operator includes a specified point or intersects a specified region. The area is determined according to the even-odd rule (see "Even-Odd Rule" on page 196). In all other respects, the behavior of **ineofill** is identical to that of **infill**.

Note that in the second form of the operator, the area inside *userpath* is determined by the nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195), *not* by the even-odd rule.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **eofill, fill, infill**

infill *x y* **infill** *bool*
userpath **infill** *bool*



tests whether the area that would be painted by filling the current path with the **fill** operator includes a specified point or intersects a specified region. The area inside the current path is determined according to the nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195). The operation does not ac-

tually place any marks on the current page, nor does it disturb the current path in the graphics state.

The first form of the operator returns *true* if the device pixel containing the point at coordinates (*x*, *y*) in user space would be painted by applying the **fill** operator to the current path. The second form returns *true* if any of the pixels enclosed by *userpath* would be painted. If the stated conditions are not met, both forms return *false*.

The operator ignores the current clipping path; that is, it returns *true* for any pixel that lies within the current path, even if the **fill** operator would not actually mark that pixel because it lies outside the clipping path. The following program fragment performs an **infill** test taking the current clipping path into account:

```
gsave
    clippath
    x y infill
grestore
x y infill and
```

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **fill, instroke, ineofill, inufill**

initclip – initclip –

sets the current clipping path in the graphics state to the default clipping path for the current output device. This path usually corresponds to the boundary of the maximum imageable area on the current device. For a page device, its dimensions are those established by the **setpagedevice** operator. For a display device, the clipping region established by **initclip** is not well defined.

There are few situations in which a PostScript program should invoke **initclip** explicitly. A page description that invokes **initclip** usually produces incorrect results if it is embedded within another, composite page.

Errors: none
See Also: **clip, eoclip, rectclip, clippath, initgraphics**

initgraphics – **initgraphics** –

resets the following parameters of the current graphics state to their default values, as follows:

current transformation matrix (CTM)—default for device
current position (current point)—undefined
current path—empty
current clipping path—default for device
current color space—**DeviceGray**
current color—black
current line width—1 user space unit
current line cap—butt end caps
current line join—miter joins
current miter limit—10
current dash pattern—solid, unbroken lines

All other graphics state parameters are left unchanged. These include the current output device, font parameter, stroke adjustment, clipping path stack, and all device-dependent parameters. **initgraphics** affects only the graphics state, not the contents of raster memory or the configuration of the current output device.

initgraphics is equivalent to the following code:

```
initmatrix
newpath
initclip
0 setgray
1 setlinewidth
0 setlinecap
0 setlinejoin
10 setmiterlimit
[ ] 0 setdash
```

There are few situations in which a PostScript program should invoke **initgraphics** explicitly. A page description that invokes **initgraphics** usually produces incorrect results if it is embedded within another, composite page. A program requiring information about its initial graphics state should explicitly read and save that state at the beginning of the program rather than assume that the default state prevailed initially.

Errors: none
See Also: **grestoreall**

initmatrix – **initmatrix** –

sets the current transformation matrix (CTM) in the graphics state to the default matrix for the current output device. This matrix transforms the default user coordinate system to device space (see Section 4.3.1, "User Space and Device Space"). For a page device, the default matrix is initially established by the **setpagedevice** operator.

There are few situations in which a PostScript program should invoke **initmatrix** explicitly. A page description that invokes **initmatrix** usually produces incorrect results if it is embedded within another, composite page.

Errors: none
See Also: **defaultmatrix, setmatrix, currentmatrix**

instroke *x y* **instroke** *bool*
userpath **instroke** *bool*



tests whether the area that would be painted by stroking the current path with the **stroke** operator includes a specified point or intersects a specified region. The operation does not actually place any marks on the current page, nor does it disturb the current path in the graphics state.

The first form of the operator returns *true* if the device pixel containing the point at coordinates (*x*, *y*) in user space would be painted by applying the **stroke** operator to the current path. The second form returns *true* if any of the pixels enclosed by *userpath* would be painted. If the stated conditions are not met, both forms return *false*.

In computing the shape of the stroke, **instroke** takes into account all current stroke-related parameters in the graphics state: line width, line cap, line join, miter limit, dash pattern, and stroke adjustment. (If the current line width is 0, the set of pixels considered to be part of the stroke is device-dependent.) However, the operator ignores the current clipping path; that is, it returns *true* for any pixel that lies within the computed stroke, even if the **stroke** operator would not actually mark that pixel because it lies outside the clipping path.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **stroke, infill, ineofill, inustroke**

internaldict *int* **internaldict** *dict*

pushes the internal dictionary object on the operand stack. The *int* operand *must* be the integer 1183615869. The internal dictionary is in local VM and is writeable. It contains operators and other information whose purpose is internal to the PostScript interpreter. It should be referenced only in special circumstances, such as during construction of Type 1 font programs. (See the book *Adobe Type 1 Font Format* for specific information about constructing Type 1 fonts.) The contents of **internaldict** are undocumented and subject to change at any time.

This operator is not present in some PostScript interpreters.

Errors: **invalidaccess, stackunderflow, undefined**

interrupt *(error)*

processes an external request to interrupt execution of a PostScript program. When the interpreter receives an interrupt request, it executes **interrupt** as if it were an error—in other words, it looks up the name **interrupt** in **errordict**. Execution of **interrupt** is sandwiched between execution of two objects being interpreted in normal sequence.

Unlike most other errors, occurrence of an **interrupt** error does not cause the object being executed to be pushed on the operand stack, nor does it disturb the operand stack in any way.

The precise nature of an external interrupt request depends on the environment in which the PostScript interpreter is running. For example, in some environments, receipt of a Control-C character from a serial communication channel gives rise to the **interrupt** error. This enables a user to explicitly abort a PostScript computation. The default definition of **interrupt** executes a **stop** operation.

inueofill *x y userpath* **inueofill** *bool*
userpath$_1$ *userpath*$_2$ **inueofill** *bool*



interprets a user path definition (see Section 4.6, "User Paths") and tests whether the area that would be painted by filling the resulting path with the **ueofill** operator includes a specified point or intersects a specified region. The area inside the path is determined according to the even-odd rule (see "Even-Odd Rule" on page 196). In all other respects, the behavior of **inueofill** is identical to that if **inufill**.

Note that in the second form of the operator, the area inside *userpath*$_1$ is determined by the nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195), *not* by the even-odd rule.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **eofill, ueofill, inufill, ineofill**

inufill *x y userpath* **inufill** *bool*
userpath$_1$ *userpath*$_2$ **inufill** *bool*



interprets a user path definition (see Section 4.6, "User Paths") and tests whether the area that would be painted by filling the resulting path with the **ufill** operator includes a specified point or intersects a specified region. The area inside the path is determined according to the nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195). The operation does not actually place any marks on the current page, nor does it disturb the current path in the graphics state. Except for the manner in which the path is specified, the behavior of **inufill** is identical to that of **infill**.

The first form of the operator returns *true* if the device pixel containing the point at coordinates (*x*, *y*) in user space would be painted by applying the **ufill** operator to *userpath*. The second form returns *true* if any of the pixels enclosed by *userpath*$_1$ would be painted by applying **ufill** to *userpath*$_2$. If the stated conditions are not met, both forms return *false*.

In itself, **inufill** would seem to be a trivial composition of several other operators:

```
gsave                   % Save graphics state
    newpath             % Clear current path
    uappend             % Interpret userpath
    infill              % Test for inclusion
grestore                % Restore graphics state
```

However, when used with a user path that includes the **ucache** operator, **inufill** can potentially take advantage of cached information to optimize execution.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See also: **fill, ufill, inueofill, infill**

inustroke *x y userpath* **inustroke** *bool*
x y userpath matrix **inustroke** *bool*
userpath$_1$ *userpath*$_2$ **inustroke** *bool*
userpath$_1$ *userpath*$_2$ *matrix* **inustroke** *bool*



interprets a user path definition (see Section 4.6, "User Paths") and tests whether the area that would be painted by stroking the resulting path with the **ustroke** operator includes a specified point or intersects a specified region. The operation does not actually place any marks on the current page, nor does it disturb the current path in the graphics state. Except for the manner in which the path is specified, the behavior of **inustroke** is identical to that of **instroke**.

The first and second forms of the operator return *true* if the device pixel containing the point at coordinates (*x*, *y*) in user space would be painted by applying the **ustroke** operator to *userpath*. The third and fourth forms return *true* if any of the pixels enclosed by *userpath*$_1$ would be painted by applying **ustroke** to *userpath*$_2$. If the stated conditions are not met, all four forms return *false*.

The second and fourth forms concatenate *matrix* to the current transformation matrix after interpreting the user paths, but before computing the area occupied by the stroke (see **ustroke**).

If *userpath* is already present in the user path cache, **inustroke** can take advantage of the cached information to optimize execution.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See also: **stroke, ustroke, instroke**

invalidaccess *(error)*

An access violation has occurred. The principal causes of this error are:

- Accessing the value of a composite object in violation of its access attribute (for example, storing into a read-only array or reading an execute-only string)
- Storing a composite object in local VM as an element of a composite object in global VM
- Executing **pathforall** if the current path contains an outline for a protected font

See Also: **rcheck, wcheck, gcheck, readonly, executeonly, noaccess**

invalidexit *(error)*

An **exit** operator has been executed for which there is no dynamically enclosing looping context (for example, **for**, **loop**, **repeat**, or **pathforall**), or it has attempted to leave the context of a **run** or **stopped** operator.

invalidfileaccess *(error)*

The access string specification to the **file** operator is unacceptable or a file operation has been attempted (for example, **deletefile**) that is not permitted by the storage device. See Section 3.8.2, "Named Files."

invalidfont *(error)*

The *key* operand to **findfont** is not a valid **Font** resource name, the dictionary operand to other font operators is not a well-formed or valid font or CIDFont dictionary, as required, or the **show** operator (or **show** variant) is executed before the font parameter in the graphics state has been set. The **invalidfont** error may also be executed by other font operators upon discovering that a font or CIDFont dictionary has the wrong **FontType** or **CIDFontType** value for the operation or that a glyph description is malformed.

invalidrestore *(error)*

An improper **restore** operation has been attempted. One or more of the operand, dictionary, and execution stacks contain composite objects whose values were created more recently than the **save** operation whose context is being restored. Since **restore** would destroy those values, but the stacks are unaffected by **restore**, the outcome would be undefined and cannot be allowed.

See Also: restore, save

invertmatrix *matrix*$_1$ *matrix*$_2$ **invertmatrix** *matrix*$_2$

replaces the value of *matrix*$_2$ with the inverse of *matrix*$_1$ and pushes the result back on the operand stack. If *matrix*$_1$ transforms coordinates (x, y) to (x', y'), then its inverse transforms (x', y') to (x, y) (see Section 4.3.3, "Matrix Representation and Manipulation").

Errors: **rangecheck, stackunderflow, typecheck, undefinedresult**
See Also: **itransform, idtransform**

ioerror *(error)*

An exception other than end-of-file has occurred during execution of one of the file operators. The nature of the exception is environment-dependent, but may include events such as parity or checksum errors or broken network connections. Attempting to write to an input file or to a file that has been closed will also cause an **ioerror**. Occurrence of an **ioerror** does not cause the file to become closed unless it was already closed or unless the error occurs during **closefile**.

ISOLatin1Encoding – **ISOLatin1Encoding** *array*



pushes the ISO Latin-1 encoding vector on the operand stack. This is a 256-element literal array object, indexed by character codes, whose values are the character names for those codes. **ISOLatin1Encoding** is not an operator; it is a name in **systemdict** associated with the array object.

Latin-text fonts produced by Adobe usually use the **StandardEncoding** encoding vector. However, they contain all the characters needed to support the use of **ISOLatin1Encoding**. A font can have its **Encoding** array changed to **ISOLatin1Encoding** by means of the procedure shown in Section 5.9.1, "Changing the Encoding Vector." The contents of **ISOLatin1Encoding** are documented in Appendix E.

Errors: **stackoverflow**
See Also: **StandardEncoding, findencoding**

itransform *x′ y′* **itransform** *x y*
x′ y′ matrix **itransform** *x y*

(inverse transform) applies the inverse of a transformation matrix to the coordinates (*x′*, *y′*), returning the transformed coordinates (*x*, *y*). The first form of the operator uses the inverse of the current transformation matrix in the graphics state to transform device space coordinates to user space. The second form applies the inverse of the transformation specified by the *matrix* operand rather than that of the CTM.

Errors: **rangecheck, stackunderflow, typecheck, undefinedresult**
See Also: **transform, idtransform, invertmatrix**

known *dict key* **known** *bool*

returns *true* if there is an entry in the dictionary *dict* whose key is *key*; otherwise, it returns *false*. *dict* does not have to be on the dictionary stack.

Examples

```
/mydict 5 dict def
mydict /total 0 put
mydict /total known      ⇒  true
mydict /badname known    ⇒  false
```

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **where, load, get**

kshow *proc string* **kshow** –

paints glyphs for the characters of *string* in a manner similar to **show**, but allows program intervention between characters. If the character codes in *string* are $char_0$, $char_1$, … $char_n$, **kshow** proceeds as follows: First it shows the glyph for $char_0$ at the current point, updating the current point by the width of that glyph. Then it pushes the character codes $char_0$ and $char_1$ on the operand stack (as integers) and executes *proc*. *proc* may perform any actions it wishes; typically, it will modify the current point to affect the subsequent placement of the glyph for $char_1$. **kshow** continues by showing the glyph for $char_1$, pushing $char_1$ and $char_2$ on the stack, executing *proc*, and so on. It finishes by pushing $char_{n-1}$ and $char_n$ on the stack, executing *proc*, and finally showing the glyph for $char_n$.

When *proc* is called for the first time, the graphics state (in particular, the current transformation matrix) is the same as it was at the time **kshow** was invoked, except that the current point has been updated by the width of the glyph for $char_0$. Execution of *proc* is permitted to have any side effects, including changes to the graphics state. Such changes persist from one call of *proc* to the next and may affect graphical output for the remainder of **kshow**'s execution and afterward. When *proc* completes execution, the value of **currentfont** is restored.

The name **kshow** is derived from "kern-show." To *kern* glyphs is to adjust the spacing between adjacent glyphs in order to achieve a visually pleasing result. The **kshow** operator enables user-defined kerning and other manipulations, because arbitrary computations can be performed between pairs of glyphs.

kshow can be applied only to base fonts. If the current font is a composite font or a CIDFont, an **invalidfont** error occurs.

Errors: **invalidaccess, invalidfont, nocurrentpoint, stackunderflow, typecheck**
See Also: **show, ashow, awidthshow, widthshow, xshow, xyshow, yshow, cshow**

languagelevel

– **languagelevel** *int*



returns an integer designating the LanguageLevel supported by the PostScript interpreter. If **languagelevel** is not defined in **systemdict**, the interpreter supports only LanguageLevel 1 features.

Errors: **stackoverflow, undefined**
See Also: **product, revision, serialnumber, version**

le

num_1 num_2 **le** *bool*
$string_1$ $string_2$ **le** *bool*

pops two objects from the operand stack and pushes *true* if the first operand is less than or equal to the second, or *false* otherwise. If both operands are numbers, **le** compares their mathematical values. If both operands are strings, **le** compares them element by element, treating the elements as integers in the range 0 to 255, to determine whether the first string is lexically less than or equal to the second. If the operands are of other types or one is a string and the other is a number, a **typecheck** error occurs.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **lt, eq, ne, ge, gt**

length *array* **length** *int*
packedarray **length** *int*
dict **length** *int*
string **length** *int*
name **length** *int*

returns the number of elements in the value of its operand if the operand is an array, a packed array, or a string. If the operand is a dictionary, **length** returns the current number of entries it contains (as opposed to its maximum capacity, which is returned by **maxlength**). If the operand is a name object, the length returned is the number of characters in the text string that defines it.

Examples

```
[1  2  4] length          ⇒  3
[ ] length                ⇒  0     % An array of zero length

/ar 20 array def
ar length                 ⇒  20

/mydict 5 dict def
mydict length             ⇒  0

mydict /firstkey (firstvalue) put
mydict length             ⇒  1

(abc\n) length            ⇒  4     % Newline (\n) is one character
( ) length                ⇒  0     % No characters between ( and )
/foo length               ⇒  3
```

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **maxlength**

limitcheck *(error)*

An implementation limit has been exceeded (for example, too many files have been opened simultaneously or a path has become too complex). Appendix B gives typical values for such limits.

lineto *x y* **lineto** –

appends a straight line segment to the current path (see Section 4.4, "Path Construction"), starting from the current point and extending to the coordinates (*x*, *y*) in user space. The endpoint (*x*, *y*) becomes the new current point.

If the current point is undefined because the current path is empty, a **nocurrentpoint** error occurs.

Errors: **limitcheck, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **rlineto, moveto, curveto, arc, closepath**

ln *num* **ln** *real*

returns the natural logarithm (base *e*) of *num*. The result is a real number.

Examples

```
10 ln    ⇒  2.30259
100 ln   ⇒  4.60517
```

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **log, exp**

load *key* **load** *value*

searches for *key* in each dictionary on the dictionary stack, starting with the topmost (current) dictionary. If *key* is found in some dictionary, **load** pushes the associated value on the operand stack; otherwise, an **undefined** error occurs.

load looks up *key* the same way the interpreter looks up executable names that it encounters during execution. However, **load** always pushes the associated value on the operand stack; it never executes the value.

Examples

```
/avg {add 2 div} def
/avg load              ⇒  {add 2 div}
```

Errors: **invalidaccess, stackunderflow, typecheck, undefined**
See Also: **where, get, store**

log *num* **log** *real*

returns the common logarithm (base 10) of *num*. The result is a real number.

Examples

```
10 log   ⇒  1.0
100 log  ⇒  2.0
```

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **ln, exp**

loop *proc* **loop** –

repeatedly executes *proc* until *proc* executes the **exit** operator, at which point interpretation resumes at the object next in sequence after the **loop** operator. Control also leaves *proc* if the **stop** operator is executed. If *proc* never executes **exit** or **stop**, an infinite loop results, which can be broken only via an external interrupt (see **interrupt**).

Errors: **stackunderflow, typecheck**
See Also: **for, repeat, forall, exit**

lt num_1 num_2 **lt** *bool*
$string_1$ $string_2$ **lt** *bool*

pops two objects from the operand stack and pushes *true* if the first operand is less than the second, or *false* otherwise. If both operands are numbers, **lt** compares their mathematical values. If both operands are strings, **lt** compares them element by element, treating the elements as integers in the range 0 to 255, to determine whether the first string is lexically less than the second. If the operands are of other types or one is a string and the other is a number, a **typecheck** error occurs.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **le, eq, ne, ge, gt**

makefont *font matrix* **makefont** *font′*
cidfont matrix **makefont** *cidfont′*

applies *matrix* to *font* or *cidfont*, producing a new *font′* or *cidfont′* whose glyphs are transformed by *matrix* when they are shown. **makefont** first creates a copy of *font* or *cidfont*. Then it replaces the copy's **FontMatrix** entry with the result of concatenating the existing entry with *matrix*. It inserts two additional entries, **OrigFont** and **ScaleMatrix**, whose purpose is internal to the implementation. Finally, it returns the result as *font′* or *cidfont′*.

Normally, **makefont** does not copy subsidiary objects in the dictionary, such as the **CharStrings** and **FontInfo** subdictionaries; these are shared with the original font or CIDFont. However, if *font* is a composite font, **makefont** recursively copies any descendant Type 0 font dictionaries and updates their **FontMatrix** entries as well. It does not copy descendant base fonts or CIDFonts.

Showing glyphs from *font′* or *cidfont′* produces the same results as showing from *font* or *cidfont* after having transformed user space by *matrix*. **makefont** is essentially a convenience operator that permits the desired transformation to be encapsulated in the font or CIDFont description. The most common transformation is to scale by a uniform factor in both the *x* and *y* dimensions. **scalefont** is a special case of the more general **makefont** and should be used for such uniform scaling. Another operator, **selectfont**, combines the effects of **findfont** and **makefont**.

The interpreter keeps track of font or CIDFont dictionaries recently created by **makefont**. Calling **makefont** multiple times with the same *font* or *cidfont* and *matrix* will usually return the same result. However, it is usually more efficient for a PostScript program to apply **makefont** only once for each font or CIDFont that it needs and to keep track of the resulting dictionaries on its own.

See Chapter 5 for general information about fonts and CIDFonts, and Section 4.3, "Coordinate Systems and Transformations," for a discussion of transformations.

The derived dictionary is allocated in local or global VM according to whether the original dictionary is in local or global VM. This behavior is independent of the current VM allocation mode.

Example

```
/Helvetica findfont [10 0 0 12 0 0] makefont setfont
```

This example obtains the standard Helvetica font, which is defined with a 1-unit line height, and scales it by a factor of 10 in the *x* dimension and 12 in the *y* dimension. This produces a font 12 units high (that is, a 12-point font in default user space) whose glyphs are "condensed" in the *x* dimension by a ratio of 10/12.

Errors: **invalidfont, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **scalefont, setfont, findfont, selectfont**

makepattern *dict matrix* **makepattern** *pattern*



instantiates the pattern defined by the pattern dictionary *dict*, producing an instance of the pattern locked to the current user space. After verifying that *dict* is a prototype pattern dictionary with all required entries (see Section 4.9, "Patterns"), **makepattern** creates a copy of *dict* in local VM, adding an **Implementation** entry for use by the PostScript interpreter. Only the contents of *dict* itself are copied; any subsidiary composite objects the dictionary contains are not copied, but are shared with the original dictionary.

makepattern saves a copy of the current graphics state, to be used later when the pattern's **PaintProc** procedure is called to render the pattern cell. It then modifies certain parameters in the *saved* graphics state, as follows:

- Concatenates *matrix* with the saved copy of the current transformation matrix
- Adjusts the resulting matrix to ensure that the device space can be tiled properly with a pattern cell of the given size in accordance with the pattern's tiling type
- Resets the current path to empty
- Replaces the clipping path with the pattern cell bounding box specified by the pattern dictionary's **BBox** entry
- Replaces the current device with a special one provided by the PostScript implementation

Finally, **makepattern** makes the new dictionary read-only and returns it on the operand stack. The resulting pattern dictionary is suitable for use as an operand to **setpattern** or as a color value in a **Pattern** color space.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, undefined, VMerror**
See Also: **setpattern**

mark – **mark** *mark*

pushes a mark object on the operand stack. All marks are identical, and the operand stack may contain any number of them at once.

The primary use of marks is to indicate the stack position of the beginning of an indefinitely long list of operands being passed to an operator or procedure. The **]** operator (array construction) is the most common operator that works this way; it treats as operands all elements of the stack down to a mark that was pushed by the **[** operator (**[** is a synonym for **mark**). It is possible to define procedures that

work similarly. Operators such as **counttomark** and **cleartomark** are useful within such procedures.

Errors: **stackoverflow**
See Also: **counttomark, cleartomark, pop**

matrix – **matrix** *matrix*

returns a six-element array object filled with the identity matrix

```
[1 0 0 1 0 0]
```

This matrix represents the identity transformation, which leaves all coordinates unchanged. The array is allocated in local or global VM according to the current VM allocation mode (see Section 3.7.2, "Local and Global VM").

Example

```
matrix
6 array identmatrix
```

Both lines of code above return the same result on the stack.

Errors: **stackoverflow, VMerror**
See Also: **identmatrix, defaultmatrix, setmatrix, currentmatrix, array**

maxlength *dict* **maxlength** *int*

returns the capacity of the dictionary *dict*—in other words, the maximum number of entries that *dict* can hold using the virtual memory currently allocated to it. In LanguageLevel 1, **maxlength** returns the length operand of the **dict** operator that created the dictionary; this is the dictionary's maximum capacity (exceeding it causes a **dictfull** error). In a LanguageLevels 2 and 3, which permit a dictionary to grow beyond its initial capacity, **maxlength** returns its current capacity, a number at least as large as that returned by the **length** operator.

Examples

```
/mydict 5 dict def
mydict length      ⇒  0
mydict maxlength   ⇒  5
```

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **length, dict**

mod *int*$_1$ *int*$_2$ **mod** *remainder*

returns the remainder that results from dividing *int*$_1$ by *int*$_2$. The sign of the result is the same as the sign of the dividend *int*$_1$. Both operands must be integers and the result is an integer.

Examples

```
 5 3 mod  ⇒  2
 5 2 mod  ⇒  1
−5 3 mod  ⇒ −2
```

The last example above demonstrates that **mod** is a *remainder* operation rather than a true *modulo* operation.

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **idiv, div**

moveto *x y* **moveto** –

starts a new subpath of the current path (see Section 4.4, "Path Construction") by setting the current point in the graphics state to the coordinates (*x*, *y*) in user space. No new line segments are added to the current path.

If the previous path operation in the current path was **moveto** or **rmoveto**, that point is deleted from the current path and the new **moveto** point replaces it.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **rmoveto, lineto, curveto, arc, closepath**

mul *num*$_1$ *num*$_2$ **mul** *product*

returns the product of *num*$_1$ and *num*$_2$. If both operands are integers and the result is within integer range, the result is an integer; otherwise, the result is a real number.

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **div, idiv, add, sub, mod**

ne *any*$_1$ *any*$_2$ **ne** *bool*

pops two objects from the operand stack and pushes *false* if they are equal, or *true* if not. What it means for objects to be equal is presented in the description of the **eq** operator.

Errors: **invalidaccess, stackunderflow**
See Also: **eq, ge, gt, le, lt**

neg *num*$_1$ **neg** *num*$_2$

returns the negative of *num*$_1$. The type of the result is the same as the type of *num*$_1$ unless *num*$_1$ is the smallest (most negative) integer, in which case the result is a real number.

Examples

```
 4.5 neg  ⇒  –4.5
–3 neg    ⇒   3
```

Errors: **stackunderflow, typecheck**
See Also: **abs**

newpath – **newpath** –

initializes the current path in the graphics state to an empty path. The current point becomes undefined.

Errors: none
See Also: **closepath, stroke, fill, eofill, currentpoint**

noaccess

array **noaccess** *array*
packedarray **noaccess** *packedarray*
dict **noaccess** *dict*
file **noaccess** *file*
string **noaccess** *string*

reduces the access attribute of an array, packed array, dictionary, file, or string object to none (see Section 3.3.2, "Attributes of Objects"). The value of a no-access object cannot be executed or accessed directly by PostScript operators. No-access

objects are of no use to PostScript programs, but serve certain internal purposes that are not documented in this book.

For an array, packed array, file, or string object, **noaccess** affects the access attribute only of the object that it returns. If there are other objects that share the same value, their access attributes are unaffected. However, in the case of a dictionary, **noaccess** affects the *value* of the object, so all dictionary objects sharing the same dictionary are affected. Applying **noaccess** to a dictionary whose access is already read-only causes an **invalidaccess** error.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **rcheck, wcheck, xcheck, readonly, executeonly**

nocurrentpoint *(error)*

The current path is empty, thus there is no current point, but an operator requiring a current point has been executed (for example, **lineto**, **curveto**, **currentpoint**, or **show**). The most common cause of this error is neglecting to perform an initial **moveto**.

See Also: **moveto**

not *bool*$_1$ **not** *bool*$_2$
int$_1$ **not** *int*$_2$

returns the logical negation of the operand if it is boolean. If the operand is an integer, **not** returns the bitwise complement (ones complement) of its binary representation.

Examples

```
true not   ⇒  false        % A complete truth table
false not  ⇒  true

52 not     ⇒  –53
```

Errors: **stackunderflow, typecheck**
See Also: **and, or, xor, if**

null – **null** *null*

pushes a literal null object on the operand stack. **null** is not an operator; it is a name in **systemdict** associated with the null object.

Errors: **stackoverflow**

nulldevice – **nulldevice** –

installs the null device as the current output device. The null device corresponds to no physical output device and has no raster memory associated with it. When it is the current device, marks placed on the current page by painting operators such as **stroke** or **show** are discarded, and the output operators **showpage** and **copypage** do nothing. In all other respects, the null device behaves like a real raster output device: graphics operators have their normal side effects on the graphics state, font operators invoke the font machinery, and so on.

nulldevice sets the default transformation matrix to be the identity transformation [1.0 0.0 0.0 1.0 0.0 0.0]. If desired, a PostScript program may change this to any other matrix (using **setmatrix**) in order to simulate the device coordinate system of some real device. **nulldevice** also establishes the clipping path as a degenerate path consisting of a single point at the coordinate origin.

The null device is useful for exercising the PostScript interpreter's graphics and font machinery for such purposes as operating on paths, computing bounding boxes for graphical shapes, and performing coordinate transformations without generating output. Such manipulations should be bracketed by **gsave** and **grestore** so that the previous current device can be reinstated and other side effects of **nulldevice** undone.

Errors: none
See Also: **setpagedevice**

or *bool*$_1$ *bool*$_2$ **or** *bool*$_3$
int$_1$ *int*$_2$ **or** *int*$_3$

returns the logical disjunction of the operands if they are boolean. If the operands are integers, **or** returns the bitwise "inclusive or" of their binary representations.

Examples

```
true true or    ⇒ true      % A complete truth table
true false or   ⇒ true
false true or   ⇒ true
false false or  ⇒ false

17 5 or         ⇒ 21
```

Errors: **stackunderflow, typecheck**
See Also: **and, not, xor**

packedarray *any*$_0$ … *any*$_{n-1}$ *n* **packedarray** *packedarray*



creates a packed array object of length *n* containing the objects *any*$_0$ through *any*$_{n-1}$ as elements. **packedarray** first removes the nonnegative integer *n* from the operand stack. It then removes that number of objects from the operand stack, creates a packed array containing those objects as elements, and finally pushes the resulting packed array object on the operand stack.

The resulting object has a type of packedarraytype, a literal attribute, and read-only access. In all other respects, its behavior is identical to that of an ordinary array object.

The packed array is allocated in local or global VM according to the current VM allocation mode. An **invalidaccess** error occurs if the packed array is in global VM and any of the objects *any*$_0$ through *any*$_{n-1}$ are in local VM (see Section 3.7.2, "Local and Global VM").

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **aload**

pathbbox – **pathbbox** *ll*$_x$ *ll*$_y$ *ur*$_x$ *ur*$_y$

returns the bounding box of the current path, the smallest rectangle enclosing all elements of the path. The results are four real numbers describing a rectangle in user space, oriented with its sides parallel to the axes of the user coordinate sys-

tem: ll_x and ll_y are the coordinates of the rectangle's lower-left corner, ur_x and ur_y the upper-right corner. If the current path is empty, a **nocurrentpoint** error occurs.

If an explicit bounding box has been established with the **setbbox** operator, **pathbbox** returns a result derived from that bounding box rather than from the actual path. Otherwise, **pathbbox** first computes the bounding box of the current path in *device* space. It then transforms the corners of this box from device to user coordinates, by applying the inverse of the current transformation matrix, and constructs the smallest rectangle in user space that is oriented parallel to the user space axes and encloses all four corners of the resulting figure. If the user coordinate system is rotated (other than by a multiple of 90 degrees) or skewed, the bounding box returned may be larger than expected.

If the current path includes curve segments, the computed bounding box will enclose the curves' control points as well as the curves themselves. A bounding box fitting the path more tightly can be obtained by first "flattening" the curve segments with the **flattenpath** operator.

In LanguageLevel 2 or 3, if the current path ends with a **moveto** operation, the coordinates of the **moveto** are not considered during the computation of the bounding box unless the **moveto** is the only element of the path.

Errors: **nocurrentpoint, stackoverflow**
See Also: **setbbox, flattenpath**

pathforall *move line curve close* **pathforall** –

enumerates the elements of the current path in order, executing one of the four procedures *move*, *line*, *curve*, or *close* for each element, depending on its nature.

The four basic kinds of element that can occur in a path are **moveto**, **lineto**, **curveto**, and **closepath**. (The relative variants **rmoveto**, **rlineto**, and **rcurveto** are converted to the corresponding absolute forms; **arc**, **arcn**, **arct**, and **arcto** are converted to equivalent sequences of **curveto**.) For each element in the path, **pathforall** pushes the element's coordinates on the operand stack and executes one of the four procedures, as follows:

moveto	Push x y; execute *move*
lineto	Push x y; execute *line*
curveto	Push x_1 y_1 x_2 y_2 x_3 y_3; execute *curve*
closepath	Execute *close*

The operands passed to the procedures are coordinates in user space; **pathforall** transforms them from device space using the inverse of the current transforma-

tion matrix. Ordinarily, these coordinates will be the same as the ones originally specified to **moveto**, **lineto**, and so forth. However, if the CTM has been changed since the path was constructed, the coordinates reported by **pathforall** will be different from those originally specified. Thus, among other uses, **pathforall** enables a path constructed in one user coordinate system to be read out in another user coordinate system.

pathforall enumerates the current path existing at the time it begins execution. If any of the operand procedures change the current path, such changes do not affect the results of the operation.

If **charpath** was used to construct any portion of the current path from a font whose outlines are protected, **pathforall** is not allowed. Its execution will cause an **invalidaccess** error (see **charpath**).

Errors: **invalidaccess, rangecheck, stackoverflow, stackunderflow, typecheck**
See Also: **moveto, lineto, curveto, closepath, charpath**

pop *any* **pop** –

removes the top element from the operand stack and discards it.

Examples

```
1 2 3 pop       ⇒  1 2
1 2 3 pop pop   ⇒  1
```

Errors: **stackunderflow**
See Also: **clear, dup**

print *string* **print** –

writes the characters of *string* to the standard output file (see Section 3.8, "File Input and Output"). This operator provides the simplest means of sending text to an application or an interactive user. Note that **print** is a *file* operator; it has nothing to do with painting glyphs for characters on the current page (see **show**) or with sending the current page to a raster output device (see **showpage**).

Errors: **invalidaccess, ioerror, stackunderflow, typecheck**
See Also: **write, flush, =, ==, printobject**

printobject *obj tag* **printobject** –



writes a binary object sequence to the standard output file (see Section 3.14.6, "Structured Output"). The binary object sequence contains a top-level array consisting of a single element that is an encoding of *obj*. If *obj* is composite, the binary object sequence also includes subsidiary array and string values for the components of *obj*. The *tag* operand, which must be an integer in the range 0 to 255, is used to tag the top-level object; it appears as the second byte of the object's representation. Tag values 0 to 249 are available for general use; tag values 250 to 255 are reserved for special purposes, such as reporting errors.

The binary object sequence uses the number representation established by the most recent execution of **setobjectformat**. The token type given as the first byte of the binary object sequence reflects the number representation that was used. If the object format parameter has been set to 0, an **undefined** error occurs.

The object *obj* and its components must be of type null, integer, real, name, boolean, string, array, or mark (see Section 3.14, "Binary Encoding Details"). Appearance of an object of any other type, including a packed array, results in a **typecheck** error. If arrays are nested too deeply or are cyclical, a **limitcheck** error occurs.

printobject always encodes a name object as a reference to a text name in the string value portion of the binary object sequence, never as a system name index.

As is the case for all operators that write to files, the output produced by **printobject** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, invoking **flush** is required. This is particularly important in situations where the output produced by **printobject** is the response to a query from the application.

Errors: **invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **print, setobjectformat, writeobject**

product – **product** *string*



is a read-only string object that is the name of the product in which the PostScript interpreter is running. The value of this string is typically a manufacturer-defined trademark; it has no direct connection with specific features of the PostScript language.

Errors: **stackoverflow**
See Also: **languagelevel, revision, serialnumber, version**

prompt – **prompt** –

is a procedure executed by **executive** whenever it is ready for the user to enter a new statement. The standard definition of **prompt** is {(PS>) print flush} and is defined in **systemdict**; it can be overridden by defining **prompt** in **userdict** or some other dictionary higher on the dictionary stack. **prompt** is not defined in products that do not support **executive**. See Section 2.4.4, "Using the Interpreter Interactively."

Errors: none
See Also: **executive**

pstack ⊢ any_1 … any_n **pstack** ⊢ any_1 … any_n

writes text representations of every object on the stack to the standard output file, but leaves the stack unchanged. **pstack** applies the **==** operator to each element of the stack, starting with the topmost element. See the **==** operator for a description of its effects.

Errors: none
See Also: **stack**, **=**, **==**

put *array index any* **put** –
dict key any **put** –
string index int **put** –

replaces a single element of the value of the first operand. If the first operand is an array or a string, **put** treats the second operand as an index and stores the third operand at the position identified by the index, counting from 0. *index* must be in the range 0 to $n - 1$, where n is the length of the array or string. If it is outside this range, a **rangecheck** error occurs.

If the first operand is a dictionary, **put** uses the second operand as a key and the third operand as a value, and stores this key-value pair into *dict*. If *key* is already present as a key in *dict*, **put** simply replaces its value by *any*; otherwise, **put** creates a new entry for *key* and associates *any* with it. In LanguageLevel 1, if *dict* is already full, a **dictfull** error occurs.

If the value of *array* or *dict* is in global VM and *any* is a composite object whose value is in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Examples

```
/ar [5 17 3 8] def
ar 2 (abcd) put
ar                  ⇒  [5 17 (abcd) 8]

/d 5 dict def
d /abc 123 put
d {} forall         ⇒  /abc 123

/st (abc) def
st 0 65 put                 % 65 is the ASCII code for the character A
st                  ⇒  (Abc)
```

Errors: **dictfull, invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **get, putinterval**

putinterval

array$_1$ *index array*$_2$ **putinterval** –
array$_1$ *index packedarray*$_2$ **putinterval** –
string$_1$ *index string*$_2$ **putinterval** –

replaces a subsequence of the elements of the first operand by the entire contents of the third operand. The subsequence that is replaced begins at *index* in the first operand; its length is the same as the length of the third operand.

The objects are copied from the third operand to the first, as if by a sequence of individual **get** and **put** operations. In the case of arrays, if the copied elements are themselves composite objects, the values of those objects are shared between *array*$_2$ and *array*$_1$ (see Section 3.3.1, "Simple and Composite Objects").

putinterval requires *index* to be a valid index in *array*$_1$ or *string*$_1$ such that *index* plus the length of *array*$_2$ or *string*$_2$ is not greater than the length of *array*$_1$ or *string*$_1$.

If the value of *array*$_1$ is in global VM and any of the elements copied from *array*$_2$ or *packedarray*$_2$ are composite objects whose values are in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Examples

```
/ar [5 8 2 7 3] def
ar 1 [(a) (b) (c)] putinterval
ar                          ⇒  [5 (a) (b) (c) 3]

/st (abc) def
st 1 (de) putinterval
st                          ⇒  (ade)
```

Errors: **invalidaccess, rangecheck, stackunderflow, typecheck**
See Also: **getinterval, put**

quit – **quit** –

terminates operation of the PostScript interpreter. The precise action of **quit** depends on the environment in which the interpreter is running. It may, for example, give control to an operating system command interpreter, or halt or restart the machine.

In an interpreter that supports multiple execution contexts, the **quit** operator causes termination of the current context only.

In a context that is under the control of a job server (see Section 3.7.7, "Job Execution Environment"), the definition of the **quit** operator in **systemdict** is masked by another definition of **quit** in **userdict**, which usually is searched before **systemdict**. The default definition of **quit** in **userdict** is the same as **stop**, which terminates the current job but not the interpreter as a whole. The **quit** operator in **systemdict** can be executed only by an unencapsulated job; in an encapsulated job, it causes an **invalidaccess** error.

Errors: **invalidaccess**
See Also: **stop, start**

rand – **rand** *int*

returns a random integer in the range 0 to $2^{31} - 1$, produced by a pseudo-random number generator. The random number generator's state can be reset by **srand** and interrogated by **rrand**.

Errors: **stackoverflow**
See Also: **srand, rrand**

rangecheck *(error)*

A numeric operand's value is outside the range expected by an operator—for example, an array or string index is out of bounds, or a negative number appears where a nonnegative number is required. A **rangecheck** error can also occur if an object's length differs from what is expected, such as when a matrix operand does not contain exactly six elements.

rcheck

array **rcheck** *bool*
packedarray **rcheck** *bool*
dict **rcheck** *bool*
file **rcheck** *bool*
string **rcheck** *bool*

tests whether the operand's access permits its value to be read explicitly by PostScript operators. It returns *true* if the operand's access is unlimited or read-only, or *false* otherwise.

Errors: **stackunderflow, typecheck**
See Also: **executeonly, noaccess, readonly, wcheck**

rcurveto $dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$ **rcurveto** –

(relative **curveto**) appends a section of a cubic Bézier curve to the current path in the same manner as **curveto**. However, the operands are interpreted as relative displacements from the current point rather than as absolute coordinates. That is, **rcurveto** constructs a curve between the current point (x_0, y_0) and the endpoint $(x_0 + dx_3, y_0 + dy_3)$, using $(x_0 + dx_1, y_0 + dy_1)$ and $(x_0 + dx_2, y_0 + dy_2)$ as the Bézier control points. In all other respects, the behavior of **rcurveto** is identical to that of **curveto**.

Errors: **limitcheck, nocurrentpoint, stackunderflow, typecheck, undefinedresult**
See Also: **curveto, rlineto, rmoveto**

read *file* **read** *int true (if not end-of-file)*
false (if end-of-file)

reads the next character from the input file *file*, pushes it on the operand stack as an integer, and pushes *true* as an indication of success. If an end-of-file indication is encountered before a character has been read, **read** returns *false*. If some other error indication is encountered (for example, a parity or checksum error), an **ioerror** occurs.

Errors: **invalidaccess, ioerror, stackoverflow, stackunderflow, typecheck**
See Also: **readhexstring, readline, readstring, bytesavailable**

readhexstring *file string* **readhexstring** *substring bool*

reads characters from *file*, expecting to encounter a sequence of hexadecimal digits 0 to 9 and A through F (or a through f). **readhexstring** interprets each successive pair of digits as a two-digit hexadecimal number representing an integer value in the range 0 to 255. It then stores these values into successive elements of *string* starting at index 0 until either the entire string has been filled or an end-of-file indication is encountered in *file*. Finally, **readhexstring** returns the substring of *string* that was filled and a boolean value indicating the outcome (*true* normally, *false* if end-of-file was encountered before the string was filled).

readhexstring ignores any characters that are not valid hexadecimal digits, so the data in *file* may be interspersed with spaces, newlines, and so on, without changing the interpretation of the data.

See Section 3.8.4, "Filters," for more information about ASCII-encoded, binary data representations and how to deal with them.

Errors: **invalidaccess, ioerror, rangecheck, stackunderflow, typecheck**
See Also: **read, readline, readstring, filter**

readline *file string* **readline** *substring bool*

reads a line of characters (terminated by a newline character) from *file* and stores them into successive elements of *string*. **readline** returns the substring of *string* that was filled and a boolean value indicating the outcome (*true* normally, *false* if end-of-file was encountered before a newline character was read).

A line of characters is a sequence of ASCII characters, including space, tab, and control characters, that terminates with a *newline*—a carriage return character, a

line feed character, or both. See Sections 3.2, "Syntax," and 3.8, "File Input and Output."

The terminating newline character is not stored into *string* or included at the end of the returned *substring*. If **readline** completely fills *string* before encountering a newline character, a **rangecheck** error occurs.

Errors: **invalidaccess, ioerror, rangecheck, stackunderflow, typecheck**
See Also: **read, readhexstring, readonly**

readonly

array **readonly** *array*
packedarray **readonly** *packedarray*
dict **readonly** *dict*
file **readonly** *file*
string **readonly** *string*

reduces the access attribute of an array, packed array, dictionary, file, or string object to read-only (see Section 3.3.2, "Attributes of Objects"). Access can only be reduced by this operator, never increased. When an object is read-only, its value cannot be modified by PostScript operators (an **invalidaccess** error will result), but it can still be read by operators or executed by the PostScript interpreter.

For an array, packed array, file, or string object, **readonly** affects the access attribute only of the object that it returns. If there are other objects that share the same value, their access attributes are unaffected. However, in the case of a dictionary, **readonly** affects the *value* of the object, so all dictionary objects sharing the same dictionary are affected.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **executeonly, noaccess, rcheck, wcheck**

readstring *file string* **readstring** *substring bool*

reads characters from *file* and stores them into successive elements of *string* until either the entire string has been filled or an end-of-file indication is encountered in *file*. **readstring** then returns the substring of *string* that was filled and a boolean value indicating the outcome (*true* normally, *false* if end-of-file was encountered before the string was filled). If *string* is zero-length, a **rangecheck** error occurs.

All character codes are treated the same—as integers in the range 0 to 255. There are no special characters (in particular, the newline character is not treated specially). However, the communication channel may usurp certain control characters; see Section 3.8, "File Input and Output."

Errors: **invalidaccess, ioerror, rangecheck, stackunderflow, typecheck**
See Also: **read, readhexstring, readline**

realtime – **realtime** *int*



returns the value of a clock that counts in real time, independently of the execution of the PostScript interpreter. The clock's starting value is arbitrary; it has no defined meaning in terms of calendar time. The unit of time represented by the **realtime** value is one millisecond. However, the rate at which it changes is implementation-dependent. As the time value becomes greater than the largest integer allowed in a particular implementation, it "wraps" to the smallest (most negative) integer.

Errors: **stackoverflow**
See Also: **usertime**

rectclip *x y width height* **rectclip** –
numarray **rectclip** –
numstring **rectclip** –



intersects the area inside the current clipping path with a rectangular path defined by the operands to produce a new, smaller clipping path. In the first form, the operands are four numbers that define a single rectangle. In the other two forms, the operand is an array or an encoded number string that defines an arbitrary number of rectangles (see Sections 3.14.5, "Encoded Number Strings," and 4.6.5, "Rectangles"). After computing the new clipping path, **rectclip** clears the current path with an implicit **newpath** operation.

Assuming *width* and *height* are positive, the first form of the operator is equivalent to the following code:

```
newpath
    x y moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    clip
newpath
```

Note that if the second or third form is used to specify multiple rectangles, the rectangles are treated together as a single path and used for a single **clip** operation. The area inside this combined path is the union of all the rectangular subpaths, because the paths are all drawn in the same direction and the nonzero winding number rule is used (see "Nonzero Winding Number Rule" on page 195).

Errors: **limitcheck, stackunderflow, typecheck**
See Also: **clip, eoclip, clippath, initclip, rectfill, rectstroke**

rectfill *x y width height* **rectfill** –
numarray **rectfill** –
numstring **rectfill** –



paints the area inside a path consisting of one or more rectangles defined by the operands, using the current color. In the first form, the operands are four numbers that define a single rectangle. In the other two forms, the operand is an array or an encoded number string that defines an arbitrary number of rectangles (see Sections 3.14.5, "Encoded Number Strings," and 4.6.5, "Rectangles"). **rectfill** neither reads nor alters the current path in the graphics state.

Assuming *width* and *height* are positive, the first form of the operator is equivalent to the following code:

```
gsave
    newpath
    x y moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    fill
grestore
```

Errors: **limitcheck, stackunderflow, typecheck**
See Also: **fill, rectstroke, rectclip**

rectstroke

x y width height **rectstroke** –
x y width height matrix **rectstroke** –
numarray **rectstroke** –
numarray matrix **rectstroke** –
numstring **rectstroke** –
numstring matrix **rectstroke** –



strokes a path consisting of one or more rectangles defined by the operands. In the first two forms, the operands *x*, *y*, *width*, and *height* are four numbers that define a single rectangle. In the remaining forms, *numarray* or *numstring* is an array or an encoded number string that defines an arbitrary number of rectangles (see Sections 3.14.5, "Encoded Number Strings," and 4.6.5, "Rectangles"). **rectstroke** neither reads nor alters the current path in the graphics state.

The forms of the operator that include a *matrix* operand concatenate it to the current transformation matrix after defining the path, but before stroking it. The resulting matrix affects the line width and dash pattern, if any, but not the path itself.

Assuming *width* and *height* are positive, the first two forms of the operator are equivalent to the following code:

```
gsave
    newpath
    x y moveto
    width 0 rlineto
    0 height rlineto
    width neg 0 rlineto
    closepath
    matrix concat          % Second form only
    stroke
grestore
```

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **stroke, rectfill, rectclip**

removeall *cidfont* **removeall** – (***BitmapFontInit*** *procedure set)*



removes all glyph bitmaps defined for *cidfont*, which must be a Type 4 CIDFont (see "Type 4 CIDFonts" on page 379). The glyphs are removed from the font cache immediately, although they may continue to occupy memory until all pages on which they appear have been produced.

Errors: **invalidfont, stackunderflow, typecheck**
See Also: **addglyph, removeglyphs**

removeglyphs *firstcid lastcid cidfont* **removeglyphs** – (***BitmapFontInit*** *procedure set)*



removes the glyph bitmaps for the characters identified by CID numbers from *firstcid* through *lastcid* in *cidfont*, which must be a Type 4 CIDFont (see "Type 4 CIDFonts" on page 379). The glyphs are removed from the font cache immediately, although they may continue to occupy memory until all pages on which they appear have been produced.

A **rangecheck** error occurs if *lastcid* is less than *firstcid* or if these numbers are outside the valid range of CIDs (see Appendix B). However, no error arises from references to nonexistent glyphs.

Errors: **invalidfont, rangecheck, stackunderflow, typecheck**
See Also: **addglyph, removeall**

renamefile $filename_1$ $filename_2$ **renamefile** –



changes the name of a file from $filename_1$ to $filename_2$. The operands are strings that specify file names on the same storage device (see Section 3.8.2, "Named Files"). If the file named $filename_1$ does not exist, an **undefinedfilename** error occurs. Whether or not an error occurs if a file named $filename_2$ already exists is environment-dependent.

If the device does not allow this operation, an **invalidfileaccess** error occurs. If an environment-dependent error is detected, an **ioerror** occurs.

Errors: **invalidfileaccess, ioerror, stackunderflow, typecheck, undefinedfilename**
See Also: **file, deletefile, status**

repeat *int proc* **repeat** –

executes the procedure *proc int* times, where *int* is a nonnegative integer. This operator removes both operands from the stack before executing *proc* for the first time. If *proc* executes the **exit** operator, **repeat** terminates prematurely. **repeat** leaves no results of its own on the stack, but *proc* may do so.

Examples

```
4 {(abc)} repeat                    ⇒ (abc) (abc) (abc) (abc)
1 2 3 4 3 {pop} repeat              ⇒ 1     % Pops 3 values (down to the 1)
4 {} repeat                         ⇒       % Does nothing four times
mark 0 {(will not happen)} repeat   ⇒ mark
```

In the last example above, a 0 repeat count means that the procedure is not executed at all, thus the mark is still topmost on the stack.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **for, loop, forall, exit**

resetfile *file* **resetfile** –

discards buffered characters belonging to a file object. For an input file, **resetfile** discards any characters that have been received from the source but not yet consumed. For an output file, it discards any characters that have been written to the file but not yet delivered to their destination. **resetfile** never generates an **ioerror**.

This operator may have other side effects that depend on the properties of the underlying file. For example, it may restart communication via a channel that was blocked waiting for buffer space to become available. **resetfile** never waits for characters to be received or transmitted.

Errors: **stackunderflow, typecheck**
See Also: **file, closefile, flushfile**

resourceforall *template proc scratch category* **resourceforall** –



enumerates the names of all instances of a specified resource category or a subset selected by *template*. *category* is a name object that identifies a resource category, such as **Font** (see Section 3.9.2, "Resource Categories"). *template* is a string object to be matched against names of resource instances. For each matching name, **resourceforall** copies the name into the supplied *scratch* string, pushes a string object designating the substring of *scratch* actually used, and calls *proc*. **resourceforall** does not return any results of its own, but *proc* may do so.

template is matched against the names of resource instances, treating them as if they were strings. Within the template, all characters are case-sensitive and are treated literally, with the exception of the following:

* Matches zero or more consecutive characters.

? Matches exactly one character.

\ Causes the next character of the template to be treated literally, even if it is *, ?, or \. Note that \ is treated as an escape character in a string literal. Thus, if *template* is a string literal, \\ must be used to represent \ in the resulting string.

The scratch string is reused during every call to *proc*. If *proc* wishes to save the string that is passed to it, it must make a copy or use the **cvn** operator to convert the string to a name. The use of strings instead of names allows **resourceforall** to function without creating new name objects, which would consume virtual memory needlessly during a large enumeration. It is prudent to provide a scratch string at least as long as the implementation limit for names (see Appendix B).

A resource instance can have a key that is not a name or a string, but such a key matches only the template (*). In this case, **resourceforall** passes the key directly to *proc* instead of copying it into the *scratch* string. This case can arise only for a resource instance defined in virtual memory by a previous **defineresource**; the keys for external resource instances are always names or strings.

Like **resourcestatus**, but unlike **findresource**, **resourceforall** never loads a resource instance into VM.

resourceforall enumerates the resource instances in order of status (the status value returned by **resourcestatus**); that is, it enumerates groups in this order:

1. Instances defined in VM by an explicit **defineresource**; not subject to automatic removal
2. Instances defined in VM by a previous execution of **findresource**; subject to automatic removal
3. Instances not currently defined in VM, but available from external storage

Within each group, the order of enumeration is unpredictable; it is unrelated to order of definition or to whether the definition is local or global. A given resource instance is enumerated only once, even if it exists in more than one group. If *proc* adds or removes resource instances, those instances may or may not appear later in the same enumeration.

Like **resourcestatus**, **resourceforall** considers both local and global definitions if the current VM allocation mode is local, but only global definitions if the current VM allocation mode is global (see **resourcestatus** and **defineresource**).

If the specified resource category does not exist, an **undefined** error occurs. However, no error occurs if there are no instances whose names match the template. Of course, *proc* can generate errors of its own.

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck, undefined**
See Also: **defineresource, undefineresource, findresource, resourcestatus**

resourcestatus *key category* **resourcestatus** *status size true* *(if resource exists)*
false *(if not)*



returns status information about a named resource instance. *category* is a name object that identifies a resource category, such as **Font** (see Section 3.9.2, "Resource Categories"). *key* is a name or string object that identifies the resource instance. (Names and strings are interchangeable; keys of other types are permitted but are not recommended.)

If the named resource instance exists, either defined in virtual memory or available from some external source, **resourcestatus** returns *status*, *size*, and the value *true*; otherwise, it returns *false*. Unlike **findresource**, **resourcestatus** never loads a resource instance into virtual memory.

status is an integer with the following meanings:

0 Defined in VM by an explicit **defineresource**; not subject to automatic removal

1 Defined in VM by a previous execution of **findresource**; subject to automatic removal

2 Not currently defined in VM, but available from external storage

size is an integer giving the estimated VM consumption of the resource instance in bytes. This information may not be available for certain resources; if the size is unknown, −1 is returned. Usually, **resourcestatus** can obtain the size of a status 1 or 2 resource (derived from the %%VMusage: comment in the resource file), but it has no general way to determine the size of a status 0 resource. See Section 3.9.4,

"Resources as Files," for an explanation of how the size is determined. A *size* value of 0 is returned for implicit resources, whose instances do not occupy VM.

If the current VM allocation mode is local, **resourcestatus** considers both local and global resource definitions, in that order (see **defineresource**). However, if the current VM allocation mode is global, only global resource definitions are visible to **resourcestatus**. Resource instances in external storage are visible without regard to the current VM allocation mode.

If the specified resource category does not exist, an **undefined** error occurs.

Errors: **stackoverflow, stackunderflow, typecheck, undefined**
See Also: **defineresource, undefineresource, findresource, resourceforall**

restore *save* **restore** –

resets virtual memory (VM) to the state represented by the supplied save object—in other words, the state at the time the corresponding **save** operator was executed. See Section 3.7, "Memory Management," for a description of VM and the effects of **save** and **restore**.

If the current execution context supports job encapsulation and if *save* represents the outermost saved VM state for this context, then objects in both local and global VM revert to their saved state. If the current context does not support job encapsulation or if *save* is not the outermost saved VM state for this context, then only objects in local VM revert to their saved state; objects in global VM are undisturbed. Job encapsulation is described in Section 3.7.7, "Job Execution Environment."

restore can reset VM to the state represented by any save object that is still valid, not necessarily the one produced by the most recent **save**. After restoring VM, **restore** invalidates its *save* operand along with any other save objects created more recently than that one. That is, a VM snapshot can be used only once; to restore the same environment repeatedly, it is necessary to do a new **save** each time.

restore does not alter the contents of the operand, dictionary, or execution stack, except to pop its *save* operand. If any of these stacks contains composite objects whose values reside in local VM and are newer than the snapshot being restored, an **invalidrestore** error occurs. This restriction applies to save objects and, in LanguageLevel 1, to name objects.

restore does alter the graphics state stack: it performs the equivalent of a **grestoreall** and then removes the graphics state created by **save** from the graphics

state stack. **restore** also resets several per-context parameters to their state at the time of **save**. These include:

- Array packing mode (see **setpacking**)
- VM allocation mode (see **setglobal**)
- Object output format (see **setobjectformat**)
- All user interpreter parameters (see **setuserparams**)

Errors: **invalidrestore, stackunderflow, typecheck**
See Also: **save, grestoreall, vmstatus, startjob**

reversepath – **reversepath** –

replaces the current path with an equivalent one whose segments are defined in the reverse order. The operation reverses the directions and order of segments within each subpath of the current path; the relative order of subpaths within the path as a whole is unspecified and unpredictable.

If a subpath ends with a **closepath** operation, the reversed subpath begins at the point that was the beginning of the original **closepath** segment. The segment added by **closepath** thus remains at the end of the subpath, though it is traversed in the opposite direction.

Errors: **limitcheck**
See Also: **closepath**

revision – **revision** *int*



is an integer designating the current revision level of the product in which the PostScript interpreter is running. Each product has its own numbering system for revisions, independent of those of any other product. This is distinct from the value of **version** in **systemdict**, which is the revision level of the PostScript interpreter, without regard to the product in which it is running.

Errors: **stackoverflow**
See Also: **languagelevel, product, serialnumber, version**

rlineto *dx dy* **rlineto** –

(relative **lineto**) appends a straight line segment to the current path (see Section 4.4, "Path Construction"), starting from the current point and extending *dx* user space units horizontally and *dy* units vertically. That is, the operands *dx* and *dy* are interpreted as relative displacements from the current point rather than as absolute coordinates. In all other respects, the behavior of **rlineto** is identical to that of **lineto**.

If the current point is undefined because the current path is empty, a **nocurrentpoint** error occurs.

Errors: **limitcheck, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **lineto, rmoveto, rcurveto**

rmoveto *dx dy* **rmoveto** –

(relative **moveto**) starts a new subpath of the current path (see Section 4.4, "Path Construction") by displacing the coordinates of the current point *dx* user space units horizontally and *dy* units vertically, without connecting it to the previous current point. That is, the operands *dx* and *dy* are interpreted as relative displacements from the current point rather than as absolute coordinates. In all other respects, the behavior of **rmoveto** is identical to that of **moveto**.

If the current point is undefined because the current path is empty, a **nocurrentpoint** error occurs.

Errors: **limitcheck, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **moveto, rlineto, rcurveto**

roll any_{n-1} … any_0 *n j* **roll** $any_{(j-1) \bmod n}$ … any_0 any_{n-1} … $any_{j \bmod n}$

performs a circular shift of the objects any_{n-1} through any_0 on the operand stack by the amount *j*. Positive *j* indicates upward motion on the stack, whereas negative *j* indicates downward motion.

n must be a nonnegative integer and *j* must be an integer. **roll** first removes these operands from the stack; there must be at least *n* additional elements. It then performs a circular shift of these *n* elements by *j* positions.

If *j* is positive, each shift consists of removing an element from the top of the stack and inserting it between element $n - 1$ and element *n* of the stack, moving all in-

tervening elements one level higher on the stack. If *j* is negative, each shift consists of removing element $n-1$ of the stack and pushing it on the top of the stack, moving all intervening elements one level lower on the stack.

Examples

(a) (b) (c) 3 –1 roll ⇒ (b) (c) (a)
(a) (b) (c) 3 1 roll ⇒ (c) (a) (b)
(a) (b) (c) 3 0 roll ⇒ (a) (b) (c)

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **exch, index, copy, pop**

rootfont – **rootfont** *font*
– **rootfont** *cidfont*



returns the font or CIDFont most recently established by **setfont** or **selectfont**. Normally, **rootfont** returns the same result as **currentfont**. However, when executed inside the **BuildGlyph**, **BuildChar**, or **CharStrings** procedure of a descendant base font or CIDFont, or inside a procedure invoked by **cshow**, **rootfont** returns the root composite font, whereas **currentfont** returns the current descendant base font or CIDFont. (Of course, if the procedure calls **setfont** or **selectfont** first, **rootfont** and **currentfont** both return the newly selected font.)

Errors: **stackoverflow**
See Also: **setfont, selectfont, currentfont**

rotate *angle* **rotate** –
angle matrix **rotate** *matrix*

rotates the axes of the user coordinate space by *angle* degrees counterclockwise about the origin, or returns a matrix representing this transformation. The position of the coordinate origin and the sizes of the coordinate units are unaffected.

The transformation is represented by the matrix

$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the angle specified by the *angle* operand. The first form of the operator applies this transformation to the user coordinate system by concatenating matrix *R* with the current transformation matrix (CTM); that is, it replaces the CTM

with the matrix product $R \times \text{CTM}$. The second form replaces the value of the *matrix* operand with an array representing matrix R and pushes the result back on the operand stack without altering the CTM. See Section 4.3.3, "Matrix Representation and Manipulation," for a discussion of how matrices are represented as arrays.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **setmatrix, currentmatrix, translate, scale, concat**

round *num*$_1$ **round** *num*$_2$

returns the integer value nearest to *num*$_1$. If *num*$_1$ is equally close to its two nearest integers, **round** returns the greater of the two. The type of the result is the same as the type of the operand.

Examples

```
 3.2 round  ⇒   3.0
 6.5 round  ⇒   7.0
–4.8 round  ⇒  –5.0
–6.5 round  ⇒  –6.0
  99 round  ⇒   99
```

Errors: **stackunderflow, typecheck**
See Also: **ceiling, floor, truncate, cvi**

rrand – **rrand** *int*

returns an integer representing the current state of the random number generator used by **rand**. This may later be presented as an operand to **srand** to reset the random number generator to the current position in the sequence of numbers produced.

Errors: **stackoverflow**
See Also: **rand, srand**

run *filename* **run** –

executes the contents of the specified file—in other words, interprets the characters in that file as a PostScript program. When **run** encounters end-of-file or terminates for some other reason (for example, execution of the **stop** operator), it closes the file.

run is essentially a convenience operator for the sequence

```
(r) file cvx exec
```

except for its behavior upon abnormal termination. Also, the context of a **run** operator cannot be left by executing **exit**; an attempt to do so produces the error **invalidexit**. The **run** operator leaves no results on the operand stack, but the program executed by **run** may alter the stacks arbitrarily.

Errors: **ioerror, limitcheck, stackunderflow, typecheck, undefinedfilename**
See Also: **exec, file**

save – **save** *save*

creates a snapshot of the current state of virtual memory (VM) and returns a save object representing that snapshot. The save object is composite and logically belongs to the local VM, regardless of the current VM allocation mode.

Subsequently, the returned save object may be presented to **restore** to reset VM to this snapshot. See Section 3.7, "Memory Management," for a description of VM and of the effects of **save** and **restore**. See the **restore** operator for a detailed description of what is saved in the snapshot.

save also saves the current graphics state by pushing a copy of it on the graphics state stack in a manner similar to **gsave**. This saved graphics state is restored by **restore** and **grestoreall**.

Example

```
/saveobj save def
    … Arbitrary computation …
saveobj restore                    % Restore saved VM state
```

Errors: **limitcheck, stackoverflow**
See Also: **restore, gsave, grestoreall, vmstatus**

scale *s_x s_y* **scale** –
s_x s_y matrix **scale** *matrix*

scales the units of the user coordinate space by a factor of s_x units horizontally and s_y units vertically, or returns a matrix representing this transformation. The position of the coordinate origin and the orientation of the axes are unaffected.

The transformation is represented by the matrix

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The first form of the operator applies this transformation to the user coordinate system by concatenating matrix *S* with the current transformation matrix (CTM); that is, it replaces the CTM with the matrix product $S \times \text{CTM}$. The second form replaces the value of the *matrix* operand with an array representing matrix *S* and pushes the result back on the operand stack without altering the CTM. See Section 4.3.3, "Matrix Representation and Manipulation," for a discussion of how matrices are represented as arrays.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **setmatrix, currentmatrix, translate, rotate, concat**

scalefont *font scale* **scalefont** *font′*
cidfont scale **scalefont** *cidfont′*

applies the scale factor *scale* to *font* or *cidfont*, producing a new *font′* or *cidfont′* whose glyphs are scaled by *scale* (in both the *x* and *y* dimensions) when they are shown. **scalefont** first creates a copy of *font* or *cidfont*. Then it replaces the copy's **FontMatrix** entry with the result of scaling the existing **FontMatrix** by *scale*. It inserts two additional entries, **OrigFont** and **ScaleMatrix**, whose purpose is internal to the implementation. Finally, it returns the result as *font′* or *cidfont′*.

Showing glyphs from *font′* or *cidfont′* produces the same results as showing from *font* or *cidfont* after having scaled user space by *scale* in by means of the **scale** operator. **scalefont** is essentially a convenience operator that enables the desired scale factor to be encapsulated in the font or CIDFont description. Another operator, **makefont**, performs more general transformations than simple scaling. See the description of **makefont** for more information on how the transformed font is derived. **selectfont** combines the effects of **findfont** and **scalefont**.

Example

```
/Helvetica findfont 12 scalefont setfont
```

This example obtains the standard Helvetica font, which is defined with a 1-unit line height, and scales it by a factor of 12 in both the *x* and *y* dimensions. This produces a font 12 units high (that is, a 12-point font in default user space) whose glyphs have the same proportions as those in the original font.

Errors: **invalidfont, stackunderflow, typecheck, undefined**
See Also: **makefont, setfont, findfont, selectfont**

scheck

any **scheck** *bool*



has the same behavior as **gcheck**. This operator is defined for compatibility with earlier PostScript interpreter implementations.

Errors: **stackunderflow**
See Also: **gcheck**

search

string seek **search** *post match pre true* *(if found)*
string false *(if not found)*

looks for the first occurrence of the string *seek* within *string* and returns the results of this search on the operand stack. The topmost result is a boolean value that indicates whether the search succeeded.

If **search** finds a subsequence of *string* whose elements are equal to the elements of *seek*, it splits *string* into three segments: *pre*, the portion of *string* preceding the match; *match*, the portion of *string* that matches *seek*; and *post*, the remainder of *string*. It then pushes the string objects *post*, *match*, and *pre* on the operand stack, followed by the boolean value *true*. All three of these strings are substrings sharing intervals of the value of the original *string*.

If **search** does not find a match, it pushes the original *string* followed by *false*.

Examples

```
(abbc) (ab) search    ⇒  (bc) (ab) () true
(abbc) (bb) search    ⇒  (c) (bb) (a) true
(abbc) (bc) search    ⇒  () (bc) (ab) true
(abbc) (B) search     ⇒  (abbc) false
```

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **anchorsearch, token**

selectfont *key scale* **selectfont** –
key matrix **selectfont** –



obtains a **Font** resource instance whose name is *key*, transforms the instance (which may be a font or CIDFont dictionary) according to *scale* or *matrix*, and establishes it as the font parameter in the graphics state. **selectfont** is equivalent to one of the following, according to whether the second operand is a number or a matrix:

key findfont *scale* scalefont setfont
key findfont *matrix* makefont setfont

If the **Font** resource instance named by *key* is already defined in virtual memory, **selectfont** obtains the corresponding dictionary directly and does not execute **findfont**. However, if the **Font** resource instance is not defined, **selectfont** invokes **findfont**. In the latter case, it actually executes the name object findfont, so it uses the current definition of that name in the environment of the dictionary stack. On the other hand, redefining **scalefont**, **makefont**, or **setfont** would not alter the behavior of **selectfont**.

selectfont can give rise to any of the errors possible for the component operations, including arbitrary errors from a user-defined **findfont** procedure.

Example

```
/Helvetica 10 selectfont                    % More efficient
/Helvetica findfont 10 scalefont setfont
```

Both lines of code above have the same effect, but the first one is almost always more efficient.

Errors: **invalidfont, rangecheck, stackunderflow, typecheck**
See Also: **findfont, makefont, scalefont, setfont**

serialnumber – **serialnumber** *int*



returns an integer that purports to represent the specific machine on which the PostScript interpreter is running. The precise significance of this number (including any claim of its uniqueness) is product-dependent.

Errors: **stackoverflow**
See Also: **languagelevel, product, revision, version**

serverdict – **serverdict** *dict*

pushes a job server dictionary on the operand stack. **serverdict** is not an operator; it is a name in **systemdict** associated with the dictionary object. The only documented entry in this dictionary is **exitserver**; see Section 3.7.7, "Job Execution Environment."

Errors: **stackoverflow**
See Also: **exitserver**

setbbox llx lly urx ury **setbbox** –



establishes a bounding box for the current path, within which the coordinates of all path construction operators must fall. Any subsequent attempt to append a path element with a coordinate lying outside the bounding box will cause a **rangecheck** error; subsequent invocations of **pathbbox** will return a result derived from the bounding box rather than from the actual path. The bounding box remains in effect for the lifetime of the current path—that is, until the next **newpath** or any operator that resets the path implicitly.

The operands define a rectangle in user space, oriented with its sides parallel to the axes of the user coordinate system: llx and lly are the coordinates of the rectangle's lower-left corner, urx and ury the upper-right corner. The upper-right coordinate values must be greater than or equal to the lower-left values, or a **rangecheck** error will occur. **setbbox** transforms the corners of the specified bounding box from user to device coordinates, then constructs the smallest rectangle in device space that is oriented parallel to the device space axes and encloses all four corners. All subsequent bounding box checking is done in device space.

Note that arcs constructed with the **arc**, **arcn**, **arct**, and **arcto** operators are converted to equivalent sequences of **curveto** operations. The coordinates computed as control points for these curves must also fall within the bounding box. This

means that the figure of the arc must be entirely enclosed by the bounding box. On the other hand, the bounding box only constrains the path itself, not the results of rendering it. For example, stroking the path may place marks outside the bounding box without causing an error.

Although the **setbbox** operator can be used for any path, its main use is in defining user paths, where it is mandatory (see Section 4.6, "User Paths"). Any user path procedure passed to one of the user path rendering operators (such as **ufill**) must begin with a **setbbox** operation (optionally preceded by **ucache**). The bounding box information enables the user path rendering operator to optimize execution.

If **setbbox** is invoked more than once during the definition of a path, the path's effective bounding box is successively enlarged to enclose the union of all of the individual bounding boxes specified. Such multiple invocation is not permitted within a user path definition, but could conceivably arise in building up a single current path by concatenating several user paths with multiple invocations of **uappend**.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **pathbbox**

setblackgeneration

proc **setblackgeneration** –



sets the black-generation function in the graphics state to *proc*. The value of this parameter is a procedure that computes the value of the black color component during the conversion of color values from the **DeviceRGB** color space to **DeviceCMYK** (see Section 7.2.3, "Conversion from DeviceRGB to DeviceCMYK"). The procedure is called with a number in the range 0.0 to 1.0 on the operand stack and must return a number in the same range.

Because the effect of the black-generation function is device-dependent, **setblackgeneration** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentblackgeneration, setundercolorremoval**

setcachedevice $w_x\ w_y\ ll_x\ ll_y\ ur_x\ ur_y$ **setcachedevice** –

passes width and bounding box information to the PostScript interpreter's font machinery. **setcachedevice** can be executed only within the context of a **BuildGlyph**, **BuildChar**, or **CharStrings** procedure for a font or CIDFont. The procedure must invoke **setcachedevice**, **setcachedevice2**, or **setcharwidth** *before* executing graphics operators to define and paint the glyph. **setcachedevice** requests the font machinery to transfer the results of those operators both into the font cache, if possible, and onto the current page.

The operands to **setcachedevice** are all numbers interpreted in the glyph coordinate system (see Section 5.4, "Glyph Metric Information"). w_x and w_y define the basic width vector for this glyph—in other words, the normal position of the origin of the next glyph relative to origin of this one.

ll_x and ll_y are the coordinates of the lower-left corner, and ur_x and ur_y the upper-right corner, of the glyph bounding box. The glyph bounding box is the smallest rectangle, oriented with the glyph coordinate system axes, that completely encloses all marks placed on the page as a result of executing the glyph's description. For a glyph defined as a path, this may be determined by means of the **pathbbox** operator. The font machinery needs this information to make decisions about clipping and caching. The declared bounding box must be correct—in other words, sufficiently large to enclose the entire glyph. If any marks fall outside this bounding box, the result is unpredictable.

setcachedevice installs identical sets of metrics for writing modes 0 and 1, while **setcachedevice2** installs separate metrics.

After execution of **setcachedevice** and until the termination of the **BuildGlyph**, **BuildChar**, or **CharStrings** procedure, invocation of color-setting operators or the **image** operator is not allowed; see Section 4.8, "Color Spaces." Note that use of the **imagemask** operator is permitted.

Errors: **stackunderflow, typecheck, undefined**
See Also: **setcachedevice2, setcharwidth, setcachelimit, cachestatus**

setcachedevice2 $w0_x\ w0_y\ ll_x\ ll_y\ ur_x\ ur_y\ w1_x\ w1_y\ v_x\ v_y$ **setcachedevice2** –



passes two sets of glyph metrics to the font machinery (see Section 5.4, "Glyph Metric Information"). $w0_x$ and $w0_y$ are the distances from the current point to the new current point when showing text in writing mode 0. ll_x, ll_y and ur_x, ur_y are the distances from origin 0 to the lower-left and upper-right corners of the glyph bounding box. $w1_x$ and $w1_y$ are the distances from the current point to the new

current point when showing text in writing mode 1. v_x and v_y are the distances from origin 0 to origin 1.

Aside from its interpretation of the operands, **setcachedevice2** works the same as **setcachedevice** in all respects.

Errors: **stackunderflow, typecheck, undefined**
See Also: **setcachedevice, setcharwidth, setcachelimit, cachestatus**

setcachelimit *int* **setcachelimit** –

establishes the maximum number of bytes the pixel array (bitmap) of a single cached glyph may occupy. Any glyph larger than this (according to the glyph bounding box information passed to **setcachedevice**) is not saved in the font cache. Instead, its description is executed every time the glyph is encountered.

setcachelimit affects the decision whether to place new glyphs in the font cache; it does not disturb any glyphs already in the cache. Making the limit larger allows larger glyphs to be cached, but may decrease the total number of different glyphs that can be held in the cache simultaneously. Changing this parameter is appropriate only in very unusual situations.

The maximum limit for *int* is implementation-dependent, representing the total available size of the font cache (see **cachestatus**). As a practical matter, *int* should not be larger than a small fraction of the total font cache size.

Modifications to the cache limit parameter are subject to **save** and **restore**. In an interpreter that supports multiple contexts, this parameter is maintained separately for each context.

The parameter set by **setcachelimit** is the same as the **MaxFontItem** user parameter set by **setuserparams** (see Appendix C).

Errors: **stackunderflow, typecheck**
See Also: **cachestatus, setuserparams**

setcacheparams *mark size lower upper* **setcacheparams** –



sets font cache parameters as specified by the integer objects above the topmost mark on the stack, then removes all operands and the mark object as if by **cleartomark**.

The number of cache parameters is variable. If more operands are supplied to **setcacheparams** than are needed, the topmost ones are used and the remainder ignored. If fewer are supplied than are needed, **setcacheparams** implicitly inserts default values between the mark and the first supplied operand.

The *size*, *lower*, and *upper* parameters set by **setcacheparams** are the same as the **MaxFontCache** system parameter and the **MinFontCompress** and **MaxFontItem** user parameters, respectively (see Appendix C). If a specified value lies outside the range achievable by the implementation, the nearest achievable value is substituted with no error indication.

Changing the font cache size is allowed only in a system administrator job, since it is equivalent to changing a system parameter. If *size* is not specified, the font cache size is unchanged.

Errors: **invalidaccess, typecheck, unmatchedmark**
See Also: **currentcacheparams, setcachelimit, setsystemparams, setuserparams**

setcharwidth w_x w_y **setcharwidth** –

is similar to **setcachedevice**, but it passes only width information to the PostScript interpreter's font machinery and it declares that the glyph being defined is not to be placed in the font cache.

setcharwidth is useful in the unusual case of defining glyphs that incorporate two or more specific opaque colors, such as opaque black and opaque white. Most glyphs have no inherent color, but are painted with the current color within the glyph's outline, leaving the area outside unpainted (transparent).

Another use of **setcharwidth** is in defining glyphs that intentionally change their behavior based on the environment in which they execute. Such glyphs must not be cached, because that would subvert the intended variable behavior.

Errors: **stackunderflow, typecheck, undefined**
See Also: **setcachedevice, setcachedevice2**

setcmykcolor *cyan magenta yellow black* **setcmykcolor** –



sets the current color space in the graphics state to **DeviceCMYK** and the current color to the component values specified by *cyan*, *magenta*, *yellow*, and *black*. Each component must be a number in the range 0.0 to 1.0. If any of the operands is outside this range, the nearest valid value is substituted without error indication.

Color values set by **setcmykcolor** are not affected by black-generation and undercolor-removal computations (see Section 7.2.3, "Conversion from DeviceRGB to DeviceCMYK").

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentrgbcolor, setcolorspace, setcolor, setgray, setrgbcolor, sethsbcolor**

setcolor *comp*$_1$ … *comp*$_n$ **setcolor** –
pattern **setcolor** –
comp$_1$ … *comp*$_n$ *pattern* **setcolor** –



sets the current color in the graphics state.

The appropriate form of the operator depends on the current color space. All color spaces except **Pattern** use the first form, in which the operands *comp*$_1$ through *comp*$_n$ specify the values of the color components describing the desired color. The number of components and the valid ranges of their values depend on the specific characteristics of the color space; see Section 4.8, "Color Spaces." (In the case of an **Indexed** color space, the single operand *comp*$_1$ is actually an index into the space's color table rather than a true color component.) If the wrong number of components is specified, an error will occur, such as **stackunderflow** or **typecheck**. If a component value is outside the valid range, the nearest valid value will be substituted without error indication.

The second and third forms of **setcolor** are used when the current color space is a **Pattern** space. In both forms, the *pattern* operand is a pattern dictionary describing the pattern to be established as the current color. The values of the dictionary's **PatternType** and **PaintType** entries determine whether additional operands are needed:

- Shading patterns (**PatternType** 2) or colored tiling patterns (**PatternType** 1, **PaintType** 1) use the second form of the operator, in which the pattern dictionary is the only operand.

- Uncolored tiling patterns (**PatternType** 1, **PaintType** 2) use the third form, in which the dictionary is accompanied by one or more component values in the pattern's underlying color space, defining the color in which the pattern is to be painted.

The **setcolorspace** operator initializes the current color to a value that depends on the specific color space selected.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentcolor, setcolorspace, setgray, setrgbcolor, sethsbcolor, setcmykcolor**

setcolorrendering

dict **setcolorrendering** –



sets the current CIE-based color rendering dictionary (CRD) in the graphics state to *dict*. The default CRD is device-dependent (see Section 7.1, "CIE-Based Color to Device Color").

Because the effect of the CRD is device-dependent, this operator ordinarily should not be used in a page description that is intended to be device-independent. However, it is acceptable to use it to establish a CRD that has been obtained by means of the **findcolorrendering** operator; this does not compromise the device independence of the page description, even though the CRD itself is device-dependent.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **currentcolorrendering, findcolorrendering**

setcolorscreen

redfreq redang redproc
greenfreq greenang greenproc
bluefreq blueang blueproc
grayfreq grayang grayproc **setcolorscreen** –

redfreq redang redhalftone
greenfreq greenang greenhalftone
bluefreq blueang bluehalftone
grayfreq grayang grayhalftone **setcolorscreen** –



sets the halftone screen parameter in the graphics state (see Section 7.4, "Halftones") as specified by the operands. **setcolorscreen** sets independent halftone screens for the four primary color components of the output device (red, green, blue, and gray) or their complements (cyan, magenta, yellow, and black); this distinguishes it from **setscreen**, which sets the screen identically for all four primary components.

In the first form of the operator, the operands define a separate frequency, angle, and spot function for each component, which are interpreted the same as in the **setscreen** operator. The second form substitutes halftone dictionaries in place of the spot functions. This form sets all the halftone screens for all four components identically, using the *grayfreq*, *grayang*, and *grayhalftone* operands in the same manner as **setscreen**; the first nine operands (*redfreq* through *bluehalftone*) are ignored.

In LanguageLevel 3, the behavior of **setcolorscreen** can be altered by the user parameters **AccurateScreens** (see "Type 1 Halftone Dictionaries" on page 487), **HalftoneMode** ("Halftone Setting" on page 757), and **MaxSuperScreen** (Section 7.4.8, "Supercells").

Because the effect of the halftone screen is device-dependent, **setcolorscreen** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Example

```
/sfreq  50  def                    % 50 halftone cells per inch
/sproc {    dup mul                % Dot-screen spot function
            exch dup mul
            add 1
            exch sub
        } def
```

```
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
```

This example establishes 50-line dot screens angled at 75 degrees for cyan, 15 degrees for magenta, 0 degrees for yellow, and 45 degrees for black, which are commonly used for color printing.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **currentcolorscreen, setscreen, sethalftone**

setcolorspace

array **setcolorspace** –
name **setcolorspace** –



sets the current color space in the graphics state. It also initializes the current color to a value that depends on the specific color space selected. The initial value of the current color space is **DeviceGray**.

In the first form of the operator, the color space is specified by an array of the form

[*family* $param_1$ … $param_n$]

where *family* is the name of the color space family and the parameters $param_1$ through $param_n$ further describe the space within that family. The number and meanings of these parameters vary depending on the family; see Section 4.8, "Color Spaces," for details.

In the second form, the color space is specified by its family name only. This is allowed only for those color space families that require no parameters: **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, and **Pattern**. Specifying a color space by name is equivalent to specifying it by a one-element array containing just that name with no other parameters.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **rangecheck, stackunderflow, typecheck, undefined**
See Also: **currentcolorspace, setcolor**

setcolortransfer *redproc greenproc blueproc grayproc* **setcolortransfer** –



sets the transfer function parameter in the graphics state. **setcolortransfer** specifies transfer functions for all four primary color components of the output device (red, green, blue, and gray) or their complements (cyan, magenta, yellow, and black). Each operand must be a PostScript procedure that is called with a number in the range 0.0 to 1.0 on the operand stack and will return a number in the same range.

These procedures adjust the values of device color components (see Section 7.3, "Transfer Functions"). A device's output will be affected only by those transfer functions corresponding to color components that the device supports. 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. On a device whose **ProcessColorModel** is **DeviceN**, none of the transfer functions set by **setcolortransfer** have any effect; in this case, all components' transfer functions must be specified via **TransferFunction** entries in a halftone dictionary supplied to **sethalftone**.

Because the effect of the transfer function parameter is device-dependent, **setcolortransfer** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentcolortransfer, settransfer**

setdash *array offset* **setdash** –

sets the dash pattern parameter in the graphics state. This parameter controls the lines to be drawn by subsequent invocations of **stroke** and related operators, such as **rectstroke** and **ustroke**. An empty (zero-length) *array* operand denotes solid, unbroken lines. If *array* is not empty, its elements (which must be nonnegative numbers and not all zero) define the sequence of dashes and gaps constituting the dash pattern.

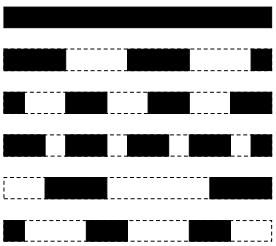
The elements of *array* alternately specify the length of a dash and the length of a gap between dashes, expressed in units of the user coordinate system. The **stroke** operator uses these elements cyclically; when it reaches the end of the array, it starts again at the beginning.

Dashed strokes wrap around curves and corners in the same way as solid strokes. The ends of each dash are treated with the current line cap, and corners within a dash are treated with the current line join. **stroke** takes no measures to coordinate

the dash pattern with features of the path itself; it simply dispenses dashes along the path in the pattern defined by *array*.

The *offset* operand can be thought of as the "phase" of the dash pattern relative to the start of the path. It is interpreted as a distance into the dash pattern (measured in user space units) at which to start the pattern. Before beginning to stroke a path, the **stroke** operator cycles through the elements of *array*, adding up distances and alternating dashes and gaps as usual, but without generating any output. When the accumulated distance reaches the value specified by *offset*, it begins stroking from the starting point of the path, using the dash pattern from the point that has been reached. Each subpath of a path is treated independently; the dash pattern is restarted and the offset reapplied at the beginning of each subpath.

Examples



```
[] 0 setdash          % Solid, unbroken lines
[3] 0 setdash         % 3 units on, 3 units off, …
[2] 1 setdash         % 1 on, 2 off, 2 on, 2 off, …
[2 1] 0 setdash       % 2 on, 1 off, 2 on, 1 off, …
[3 5] 6 setdash       % 2 off, 3 on, 5 off, 3 on, 5 off, …
[2 3] 11 setdash      % 1 on, 3 off, 2 on, 3 off, 2 on, …
```

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **currentdash, stroke**

setdevparams *string dict* **setdevparams** –



attempts to set one or more parameters for the device identified by *string* according to keys and new values contained in the dictionary *dict*. *string* identifies a named parameter set, which usually but not always corresponds to an input/output or other device (see Section C.4, "Device Parameters"). The dictionary is merely a container for key-value pairs; **setdevparams** reads the information from the dictionary but does not retain the dictionary itself. Device parameters whose keys are not mentioned in the dictionary are left unchanged.

Each parameter is identified by a key, which is always a name object. The value is usually (but not necessarily) an integer. String values should consist of nonnull characters; if a null character is present, it will terminate the string. The names of parameter sets and the names and semantics of the parameters are product-dependent. They are not documented in this book, but rather in the *PostScript Language Reference Supplement* and in product-specific documentation.

Permission to alter device parameters is controlled by a password. The dictionary may need to contain an entry named **Password** whose value is a string or integer equal to the system parameter password (see Section C.1.2, "System Parameters"). If the password is incorrect, an **invalidaccess** error occurs and **setdevparams** does not alter any parameters.

Some device parameters can be set permanently in nonvolatile storage that survives restarts of the PostScript interpreter. This capability is implementation-dependent. No error occurs if parameters cannot be stored permanently. For more details on device parameters, see Appendix C.

Various errors are possible. Details of error behavior are product-dependent, but the following behavior is typical:

- If a parameter name is not known to the implementation, an **undefined** error occurs.
- If a parameter value is of the wrong type, a **typecheck** error occurs.
- If a numeric parameter value is unreasonable—for instance, a negative integer for a parameter that must be positive—a **rangecheck** error occurs.
- If a numeric parameter value is reasonable but cannot be achieved by the implementation, either the nearest achievable value is substituted or a **configurationerror** occurs, depending on the device and the parameter.
- If a string parameter value exceeds either the general implementation limit on strings (noted in Appendix B) or an implementation-dependent limit specific to that parameter, a **limitcheck** error occurs.

Errors: **configurationerror, invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **currentdevparams, setsystemparams, setuserparams**

setfileposition

file position **setfileposition** –



repositions an existing open file to a new position so that the next read or write operation will commence at that position. The *position* operand is a nonnegative integer interpreted as number of bytes from the beginning of the file. For an output file, **setfileposition** first performs an implicit **flushfile** operation (see Section 3.8, "File Input and Output").

The result of positioning beyond end-of-file for both reading and writing depends on the behavior of the underlying file system. Typically, positioning beyond the existing end-of-file will lengthen the file if it is open for writing and the file's access permits this. The storage appended to the file has unspecified contents. If

lengthening the file is not permitted, an **ioerror** occurs. Possible causes of an **ioerror** are that the file object is not valid, the underlying file is not positionable, the specified position is invalid for the file, or a device-dependent error condition is detected.

Errors: **ioerror, rangecheck, stackunderflow, typecheck, undefinedfilename**
See Also: **fileposition, file**

setflat *num* **setflat** –

sets the flatness parameter in the graphics state to *num*, which must be a positive number. This parameter controls the precision with which curved path segments are rendered on the raster output device by operators such as **stroke**, **fill**, and **clip**. These operators render curves by approximating them with a series of straight line segments. *Flatness* is the error tolerance of this approximation; it is the maximum allowable distance of any point of the approximation from the corresponding point on the true curve, measured in output device pixels. The acceptable range of values is 0.2 to 100.0. If *num* is outside this range, the nearest valid value is substituted without error indication.

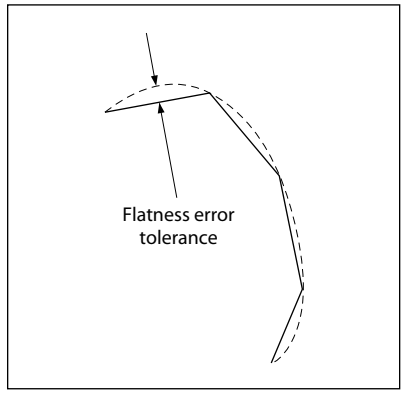


FIGURE 8.8 ***setflat*** *operator*

Figure 8.8 is exaggerated for emphasis. If the flatness parameter is large enough to cause visible straight line segments to appear, the result is unpredictable. The purpose of **setflat** is to control the precision of curve rendering, *not* to draw inscribed polygons.

The choice of a flatness value is a tradeoff between precision and execution efficiency. Very small values (less than 1 device pixel) produce very precise curves at high cost, because enormous numbers of tiny line segments must be generated. Larger values produce cruder approximations with substantially less computation. A default value of the flatness parameter is established by the device setup (**Install**) procedure for each raster output device. This value is based on the characteristics of the individual device and is suitable for most applications.

setflat sets a graphics state parameter whose effect is device-dependent. It should not be used in a page description that is intended to be device-independent.

Errors: **stackunderflow, typecheck**
See Also: **currentflat, flattenpath, stroke, fill, clip**

setfont

font **setfont** –
cidfont **setfont** –

establishes *font* or *cidfont* as the font parameter in the graphics state (subsequently returned by **rootfont**). This in turn determines the current font or CIDFont (returned by **currentfont**)—the font to be used by subsequent glyph operators, such as **show** and **stringwidth**, or the CIDFont to be used by a subsequent **glyphshow** operator. The operand must be a valid font or CIDFont dictionary. See Section 5.1, "Organization and Use of Fonts."

Example

```
/Helvetica findfont        % Obtain prototype Helvetica font
10 scalefont               % Scale it to 10-unit size
setfont                    % Establish it as font parameter
```

Errors: **invalidfont, stackunderflow, typecheck**
See Also: **currentfont, rootfont, scalefont, makefont, findfont, selectfont**

setglobal

bool **setglobal** –



sets the VM allocation mode: *true* denotes global, *false* denotes local. This controls the VM region in which the values of new composite objects are to be allocated (see Section 3.7, "Memory Management"). It applies to objects created implicitly by the scanner and to those created explicitly by PostScript operators.

Modifications to the VM allocation mode are subject to **save** and **restore**. In an interpreter that supports multiple execution contexts, the VM allocation mode is maintained separately for each context.

The standard error handlers in **errordict** execute false setglobal, reverting to local VM allocation mode if an error occurs.

Errors: **stackunderflow, typecheck**
See Also: **currentglobal**

setgray

num **setgray** –

sets the current color space in the graphics state to **DeviceGray** and the current color to the gray level specified by *num*. The gray level must be a number in the range 0.0 to 1.0, with 0.0 denoting black and 1.0 denoting white. If *num* is outside this range, the nearest valid value is substituted without error indication.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentgray, setcolorspace, setcolor, setrgbcolor, sethsbcolor, setcmykcolor**

setgstate

gstate **setgstate** –



replaces the current graphics state with the value of a gstate (graphics state) object (see Section 4.2, "Graphics State"). The contents of *gstate* are *copied* to the graphics state, so subsequent modifications to one will not affect the other. Note that this operation replaces *all* components of the graphics state; in particular, the current clipping path is replaced by the value in *gstate*, not intersected with it.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **gstate, currentgstate, gsave, grestore**

sethalftone

halftone **sethalftone** –



sets the halftone screen parameter in the graphics state (see Section 7.4, "Halftones") as specified by a halftone dictionary. This distinguishes it from **setscreen** and **setcolorscreen**, which specify the halftone's properties by passing individual frequency, angle, and spot function operands directly on the stack.

halftone must be a halftone dictionary constructed as described in Section 7.4.3, "Halftone Dictionaries." If the dictionary's **HalftoneType** value is out of bounds

or is not supported by the PostScript interpreter, a **rangecheck** error occurs; if a required entry is missing, an **undefined** error occurs; if an entry's value is of the wrong type, a **typecheck** error occurs. Once established as the current halftone, the dictionary should be treated as read-only.

In LanguageLevel 3, the behavior of **sethalftone** can be altered by the user parameters **HalftoneMode** (see "Halftone Setting" on page 757) and **MaxSuperScreen** (Section 7.4.8, "Supercells").

Because the effect of the halftone screen is device-dependent, **sethalftone** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **currenthalftone, setscreen, setcolorscreen**

sethsbcolor

hue saturation brightness **sethsbcolor** –

sets the current color space in the graphics state to **DeviceRGB** and the current color to the color described by the parameters *hue*, *saturation*, and *brightness*. Each parameter must be a number in the range 0.0 to 1.0. If any of the operands is outside this range, the nearest value is substituted without error indication.

Note that the HSB parameter values supplied to **sethsbcolor** are immediately converted into RGB color components. HSB is not a color space in its own right, but merely an alternate way of specifying color values in the **DeviceRGB** color space.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currenthsbcolor, setcolorspace, setcolor, setgray, setrgbcolor, setcmykcolor**

setlinecap *int* **setlinecap** –

sets the line cap parameter in the graphics state to *int*, which must be 0, 1, or 2. This parameter controls the shape to be painted at the ends of open subpaths (and dashes, if any) by subsequent invocations of **stroke** and related operators, such as **ustroke** (see Section 4.5.1, "Stroking"). Possible values are as follows (see Figure 8.9):

0 *Butt cap*. The stroke is squared off at the endpoint of the path. There is no projection beyond the end of the path.

1 *Round cap*. A semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.

2 *Projecting square cap*. The stroke continues beyond the endpoint of the path for a distance equal to half the line width and is then squared off.

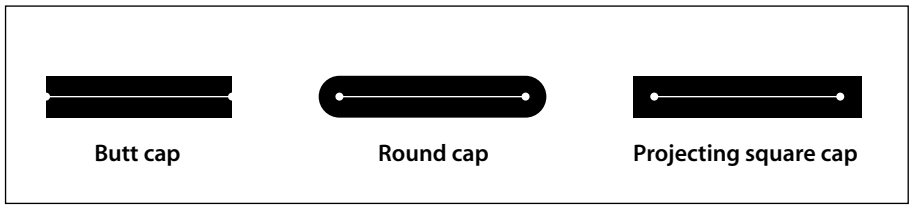


FIGURE 8.9 *Line cap parameter shapes*

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **currentlinecap, stroke, setlinejoin**

setlinejoin *int* **setlinejoin** –

sets the line join parameter in the graphics state to *int*, which must be 0, 1, or 2. This parameter controls the shape to be painted at corners by subsequent invocations of **stroke** and related operators, such as **rectstroke** and **ustroke** (see Section 4.5.1, "Stroking"). Possible values are as follows (see Figure 8.10):

0 *Miter join*. The outer edges of the strokes for the two segments are extended until they meet at an angle, as in a picture frame. If the segments meet at too sharp an angle (as defined by the miter limit parameter—see **setmiterlimit**), a bevel join is used instead.

1 *Round join*. A circular arc with a diameter equal to the line width is drawn around the point where the two segments meet and is filled in, producing a rounded corner. **stroke** draws a full circle at this point; if path segments shorter than half the line width meet at sharp angles, an unintended "wrong side" of this circle may appear.

2 *Bevel join*. The two segments are finished with butt caps (see **setlinecap**), and the resulting notch beyond the ends of the segments is filled with a triangle.

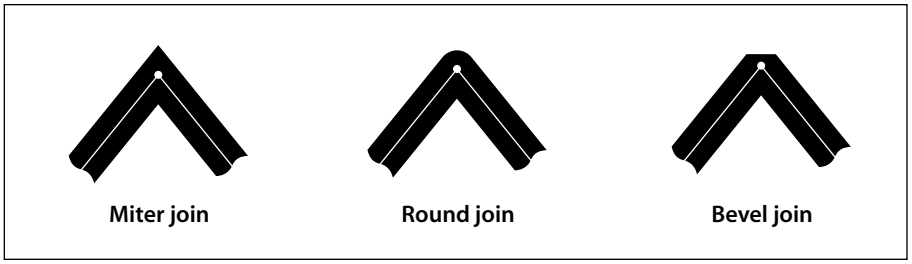


FIGURE 8.10 *Line join parameter shapes*

Join styles are significant only at points where consecutive segments of a path connect at an angle. Segments that meet or intersect fortuitously receive no special treatment. Curved segments are actually rendered as sequences of straight line segments, and the current line join is applied to the "corners" between these segments. However, for typical values of the flatness parameter (see **setflat**), the corners are so shallow that the difference between join styles is not visible.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **currentlinejoin, stroke, setlinecap, setmiterlimit**

setlinewidth

num **setlinewidth** –

sets the line width parameter in the graphics state to *num*. This parameter controls the thickness of lines to be drawn by subsequent invocations of **stroke** and related operators, such as **rectstroke** and **ustroke**. When stroking a path, **stroke** paints all points whose perpendicular distance from the path in user space is less than or equal to half the absolute value of *num*. The effect produced in device space depends on the current transformation matrix (CTM) in effect at the time the path is stroked. If the CTM specifies scaling by different factors in the *x* and *y* dimen-

sions, the thickness of stroked lines in device space will vary according to their orientation.

A line width of 0 is acceptable, and is interpreted as the thinnest line that can be rendered at device resolution—1 device pixel wide. However, some devices cannot reproduce 1-pixel lines, and on high-resolution devices, they are nearly invisible. Since the results of rendering such "zero-width" lines are device-dependent, their use is not recommended.

The actual line width achieved by **stroke** can differ from the requested width by as much as 2 device pixels, depending on the positions of lines with respect to the pixel grid. Automatic stroke adjustment (see **setstrokeadjust**) can be used to ensure uniform line width.

Errors: **stackunderflow, typecheck**
See Also: **currentlinewidth, stroke, setstrokeadjust**

setmatrix

matrix **setmatrix** –

sets the current transformation matrix (CTM) in the graphics state to *matrix* without reference to the former CTM. Except in device setup procedures, the use of this operator should be very rare. PostScript programs should ordinarily *modify* the CTM with the **translate**, **scale**, **rotate**, and **concat** operators rather than replace it.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **currentmatrix, initmatrix, translate, scale, rotate, concat**

setmiterlimit

num **setmiterlimit** –

sets the miter limit parameter in the graphics state to *num*, which must be a number greater than or equal to 1. This parameter controls the treatment of corners by **stroke** and related operators, such as **rectstroke** and **ustroke** (see Section 4.5.1, "Stroking"), when miter joins have been specified by **setlinejoin**. When path segments connect at a sharp angle, a miter join will result in a spike that extends well beyond the connection point. The purpose of the miter limit is to cut off such spikes when they become objectionably long.

At any given corner, the *miter length* is the distance from the point at which the inner edges of the strokes intersect to the point at which their outer edges intersect (see Figure 8.11). This distance increases as the angle between the segments

decreases. If the ratio of the miter length to the line width exceeds the specified miter limit, the **stroke** operator treats the corner with a bevel join instead of a miter join.

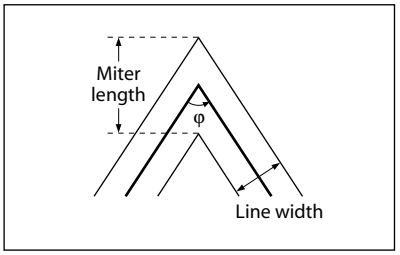


FIGURE 8.11 *Miter length*

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

$$\frac{miterLength}{lineWidth} = \frac{1}{\sin\left(\frac{\varphi}{2}\right)}$$

Example miter limit values are:

- 1.414 cuts off miters (converts them to bevels) at angles less than 90 degrees.
- 2.0 cuts off miters at angles less than 60 degrees.
- 10.0 cuts off miters at angles less than 11 degrees.
- 1.0 cuts off miters at all angles, so that bevels are always produced even when miters are specified.

The default value of the miter limit is 10.0.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **currentmiterlimit, stroke, setlinejoin**

setobjectformat *int* **setobjectformat** –



establishes the number representation to be used in binary object sequences written by subsequent execution of **printobject** and **writeobject**. Output produced by those operators will have a token type that identifies the representation used. The *int* operand is one of the following (see Section 3.14, "Binary Encoding Details"):

0 Disable binary encodings (see below)
1 High-order byte first, IEEE standard real format
2 Low-order byte first, IEEE standard real format
3 High-order byte first, native real format
4 Low-order byte first, native real format

Note that the latter four values specify the number representation only for output. Incoming binary encoded numbers use a representation that is specified as part of each token (in the initial token type byte).

The value 0 disables all binary encodings for both input and output. That is, the PostScript language scanner treats all incoming characters as part of the ASCII encoding, even if a token starts with a character code in the range 128 to 159. The **printobject** and **writeobject** operators are disabled; executing them will cause an **undefined** error. This mode is provided for compatibility with certain existing PostScript programs.

The initial value of this parameter is implementation-dependent. A program must invoke **setobjectformat** to generate output with a predictable number representation.

Modifications to the object format parameter are subject to **save** and **restore**. In an interpreter that supports multiple contexts, this parameter is maintained separately for each context.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **currentobjectformat, printobject, writeobject**

setoverprint *bool* **setoverprint** –



sets the overprint parameter in the graphics state to *bool*. On output devices capable of producing separations or of generating composite output in multiple colorants, this parameter controls whether painting in one separation or colorant causes the corresponding areas of other separations or colorants to be erased (*false*) or left unchanged (*true*); see Section 4.8.5, "Overprint Control." The default value is *false*.

When overprint is *false*, the color marked at any position on the page is whatever was painted there last; this is consistent with the normal opaque painting behavior of the Adobe imaging model. When overprint is *true*, the color at a given position may be a combined result of several painting operations in different colorants.

Because the effect of the overprint parameter is device-dependent, **setoverprint** should not be used in a program that is intended to be device-independent.

Errors: **stackunderflow, typecheck**
See Also: **currentoverprint, setcolorspace**

setpacking

bool **setpacking** –



sets the array packing mode to *bool*. This determines the type of executable arrays subsequently created by the PostScript language scanner. The value *true* selects packed arrays; *false* selects ordinary arrays.

The packing mode affects only the creation of procedures by the scanner when it encounters program text bracketed by { and } during interpretation of an executable file or string object, or during execution of the **token** operator. It does not affect the creation of literal arrays by the **[** and **]** operators or by the **array** operator.

Modifications to the array packing mode parameter are subject to **save** and **restore**. In an interpreter that supports multiple contexts, this parameter is maintained separately for each context.

Example

```
systemdict /setpacking known
        {   /savepacking currentpacking def
            true setpacking
        }
    if

… Arbitrary procedure definitions …

systemdict /setpacking known
        {savepacking setpacking}
    if
```

This example illustrates how to use packed arrays in a way that is compatible with all LanguageLevels. If the packed array facility is available, the procedures represented by the arbitrary procedure definitions are defined as packed arrays; otherwise, they are defined as ordinary arrays. The example is careful to preserve the array packing mode in effect before its execution.

Errors: **stackunderflow, typecheck**
See Also: **currentpacking, packedarray**

setpagedevice *dict* **setpagedevice** –



modifies the contents of the page device dictionary in the graphics state based on the contents of the dictionary operand. The operand is a *request dictionary* containing requested new values for one or more page device parameters. If valid for the current page device, these requested values are merged by **setpagedevice** into the current page device dictionary. The interpretation of these parameters is described in Section 6.2, "Page Device Parameters."

The results of **setpagedevice** are cumulative. The request dictionary for any given invocation is not required to include any particular keys; parameter values established in previous invocations will persist unless explicitly overridden. This cumulative behavior applies not only to the top-level dictionary, but also recursively to the subdictionaries **InputAttributes**, **OutputAttributes**, and **Policies**, as well as to some types of details dictionaries.

The result of executing **setpagedevice** is to instantiate a page device dictionary, perform the equivalent of **initgraphics** and **erasepage**, and install the new device dictionary as an implicit part of the graphics state. The effects of **setpagedevice** are subject to **save** and **restore**, **gsave** and **grestore**, and **setgstate**.

setpagedevice can be used by system administrators to establish a default state for a device by invoking it as part of an unencapsulated job (see Section 3.7.7, "Job Execution Environment"). This default state persists until the next restart of the PostScript interpreter. Some PostScript implementations store some of the device parameters in persistent storage when **setpagedevice** is executed as part of an unencapsulated job, making those parameters persist through interpreter restart.

setpagedevice reinitializes everything in the graphics state except the font parameter, including parameters not affected by **initgraphics**. Device-dependent rendering parameters, such as the halftone screen, transfer functions, flatness tolerance, and color rendering dictionary, are reset to built-in default values or to ones provided in the **Install** procedure of the page device dictionary.

When the current device in the graphics state is not a page device—for example, after **nulldevice** has been invoked or when an interactive display device is active—**setpagedevice** creates a new device dictionary from scratch before merging in the parameters from *dict*. The contents of this dictionary are implementation-dependent.

If a device's **BeginPage** or **EndPage** procedure invokes **setpagedevice**, an **undefined** error occurs.

Errors: **configurationerror, invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck, undefined, VMerror**
See Also: **currentpagedevice, nulldevice, gsave, grestore**

setpattern *pattern* **setpattern** –
comp$_1$ … *comp*$_n$ *pattern* **setpattern** –



sets the current color space in the graphics state to **Pattern** and establishes a specified pattern as the current color. Subsequent painting operations (except **image** and **colorimage**) will use this pattern to paint the required areas of the current page.

setpattern is a convenience operator for patterns that performs the equivalent of **setcolorspace** and **setcolor** in a single operation. It is equivalent to the following code:

```
currentcolorspace 0 get /Pattern ne
    {   [/Pattern currentcolorspace] setcolorspace
    } if
setcolor
```

Normally, **setpattern** establishes a **Pattern** color space whose underlying color space is the one in effect at the time **setpattern** is invoked. However, if the current color space is already a **Pattern** space, **setpattern** simply leaves it unchanged.

setpattern then invokes **setcolor** to establish the specified pattern as the current color in the graphics state, passing it the same operand values that were supplied to **setpattern** itself. The *pattern* operand is a pattern dictionary constructed as specified in Section 4.9, "Patterns," and instantiated by **makepattern**. The need for the remaining operands depends on the value of the pattern dictionary's **PatternType** and **PaintType** entries:

- Shading patterns (**PatternType** 2) and colored tiling patterns (**PatternType** 1, **PaintType** 1) define their own colors as part of the pattern itself; there is no underlying color space. Therefore, the operands *comp*$_1$ through *comp*$_n$ should not be specified.
- Uncolored tiling patterns (**PatternType** 1, **PaintType** 2) have no inherent color; the color must be specified explicitly by the components *comp*$_1$ through *comp*$_n$ in the underlying color space of the **Pattern** space. If the **Pattern** color space does not have an underlying color space, a **rangecheck** error occurs.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **rangecheck, stackunderflow, typecheck, undefined**
See Also: **makepattern, setcolor, setcolorspace**

setrgbcolor *red green blue* **setrgbcolor** –

sets the current color space in the graphics state to **DeviceRGB** and the current color to the component values specified by *red*, *green*, and *blue*. Each component must be a number in the range 0.0 to 1.0. If any of the operands is outside this range, the nearest valid value is substituted without error indication.

Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentrgbcolor, setcolorspace, setcolor, setgray, sethsbcolor, setcmykcolor**

setscreen *frequency angle proc* **setscreen** –
frequency angle halftone **setscreen** – *(LanguageLevel 2)*

sets the halftone screen parameter in the graphics state (see Section 7.4, "Halftones") as specified by the operands. **setscreen** sets the screen identically for all four primary color components of the output device (red, green, blue, and gray); this distinguishes it from **setcolorscreen**, which sets the four screens independently.

frequency is a number specifying the screen frequency, measured in halftone cells per inch in device space. *angle* specifies the number of degrees by which the cells are rotated counterclockwise with respect to the device coordinate system. (Note, however, that most output devices have left-handed device spaces; on such devices, a counterclockwise angle in device space will correspond to a clockwise angle in default user space and on the physical medium.)

In the first form of the operator, the *proc* operand is a PostScript procedure defining the *spot function*, which determines the order in which pixels within a halftone cell are whitened to produce any desired shade of gray. In the second form, *halftone* is a halftone dictionary defining the desired screen; in this case, **setscreen** performs the equivalent of **sethalftone**, except that if the dictionary is of type 1, the values of the *frequency* and *angle* operands are copied into the dictionary's

Frequency and **Angle** entries, overriding the original values of those entries. (If the dictionary is read-only, **setscreen** makes a copy of it before copying the values.) For halftone dictionaries of types other than 1, the *frequency* and *angle* operands are ignored.

A **rangecheck** error occurs if *proc* returns a result outside the range −1.0 to 1.0. A **limitcheck** error occurs if the size of the screen cell exceeds implementation limits.

In LanguageLevel 3, the behavior of **setscreen** can be altered by the user parameters **AccurateScreens** (see "Type 1 Halftone Dictionaries" on page 487), **HalftoneMode** ("Halftone Setting" on page 757), and **MaxSuperScreen** (Section 7.4.8, "Supercells").

Because the effect of the halftone screen is device-dependent, **setscreen** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **currentscreen, setcolorscreen, sethalftone**

setshared

bool **setshared** –



has the same behavior as **setglobal**. This operator is defined for compatibility with earlier PostScript interpreter implementations.

Errors: **stackunderflow, typecheck**
See Also: **setglobal**

setsmoothness

num **setsmoothness** –



sets the smoothness parameter in the graphics state to *num*, which must be a number from 0.0 to 1.0. This parameter controls the precision of smoothly shaded output, and thus indirectly affects rendering performance. *Smoothness* is the allowable color error, or tolerance, between a shading approximated by piecewise linear interpolation and the true value of a (possibly nonlinear) shading function. The error is measured for each color component, and the maximum error is used. The tolerance is expressed as a fraction of the range of the color component, from 0.0 to 1.0. Thus, a smoothness parameter of 0.1 represents a tolerance of 10 percent in each color component. If the value of *num* is outside the range 0.0 to 1.0, the nearest valid value is substituted without error indication.

Each output device may have internal limits on the maximum and minimum tolerances attainable. For example, setting smoothness to 1.0 may result in an effective smoothness of 0.5 on a high-quality color device, while setting it to 0.0 on the same device may yield an effective smoothness of 0.01.

The choice of a smoothness value is a tradeoff between precision and execution efficiency. Very small values produce very precise color shadings at high cost, because enormous numbers of tiny colored areas must be generated. Larger values produce cruder approximations with substantially less computation. A default value of the smoothness parameter is established by the device setup (**Install**) procedure for each raster output device. This value is based on the characteristics of the individual device and is suitable for most applications.

The smoothness parameter may also interact with the accuracy of color conversion. In the case of a color conversion defined by a PostScript procedure or table, the conversion function is unknown; if the color error is sampled at too low a frequency, 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.

setsmoothness sets a graphics state parameter whose effect is device-dependent. It should not be used in a page description that is intended to be device-independent.

Errors: **rangecheck, stackoverflow**
See Also: **currentsmoothness**

setstrokeadjust

bool **setstrokeadjust** –



sets the stroke adjustment parameter in the graphics state to *bool*. This parameter controls whether automatic stroke adjustment will be performed during subsequent invocations of **stroke** and related operators, including **strokepath** (see Section 7.5.2, "Automatic Stroke Adjustment").

The initial value of the stroke adjustment parameter is device-dependent; typically it is *true* for displays and *false* for printers. It is set to *false* when a font's **BuildChar**, **BuildGlyph**, or **CharStrings** procedure is called, but the procedure can change it. It is not altered by **initgraphics**.

Errors: **stackunderflow, typecheck**
See Also: **currentstrokeadjust, stroke, setlinewidth**

setsystemparams *dict* **setsystemparams** –



attempts to set one or more system parameters whose keys and new values are contained in the dictionary *dict*. The dictionary is merely a container for key-value pairs; **setsystemparams** reads the information from the dictionary but does not retain the dictionary itself. System parameters whose keys are not mentioned in the dictionary are left unchanged.

Each parameter is identified by a key, which is always a name object. If the named system parameter does not exist in the implementation, it is ignored. If a specified numeric value is not achievable by the implementation, the nearest achievable value is substituted without error indication.

String values should consist of nonnull characters; if a null character is present, it will terminate the string. String-valued parameters may be subject not only to the general implementation limit on strings (noted in Appendix B), but also to implementation-dependent limits specific to certain parameters. If either limit is exceeded, a **limitcheck** error occurs.

The names of system parameters and details of their semantics are given in Appendix C. Additional parameters are described in the *PostScript Language Reference Supplement* and in product-specific documentation. Some user parameters have default values that can be specified as system parameters with the same names.

Permission to alter system parameters is controlled by a password. The dictionary usually must contain an entry named **Password** whose value is a string or integer equal to the system parameter password (see Section C.1.2, "System Parameters"). If the password is incorrect, an **invalidaccess** error occurs and **setsystemparams** does not alter any parameters.

Some system parameters can be set permanently in nonvolatile storage that survives restarts of the PostScript interpreter. This capability is implementation-dependent. No error occurs if parameters cannot be stored permanently.

Example

```
<<    /MaxFontCache 500000
      /MaxFontItem 7500
      /Password (xxxx)
>> setsystemparams
```

This example attempts to set the **MaxFontCache** system parameter to 500,000 and the default value of the **MaxFontItem** user parameter to 7500.

Errors: **invalidaccess, limitcheck, stackunderflow, typecheck**
See Also: **currentsystemparams, setuserparams, setdevparams**

settransfer *proc* **settransfer** –

sets the transfer function parameter in the graphics state to *proc* (see Section 7.3, "Transfer Functions"). A transfer function is a procedure that adjusts the value of a color component to compensate for nonlinear response in an output device and in the human eye. The procedure is called with a number in the range 0.0 to 1.0 on the operand stack and must return a number in the same range. **settransfer** sets the transfer function identically for all four primary color components of the output device (red, green, blue, and gray); this distinguishes it from **setcolortransfer**, which sets the four transfer functions independently.

Because the effect of the transfer function is device-dependent **settransfer** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck**
See Also: **currenttransfer, setcolortransfer**

settrapparams *dict* **settrapparams** – *(**Trapping** procedure set)*



sets or updates the contents of the current trapping parameter dictionary (see Section 6.3.3, "Trapping Parameters").

dict is a dictionary with the same structure as the trapping parameter dictionary and usually contains a subset of the possible trapping parameters. **settrapparams** merges the contents of this dictionary with those of the existing trapping parameter dictionary, replacing or adding entries as appropriate. Unrecognized entries in *dict* are ignored. Changes to the current trapping parameter dictionary do not affect trapping zones already defined.

The effects of calls to **settrapparams** are cumulative. The value assigned to a given trapping parameter persists through subsequent invocations of **settrapparams** until explicitly overridden or until the contents of the trapping parameter dictionary are restored to some previous state by a **restore** operation. A PostScript program can thus specify individual trapping options independently by invoking **settrapparams** separately for each, setting a particular trapping parameter or set of parameters while leaving the values of other parameters unaffected.

This cumulative behavior does not apply to the contents of the **ColorantZoneDetails** dictionary. A program wishing to change the value of just one entry in the **ColorantZoneDetails** dictionary must retrieve the dictionary's current contents and explicitly merge them with the new value before calling **settrapparams**. This enables the program to remove entries from the **ColorantZoneDetails** dictionary.

Note that this differs from the cumulative behavior of the **setpagedevice** operator for **ColorantDetails**.

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **currenttrapparams, settrapzone**

settrapzone – **settrapzone** – *(**Trapping** procedure set)*



sets a trapping zone (see Section 6.3.2, "Trapping Zones") whose area is the region inside the current path, as determined by the normal PostScript nonzero winding number rule (see "Nonzero Winding Number Rule" on page 195). The new zone's trapping parameters are defined by the current contents of the trapping parameter dictionary. Subsequent changes to the current path or the trapping parameter dictionary will not affect this trapping zone.

Like the **fill** and **stroke** operators, **settrapzone** implicitly performs a **newpath** operation after defining a trapping zone.

Errors: **limitcheck**
See Also: **currenttrapparams, settrapparams**

setucacheparams *mark blimit* **setucacheparams** –



sets user path cache parameters as specified by the integer objects above the topmost mark on the stack, then removes all operands and the mark object itself, as if by **cleartomark**.

The number of cache parameters is variable. If more operands are supplied than are needed, the topmost ones are used and the remainder ignored. If too few are supplied, **setucacheparams** implicitly inserts default values between the mark and the first supplied operand.

blimit specifies the maximum number of bytes that can be occupied by the reduced representation of a single path in the user path cache. Any reduced path larger than this limit is not saved in the cache. Changing *blimit* does not disturb any paths that are already in the cache. A *blimit* value that is too large is automatically reduced to the maximum permissible value without error indication.

Modifications to the cache limit parameter are subject to **save** and **restore**. In an interpreter that supports multiple contexts, this parameter is maintained separately for each context.

The parameter that **setucacheparams** sets is the same as the **MaxUPathItem** user parameter set by **setuserparams** (see Appendix C).

Errors: **rangecheck, typecheck, unmatchedmark**
See Also: **ucachestatus, setuserparams**

setundercolorremoval

proc **setundercolorremoval** –



sets the undercolor-removal function in the graphics state to *proc*. The value of this parameter is a procedure that computes the amount by which to reduce the cyan, magenta, and yellow color components in order to compensate for black generation during the conversion of color values from the **DeviceRGB** color space to **DeviceCMYK** (see Section 7.2.3, "Conversion from DeviceRGB to DeviceCMYK"). The procedure is called with a number in the range 0.0 to 1.0 on the operand stack and must return a number in the range –1.0 to +1.0. Negative result values increase the levels of the cyan, magenta, and yellow color components; positive values decrease them.

Because the effect of the undercolor-removal function is device-dependent, **setundercolorremoval** should not be used in a page description that is intended to be device-independent. Execution of this operator is not permitted in certain circumstances; see Section 4.8.1, "Types of Color Space."

Errors: **stackunderflow, typecheck, undefined**
See Also: **currentundercolorremoval, setblackgeneration**

setuserparams

dict **setuserparams** –



attempts to set one or more user parameters whose keys and new values are contained in the dictionary *dict*. The dictionary is merely a container for key-value pairs; **setuserparams** reads the information from the dictionary but does not retain the dictionary itself. User parameters whose keys are not mentioned in the dictionary are left unchanged.

Each parameter is identified by a key, which is always a name object. If the named user parameter does not exist in the implementation, it is ignored. If a specified numeric value is not achievable by the implementation, the nearest achievable value is substituted without error indication.

String values should consist of nonnull characters; if a null character is present, it will terminate the string. String-valued parameters may be subject not only to the general implementation limit on strings (noted in Appendix B) but also to

implementation-dependent limits specific to certain parameters. If either limit is exceeded, a **limitcheck** error occurs.

The names of user parameters and details of their semantics are given in Appendix C. Additional parameters are described in the *PostScript Language Reference Supplement* and in product-specific documentation. Some user parameters have default values that are system parameters with the same names. These defaults can be set by **setsystemparams**.

User parameters, unlike system parameters, can be set without supplying a password. Alterations to user parameters are subject to **save** and **restore**. In an interpreter that supports multiple execution contexts, user parameters are maintained separately for each context.

Example

```
<< /MaxFontItem 7500 >> setuserparams
```

This example attempts to set the **MaxFontItem** user parameter to 7500.

Errors: **invalidaccess, limitcheck, stackunderflow, typecheck**
See Also: **currentuserparams, setsystemparams, setdevparams**

setvmthreshold

int **setvmthreshold** –



sets the allocation threshold used to trigger garbage collection. If *int* is less than the implementation-dependent minimum value, the threshold is set to that minimum value. If *int* is greater than the implementation-dependent maximum value, the threshold is set to that maximum value. If *int* is −1, the threshold is set to the implementation-dependent default value. All other negative values of *int* result in a **rangecheck** error.

Modifications to the allocation threshold parameter are subject to **save** and **restore**. In an interpreter that supports multiple contexts, this parameter is maintained separately for each context.

The parameter specified by **setvmthreshold** is the same as the **VMThreshold** user parameter set by **setuserparams** (see Appendix C).

Errors: **rangecheck**
See Also: **setuserparams**

shareddict – **shareddict** *dict*



is the same dictionary as **globaldict**. The name **shareddict** is defined for compatibility with earlier PostScript interpreter implementations.

Errors: **stackoverflow**
See Also: **globaldict**

SharedFontDirectory – **SharedFontDirectory** *dict*



is the same dictionary as **GlobalFontDirectory**. The name **SharedFontDirectory** is defined for compatibility with earlier PostScript interpreter implementations.

Errors: **stackoverflow**
See Also: **GlobalFontDirectory**

shfill *dict* **shfill** –



paints the shape and color shading described by a shading dictionary, subject to the current clipping path. The current path and current color in the graphics state are neither used nor altered. The effect of **shfill** is different from that of painting a path using a color defined as a shading pattern; see Section 4.9.3, "Shading Patterns."

dict is a shading dictionary defining the shape to be filled and the gradient fill to be used. All coordinates in the dictionary are interpreted in current user space. All color values are interpreted in the color space identified by the dictionary's **ColorSpace** entry. The **Background** entry, if present, is ignored.

shfill should be applied only to bounded or geometrically defined shadings. If applied to an unbounded shading, it will paint the corresponding field of color across the entire current clipping region, which may be time-consuming.

If the shading dictionary contains a **DataSource** entry that is a file, **shfill** will read the file exactly once. It is permissible to read from the file returned by **currentfile**, thereby reading in-line data from the executing PostScript program itself, immediately following the invocation of the **shfill** operator. Since the shading data continues until end-of-file, it is necessary to employ a filter, such as **SubFileDecode**, to provide an explicit end-of-data indication. (This differs from the case in which a shading dictionary is used to define a shading pattern, where the **DataSource** must be a positionable file, usually a **ReusableStreamDecode** filter.)

Execution of this operator is not permitted in certain circumstances; see Section 4.8, "Color Spaces."

Errors: **rangecheck, undefinedresult**
See Also: **fill, eofill, stroke**

show

string **show** –

paints glyphs for the characters identified by the elements of *string* on the current page starting at the current point, using the font face, size, and orientation specified by the current font (as returned by **currentfont**). The spacing from each glyph to the next is determined by the glyph's width, which is an (x, y) displacement that is part of the glyph description. When it is finished, **show** adjusts the current point in the graphics state by the sum of the widths of all the glyphs shown. **show** requires that the current point initially be defined (for example, by **moveto**); otherwise, a **nocurrentpoint** error occurs.

If a character code would index beyond the end of the font's **Encoding** array, or if the character mapping algorithm goes out of bounds in other ways, a **rangecheck** error occurs.

See Chapter 5 for complete information about the definition, manipulation, and rendition of fonts.

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **ashow, awidthshow, widthshow, kshow, cshow, xshow, yshow, xyshow, charpath, moveto, setfont**

showpage

– **showpage** –

transmits the contents of the current page to the current output device, causing any marks painted on the page to be rendered on the output medium. **showpage** then erases the current page and reinitializes the graphics state in preparation for composing the next page. (The actions of **showpage** may be modified by the **EndPage** procedure, as discussed below.)

If the current device is a page device that was installed by **setpagedevice** *(LanguageLevel 2)*, the precise behavior of **showpage** is determined by the values of parameters in the page device dictionary (see Sections 6.1.1, "Page Device Dictionary," and 6.2, "Page Device Parameters"). Parameters affecting the behavior of **showpage** include **NumCopies**, **Collate**, **Duplex**, and perhaps others as well.

Whether or not the current device is a page device, the precise manner in which the current page is transmitted is device-dependent. For certain devices (such as displays), no action is required, because the current page is visible while it is being composed.

The main actions of **showpage** are as follows:

1. Executes the **EndPage** procedure in the page device dictionary, passing an integer page count on the operand stack along with a reason code indicating that the procedure was called from **showpage**; see Section 6.2.6, "Device Initialization and Page Setup," for more information.
2. If the boolean result returned by the **EndPage** procedure is *true*, transmits the page's contents to the current output device and performs the equivalent of an **erasepage** operation, clearing the contents of raster memory in preparation for the next page. If the **EndPage** procedure returns *false*, **showpage** skips this step.
3. Performs the equivalent of an **initgraphics** operation, reinitializing the graphics state for the next page.
4. Executes the **BeginPage** procedure in the page device dictionary, passing an integer page count on the operand stack.

If the **BeginPage** or **EndPage** procedure invokes **showpage**, an **undefined** error occurs.

For a device that produces output on a physical medium such as paper, **showpage** can optionally transmit multiple copies of the page in step 2 above. In LanguageLevel 2 or 3, the page device parameter **NumCopies** specifies the number of copies to be transmitted. In LanguageLevel 1 (or in higher LanguageLevels if **NumCopies** is *null*), the number of copies is given by the value associated with the name **#copies** in the naming environment defined by the current dictionary stack. (The default value of **#copies** is 1, defined in **userdict**.) For example, the code

```
/#copies 5 def
showpage
```

prints five copies of the current page, then erases the current page and reinitializes the graphics state.

Errors: **limitcheck, undefined**
See Also: **copypage, erasepage, setpagedevice**

sin *angle* **sin** *real*

returns the sine of *angle*, which is interpreted as an angle in degrees. The result is a real number.

Errors: **stackunderflow, typecheck**
See Also: **cos, atan**

sqrt *num* **sqrt** *real*

returns the square root of *num*, which must be a nonnegative number. The result is a real number.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **exp**

srand *int* **srand** –

initializes the random number generator with the seed *int,* which may be any integer value. Executing **srand** with a particular value causes subsequent invocations of **rand** to generate a reproducible sequence of results. In an interpreter that supports multiple execution contexts, the random number state is maintained separately for each context.

Errors: **stackunderflow, typecheck**
See Also: **rand, rrand**

stack $\vdash$ *any*$_1$ … *any*$_n$ **stack** $\vdash$ *any*$_1$ … *any*$_n$

writes text representations of every object on the stack to the standard output file, but leaves the stack unchanged. **stack** applies the **=** operator to each element of the stack, starting with the topmost element. See the **=** operator for a description of its effects.

Errors: none
See Also: **pstack, =, ==, count**

stackoverflow *(error)*

The operand stack has grown too large; too many objects have been pushed on the stack and not popped off. See Appendix B for the limit on the size of the operand stack.

Before invoking this error, the interpreter creates an array containing all elements of the operand stack (stored as if by **astore**), resets the operand stack to empty, and pushes the array on the operand stack.

stackunderflow *(error)*

An attempt has been made to remove an object from the operand stack when it is empty. This usually occurs because some operator did not have all of its required operands on the stack.

StandardEncoding – **StandardEncoding** *array*

pushes the standard encoding vector on the operand stack. This is a 256-element literal array object, indexed by character codes, whose values are the character names for those codes. See Section 5.3, "Character Encoding," for an explanation of encoding vectors. **StandardEncoding** is not an operator; it is a name in **systemdict** associated with the array object.

StandardEncoding is the Adobe standard encoding vector used by most Latin-text fonts, but not by special fonts, such as Symbol. A new Latin-text font having no unusual encoding requirements should specify its **Encoding** entry to be the value of **StandardEncoding** rather than define its own private array. The contents of the standard encoding vector are tabulated in Appendix E.

Errors: **stackoverflow**
See Also: **ISOLatin1Encoding, findencoding**

start – **start** –

is executed by the PostScript interpreter when it starts up. After setting up the virtual memory (restoring it from a file, if appropriate), the interpreter executes the name **start** in the context of the default dictionary stack (**systemdict**, **globaldict**, and **userdict**). The procedure associated with the name **start** is expect-

ed to provide whatever top-level control is required—for example, for receiving page descriptions, interacting with a user, or recovering from errors. The precise definition of **start** depends on the environment in which the PostScript interpreter is operating. It is not of any interest to ordinary PostScript programs and the effect of executing it explicitly is undefined.

Errors: none
See Also: **quit**

StartData

key int **StartData** – *(**FontSetInit** procedure set)*
key int name **StartData** – *(**FontSetInit** procedure set)*

string int **StartData** – *(**CIDInit** procedure set)*



introduces the binary data section of a font set or Type 0 CIDFont file and registers the resulting dictionary as an instance in the **FontSet** or **CIDFont** resource category.

For either operator, *int* specifies the number of bytes in the binary data section that follows. The data begins immediately after the white-space character that terminates the invocation of **StartData**. If **StartData** is invoked directly as part of a PostScript program, it consumes this data and incorporates it into the dictionary being constructed (as the value of the **GlyphData** entry). However, if **StartData** is invoked from within a resource file being loaded by the **findresource** operator, it does not load the data into virtual memory; instead, it arranges for the data to be accessed from the file system dynamically, as needed, during glyph rendering.

For a font set, **StartData** expects to be invoked when the dictionary stack contains the **FontSetInit** procedure set dictionary. It creates a **FontSet** instance from the binary data, which is expected to conform to the Compact Font Format (CFF) specification or to other recognized multiple-font formats, such as Chameleon. If the data consists of Chameleon font descriptors, the *name* operand must be used to specify the name of the associated master font. After creating the **FontSet** instance, **StartData** invokes the equivalent of

key /FontSet defineresource pop

and removes one dictionary from the dictionary stack. See Section 5.8.1, "Type 2 and Type 14 Fonts (CFF and Chameleon)" for information on the fonts stored in these resources.

For a Type 0 CIDFont, **StartData** expects to be invoked when the dictionary stack contains the **CIDInit** procedure set dictionary and the CIDFont dictionary being built. The *string* operand specifies the format of the following data as either Binary

(which is strongly recommended) or Hex. From the data, **StartData** completes the construction of the CIDFont dictionary and then invokes the equivalent of

```
CIDFontName /CIDFont defineresource pop
```

Finally, it removes two dictionaries from the dictionary stack. See "Type 0 CIDFonts" on page 371 for more information.

Errors: **invalidfont, ioerror, stackunderflow, typecheck, VMerror**
See Also: **findresource**

startjob *bool*$_1$ *password* **startjob** *bool*$_2$



conditionally starts a new job whose execution may alter the initial virtual memory for subsequent jobs (see Section 3.7.7, "Job Execution Environment"). *bool*$_1$ specifies whether the new job's side effects are to be persistent.

The behavior of **startjob** depends on whether the following three conditions are true:

- The current execution context supports job encapsulation—in other words, is under the control of a job server.
- *password* is correct—in other words, matches the **StartJobPassword** or **SystemParamsPassword** system parameter.
- The current level of **save** nesting is no deeper than it was at the time the current job started.

If all three conditions are satisfied, **startjob** performs the following actions:

1. Ends the current job—in other words, resets the stacks and, if the current job was encapsulated, performs a **restore** operation.
2. Begins a new job. If *bool*$_1$ is *true*, the usual **save** operation at the beginning of the job is omitted, enabling the new job to make persistent alterations to the initial VM. If *bool*$_1$ is *false*, the usual **save** operation is performed, encapsulating the new job.
3. Returns *true* on the operand stack.

If any of the three conditions is not satisfied, **startjob** pushes *false* on the operand stack and has no other effect.

password is a string that authorizes switching between encapsulated and unencapsulated jobs, as well as between an ordinary encapsulated job and a system administrator job. If *password* is an integer, it is first converted to a string, as if by the **cvs**

operator. It is compared to the **StartJobPassword** and **SystemParamsPassword** system parameters; see Section C.3.1, "Passwords," for details.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **exitserver, setsystemparams, save, restore**

status *file* **status** *bool*
filename **status** *pages bytes referenced created true* *(if found)*
false *(if not found)*

If the operand is a file object, **status** returns *true* if it is still valid (that is, is associated with an open file), or *false* otherwise.

If the operand is a string, **status** treats it as a file name according to the conventions described in Section 3.8.2, "Named Files." If there is a file by that name, **status** pushes four integers of status information followed by the value *true*; otherwise, it pushes *false*. The four integer values are:

pages The storage space occupied by the file, in implementation-dependent units.

bytes The length of the file in bytes.

referenced The date and time the file was last referenced for reading or writing. This value is interpreted according to the conventions of the underlying operating system. The only assumption that a program can make is that larger values indicate later times.

created The date and time the contents of the file were created.

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **file, closefile, filenameforall**

statusdict – **statusdict** *dict*

pushes a product-dependent dictionary on the operand stack. **statusdict** is not an operator; it is a name associated with the dictionary in **systemdict**. The **statusdict** dictionary is in local VM and is writeable.

statusdict contains product-dependent operators and other data whose names and values vary from product to product, and sometimes from one version of a product to another. Information in **statusdict** is associated with unique features

of a product that cannot be accessed in any standard way. The contents of **statusdict** are not documented here, but in product-specific documentation.

In LanguageLevel 1, **statusdict** includes operators to select print-engine features, to set communication parameters, and to control other aspects of the interpreter's operating environment. In LanguageLevel 2 or 3, most of these functions have been subsumed by standard operators, such as **setpagedevice**, **setdevparams**, and **setsystemparams**.

statusdict is not necessarily defined in all products. Any reference to **statusdict** in a PostScript program impairs the portability of that program.

Errors: **stackoverflow, undefined**

stop – **stop** –

terminates execution of the innermost, dynamically enclosing instance of a **stopped** context, without regard to lexical relationship. A **stopped** context is a procedure or other executable object invoked by the **stopped** operator. **stop** pops the execution stack down to the level of the **stopped** operator. The interpreter then pushes the boolean value *true* on the operand stack and resumes execution at the next object in normal sequence after the **stopped** operator. It thus appears that **stopped** returned the value *true*, whereas it normally returns *false*.

stop does not affect the operand stack or dictionary stack. Any objects pushed on these stacks during the execution of the **stopped** context remain after the context is terminated.

If **stop** is executed when there is no enclosing **stopped** context, the interpreter prints an error message and executes the built-in operator **quit**. This never occurs during execution of ordinary user programs.

Errors: none
See Also: **stopped, exit**

stopped *any* **stopped** *bool*

executes *any*, which is typically, but not necessarily, a procedure, executable file, or executable string object. If *any* runs to completion normally, **stopped** returns *false* on the operand stack. If *any* terminates prematurely as a result of executing **stop**, **stopped** returns *true*. Regardless of the outcome, the interpreter resumes execution at the next object in normal sequence after **stopped**.

This mechanism provides an effective way for a PostScript program to "catch" errors or other premature terminations, retain control, and perhaps perform its own error recovery. See Section 3.11, "Errors."

When an error occurs, the standard error handler sets **newerror** to *true* in the **$error** dictionary. When using **stopped** to catch and continue from an error (without invoking **handleerror**), it is prudent to explicitly reset **newerror** to *false* in **$error**; otherwise, any subsequent execution of **stop** may result in inadvertent reporting of the leftover error. Also, note that the standard error handler sets the VM allocation mode to local.

Example

```
{ … } stopped
        {handleerror}
    if
```

If execution of the procedure { … } causes an error, the default error reporting procedure is invoked (by **handleerror**). In any event, normal execution continues at the token following the **if** operator.

Errors: **stackunderflow**
See Also: **stop**

store *key value* **store** –

searches for *key* in each dictionary on the dictionary stack, starting with the topmost (current) dictionary. If *key* is found in some dictionary, **store** replaces its value by the *value* operand; otherwise, **store** creates a new entry with *key* and *value* in the current dictionary.

If the chosen dictionary is in global VM and *value* is a composite object whose value is in local VM, an **invalidaccess** error occurs (see Section 3.7.2, "Local and Global VM").

Example

```
/abc 123 store

/abc where
        {}
        {currentdict}
    ifelse
/abc 123 put
```

The two code fragments above have the same effect.

Errors: **dictfull, invalidaccess, limitcheck, stackunderflow**
See Also: **def, put, where, load**

string

int **string** *string*

creates a string of length *int*, each of whose elements is initialized with the integer 0, and pushes this string on the operand stack. The *int* operand must be a nonnegative integer not greater than the maximum allowable string length (see Appendix B). The string is allocated in local or global VM according to the current VM allocation mode; see Section 3.7.2, "Local and Global VM."

Errors: **limitcheck, rangecheck, stackunderflow, typecheck, VMerror**
See Also: **length, type**

stringwidth

string **stringwidth** w_x w_y

calculates the change in the current point that would occur if *string* were given as the operand to **show** with the current font. w_x and w_y are computed by adding together the width vectors of all the individual glyphs for *string* and converting the result to user space. They form a distance vector in the x and y dimensions describing the width of the glyphs for the entire string in user space. See Section 5.4, "Glyph Metric Information," for a discussion of glyph widths.

To obtain the glyph widths, **stringwidth** may execute the descriptions of one or more of the glyphs in the current font and may cause the results to be placed in the font cache. However, **stringwidth** prevents the graphics operators that are executed from painting anything onto the current page.

Note that the width returned by **stringwidth** is defined as movement of the current point. It has nothing to do with the dimensions of the glyph outlines (see **charpath** and **pathbbox**).

Errors: **invalidaccess, invalidfont, rangecheck, stackunderflow, typecheck**
See Also: **show, setfont**

stroke – **stroke** –

paints a line centered on the current path, with sides parallel to the path segments. The line's graphical properties are defined by various parameters of the graphics state. Its thickness is determined by the current line width parameter (see **setlinewidth**) and its color by the current color (see **setcolor**). The joints between connected path segments and the ends of open subpaths are painted with the current line join (see **setlinejoin**) and the current line cap (see **setlinecap**), respectively. The line is either solid or broken according to the dash pattern established by **setdash**. Uniform stroke width can be ensured by enabling automatic stroke adjustment (see **setstrokeadjust**). All of these graphics state parameters are consulted at the time **stroke** is executed; their values during the time the path is being constructed are irrelevant.

If a subpath is degenerate (consists of a single-point closed path or of two or more points at the same coordinates), **stroke** paints it only if round line caps have been specified, producing a filled circle centered at the single point. If butt or projecting square line caps have been specified, **stroke** produces no output, because the orientation of the caps would be indeterminate. A subpath consisting of a single-point open path produces no output.

After painting the current path, **stroke** clears it with an implicit **newpath** operation. To preserve the current path across a **stroke** operation, use the sequence

```
gsave
  fill
grestore
```

Errors: **limitcheck**
See Also: **setlinewidth, setlinejoin, setlinecap, setmiterlimit, setdash, setstrokeadjust, ustroke, fill**

strokepath – *s***trokepath** –

replaces the current path with one enclosing the shape of the line that the **stroke** operator would draw if applied to the current path. The result is not influenced by the current clipping path. The rules for degenerate subpaths (see **stroke**) apply to **strokepath** as well: if **stroke** would produce no output, **strokepath** will produce an empty subpath.

The path produced by **strokepath** is suitable for use as the implicit operand to a subsequent **fill**, **clip**, or **pathbbox** operation. In general, it is not suitable for **stroke**, as it may contain interior segments or disconnected subpaths produced by **strokepath**'s conversion from stroke to outline.

Errors: **limitcheck**
See Also: **stroke, fill, clip, charpath, pathbbox**

sub *num*$_1$ *num*$_2$ **sub** *difference*

returns the result of subtracting *num*$_2$ from *num*$_1$. If both operands are integers and the result is within integer range, the result is an integer; otherwise, the result is a real number.

Errors: **stackunderflow, typecheck, undefinedresult**
See Also: **add, div, mul, idiv, mod**

syntaxerror *(error)*

The scanner has encountered program text that does not conform to the PostScript language syntax rules (see Section 3.2, "Syntax"). This can occur either during interpretation of an executable file or executable string object or during explicit invocation of the **token** operator.

Because the syntax of the PostScript language is simple, the set of possible causes for a **syntaxerror** is very small:

- An opening string or procedure bracket—(, <, <~, or {—is not matched by a corresponding closing bracket before the end of the file or string being interpreted.
- A closing string or procedure bracket—), >, ~>, or }—is not matched by a corresponding previous opening bracket.
- A character other than a hexadecimal digit or white-space character appears within a hexadecimal string literal bracketed by < … >.
- An encoding violation occurs in an ASCII base-85 string literal bracketed by <~ … ~>.
- A binary token or binary object sequence has incorrect structure (see Section 3.14, "Binary Encoding Details").

Erroneous tokens, such as malformed numbers, do not produce a **syntaxerror**; such tokens are instead treated as name objects (often producing an **undefined** error when executed). Tokens that exceed implementation limits, such as names that are too long or numbers whose values are too large, produce a **limitcheck** error (see Appendix B).

systemdict – **systemdict** *dict*

pushes the dictionary object **systemdict** on the operand stack (see Section 3.7.5, "Standard and User-Defined Dictionaries"). **systemdict** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **errordict, globaldict, userdict**

timeout *(error)*

A time limit has been exceeded; that is, a PostScript program has executed for too long or has waited too long for some external event to occur.

Execution of **timeout** is sandwiched between execution of two objects being interpreted in normal sequence. Unlike most other errors, the occurrence of a **timeout** error does not cause the object being executed to be pushed on the operand stack, nor does it disturb the operand stack in any way.

The PostScript language does not define any standard causes for **timeout** errors. However, a PostScript interpreter running in a particular environment may provide a set of timeout facilities appropriate for that environment.

token

file **token**	*any true*	*(if found)*
	false	*(if not found)*
string **token**	*post any true*	*(if found)*
	false	*(if not found)*

reads characters from *file* or *string*, interpreting them according to the PostScript language syntax rules (see Section 3.2, "Syntax"), until it has scanned and constructed an entire object.

In the *file* case, **token** normally pushes the scanned object followed by *true*. If **token** reaches end-of-file before encountering any characters besides white-space characters, it returns *false*.

In the *string* case, **token** normally pushes *post* (the substring of *string* beyond the portion consumed by **token**), the scanned object, and *true*. If **token** reaches the end of *string* before encountering any characters besides white-space characters, it simply returns *false*.

In either case, the *any* result is an ordinary object. It may be a simple object—an integer, a real number, or a name—or a composite object—a string bracketed by (…) or a procedure bracketed by { … }. The object returned by **token** is the same as the object that would be encountered by the interpreter if *file* or *string* were executed directly. However, **token** scans just a single object and it always pushes that object on the operand stack rather than executing it.

token consumes all characters of the token and sometimes the terminating character as well. If the token is a name or a number followed by a white-space character, **token** consumes the white-space character (only the first one if there are several). If the token is terminated by a special character that is part of the token—), >, or }—**token** consumes that character, but no following ones. If the token is terminated by a special character that is part of the next token—/, (, <, [, or {—**token** does not consume that character, but leaves it in the input sequence. If the token is a binary token or a binary object sequence, **token** consumes no additional characters.

Examples

```
(15 (St1) {1 2 add}) token   ⇒  ((St1) {1 2 add}) 15 true
((St1) {1 2 add}) token      ⇒  ({1 2 add}) (St1) true
({1 2 add}) token            ⇒  () {1 2 add} true
() token                     ⇒  false
```

Errors: **invalidaccess, ioerror, limitcheck, stackoverflow, stackunderflow, syntaxerror, typecheck, undefinedresult, VMerror**
See Also: **search, anchorsearch, read**

transform

x y **transform** *x′ y′*
x y matrix **transform** *x′ y′*

applies a transformation matrix to the coordinates (*x*, *y*), returning the transformed coordinates (*x′*, *y′*). The first form of the operator uses the current transformation matrix in the graphics state to transform user space coordinates to device space. The second form applies the transformation specified by the *matrix* operand rather than the CTM.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **itransform, dtransform, idtransform**

translate $t_x\ t_y$ **translate** –
$t_x\ t_y$ *matrix* **translate** *matrix*

moves the origin of the user coordinate space by t_x units horizontally and t_y units vertically, or returns a matrix representing this transformation. The orientation of the axes and the sizes of the coordinate units are unaffected.

The transformation is represented by the matrix

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

The first form of the operator applies this transformation to the user coordinate system by concatenating matrix T with the current transformation matrix (CTM); that is, it replaces the CTM with the matrix product $T \times \text{CTM}$. The second form replaces the value of the *matrix* operand with an array representing matrix T and pushes the result back on the operand stack without altering the CTM. See Section 4.3.3, "Matrix Representation and Manipulation," for a discussion of how matrices are represented as arrays.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **setmatrix, currentmatrix, scale, rotate, concat**

true – **true** *true*

pushes a boolean object whose value is *true* on the operand stack. **true** is not an operator; it is a name in **systemdict** associated with the boolean value *true*.

Errors: **stackoverflow**
See Also: **false, and, or, not, xor**

truncate *num*$_1$ **truncate** *num*$_2$

truncates *num*$_1$ toward 0 by removing its fractional part. The type of the result is the same as the type of the operand.

Examples

```
  3.2 truncate  ⇒   3.0
 –4.8 truncate  ⇒  –4.0
   99 truncate  ⇒   99
```

Errors: **stackunderflow, typecheck**
See Also: **ceiling, floor, round, cvi**

type *any* **type** *name*

returns a name object that identifies the type of the object *any*. The possible names that **type** can return are as follows:

arraytype	nametype
booleantype	nulltype
dicttype	operatortype
filetype	packedarraytype *(LanguageLevel 2)*
fonttype	realtype
gstatetype *(LanguageLevel 2)*	savetype
integertype	stringtype
marktype	

The name fonttype identifies an object of type *fontID*. It has nothing to do with a font dictionary, which is identified by dicttype the same as any other dictionary.

The returned name has the executable attribute. This makes it convenient to perform type-dependent processing of an object simply by executing the name returned by **type** in the context of a dictionary that defines all the type names to have procedure values (this is how the == operator works).

The set of types is subject to enlargement in future revisions of the language. A program that examines the types of arbitrary objects should be prepared to behave reasonably if **type** returns a name that is not in this list.

Errors: **stackunderflow**

typecheck *(error)*

An operand's type is different from what an operator expects, or an operand is a literal array or a literal packed array when an operator expects a procedure (an executable array or an executable packed array). This is probably the most frequently occurring error. It is often the result of faulty stack manipulation, such as operands supplied in the wrong order or procedures leaving results on the stack when they are not supposed to.

Certain operators require dictionaries or other composite objects as operands, constructed according to specific rules (for example, pattern dictionaries or user paths). A **typecheck** error can occur if the contents of such objects are of incorrect type or are otherwise malformed.

uappend *userpath* **uappend** –



interprets a user path definition (see Section 4.6, "User Paths") and appends the result to the current path in the graphics state. All operator names mentioned in the user path have their standard definitions, unaffected by any name redefinitions that may have occurred.

If *userpath* is an ordinary user path (an array or packed array whose length is at least 5), **uappend** is equivalent to the following code:

```
systemdict begin        % Ensure standard operator meanings
   cvx exec             % Interpret userpath
end
```

If *userpath* is an encoded user path, **uappend** interprets it and performs the encoded operations. It does not matter whether the user path object is literal or executable.

The bounding box specified by the **setbbox** operator in the user path applies only to elements of the user path itself. It does not apply to other elements of the current path constructed either before or after the execution of **uappend**.

A **ucache** operation appearing in the user path may or may not have an effect, depending on the environment in which **uappend** is executed. If the current path is initially empty and no path construction operators are executed after **uappend**, a subsequent painting operator may or may not access the user path cache; otherwise, it definitely will not. This is particularly useful in the case of **clip**.

As part of its execution, **uappend** temporarily rounds the t_x and t_y components of the current transformation matrix to the nearest integer values. The reason for this is discussed in Section 4.6.4, "User Path Operators."

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **upath, ucache**

ucache – **ucache** –



notifies the PostScript interpreter that the user path in which the operator occurs is to be retained in the user path cache if it is not already there (see Section 4.6.3, "User Path Cache"). If present, this operator must appear as the first element of a user path definition (before the mandatory **setbbox**); if executed outside a user path definition, the operator does nothing.

The **ucache** operator has no effect of its own when executed. It is useful only with a user path painting operator, such as **ufill** or **ustroke**, that takes the enclosing user path as an operand. If the user path is not already in the cache, the painting operator performs the path construction operations specified in the user path definition and places the results (referred to as the *reduced path*) in the cache. If the user path is already present in the cache, the painting operator does not interpret the user path, but rather obtains and uses the reduced path from the cache.

Errors: none
See Also: **uappend, upath**

ucachestatus – **ucachestatus** *mark bsize bmax rsize rmax blimit*



reports the current consumption and maximum limit for two quantities related to the user path cache: bytes of storage occupied by reduced paths (*bsize* and *bmax*) and total number of reduced paths cached (*rsize* and *rmax*). It also reports the limit on the number of bytes that can be occupied by a single reduced path (*blimit*); reduced paths that exceed this limit are not cached. A PostScript program can change *blimit* (see **setucacheparams**); all other results are for information only.

The number of result values returned on the operand stack is variable. They are preceded by a mark object pushed on the stack as a delimiter, enabling a program to determine how many values were returned (using **counttomark**) and to discard any unused ones (using **cleartomark**).

The *bsize*, *bmax*, and *blimit* results reported by **ucachestatus** are the same as the system parameters **CurUPathCache** and **MaxUPathCache** and the user parameter **MaxUPathItem** (see Appendix C).

Errors: **stackoverflow**
See Also: **setucacheparams, setsystemparams, setuserparams**

ueofill *userpath* **ueofill** –



interprets a user path definition (see Section 4.6, "User Paths") and fills the resulting path as if by the **eofill** operator. The even-odd rule is used to determine what points lie inside the path (see "Even-Odd Rule" on page 196). In all other respects, the behavior of **ueofill** is identical to that of **ufill**.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **eofill, ufill**

ufill *userpath* **ufill** –



interprets a user path definition (see Section 4.6, "User Paths") and fills the resulting path as if by the **fill** operator. The nonzero winding number rule is used to determine what points lie inside the path (see "Nonzero Winding Number Rule" on page 195). The entire operation is effectively bracketed between **gsave** and **grestore**, so it has no lasting effect on the graphics state.

ufill is equivalent to the following code:

```
gsave                % Save graphics state
   newpath           % Clear current path
   uappend           % Interpret userpath
   fill              % Fill the path
grestore             % Restore graphics state
```

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **fill, ueofill, uappend**

undef *dict key* **undef** –



removes *key* and its associated value from the dictionary *dict*. *dict* does not need to be on the dictionary stack. No error occurs if *key* is not present in *dict*.

If the value of *dict* is in local VM, the effect of **undef** can be undone by a subsequent **restore** operation. That is, if *key* was present in *dict* at the time of the matching **save** operation, **restore** will reinstate *key* and its former value. But if *dict* is in global VM, the effect of **undef** is permanent.

Errors: **invalidaccess, stackunderflow, typecheck**
See Also: **def, put, undefinefont**

undefined *(error)*

A name used as a dictionary key in some context cannot be found. This occurs if a name is looked up explicitly in a specified dictionary (**get**) or in the current dictionary stack (**load**) and is not found. It also occurs if an executable name is encountered by the interpreter and is not found in any dictionary on the dictionary stack.

A few PostScript operators are disabled in certain contexts—for example, it is illegal to execute **image**, or operators that specify colors or set color-related parameters in the graphics state, after invoking **setcachedevice** or **setcachedevice2** in a **BuildChar**, **BuildGlyph**, or **CharStrings** procedure. Attempting to execute such disabled operators results in an **undefined** error.

See Also: known, where, load, exec, get

undefinedfilename *(error)*

A file identified by a *filename* string operand of **file**, **run**, **deletefile**, or **renamefile** cannot be found or cannot be opened. The **undefinedfilename** error also occurs if the special file %statementedit or %lineedit is opened when the standard input file has reached end-of-file.

undefinedresource *(error)*



A named resource instance sought by **findresource** cannot be found; that is, no such instance exists either in virtual memory or in external storage. This error arises only in the case of **findresource** with a defined resource category. If the category itself is not defined, resource operators execute the **undefined** error.

See Also: findresource

undefinedresult *(error)*

A numeric computation would produce a meaningless result or one that cannot be represented as a number. Possible causes include numeric overflow or underflow, division by 0, or inverse transformation of a noninvertible matrix. A large number of graphics and font operators can generate an **undefinedresult** error if the current transformation matrix is not invertible (if it is scaled by 0, for in-

stance). See Appendix B for the limits of the values representable as integers and real numbers.

undefinefont *key* **undefinefont** –



removes *key* and its associated value (a font or CIDFont dictionary) from the font directory, reversing the effect of a previous **definefont** operation. **undefinefont** is a special case of the **undefineresource** operator applied to the **Font** category. For details, see **undefineresource** and Section 3.9, "Named Resources."

Errors: **stackunderflow, typecheck**
See Also: **definefont, undefineresource**

undefineresource *key category* **undefineresource** –



removes the named resource instance identified by *key* from the specified resource category. This undoes the effect of a previous **defineresource**. If no such resource instance exists in virtual memory, **undefineresource** does nothing; no error occurs. However, the resource category must exist, or else an **undefined** error occurs.

Local and global resource definitions are maintained separately; the precise effect of **undefineresource** depends on the current VM allocation mode:

- In local VM allocation mode, **undefineresource** removes a local definition if there is one. If there is a global definition with the same key, **undefineresource** does not disturb it; the global definition, formerly obscured by the local one, now reappears.
- In global VM allocation mode, **undefineresource** removes a local definition, a global definition, or both.

Depending on the resource category, **undefineresource** may have other side effects (see Section 3.9.2, "Resource Categories"); these side effects are determined by the **UndefineResource** procedure in the category implementation dictionary. However, **undefineresource** does not alter the resource instance in any way. If the instance is still accessible (say, stored directly in some dictionary or defined as a resource under another name), it can still be used in whatever ways are appropriate. The object becomes a candidate for garbage collection only if it is no longer accessible.

The effect of **undefineresource** is subject to normal VM semantics. In particular, removal of a local resource instance can be undone by a subsequent nonnested **restore**. In this case, the resource instance is not a candidate for garbage collection.

undefineresource removes the resource instance definition from VM only. If the resource instance also exists in external storage, it can still be found by **findresource**, **resourcestatus**, and **resourceforall**.

Errors: **stackunderflow, typecheck, undefined**
See Also: **defineresource, findresource, resourcestatus, resourceforall**

undefineuserobject

index **undefineuserobject** –



breaks the association between the nonnegative integer *index* and an object established by some previous execution of **defineuserobject**. It does so simply by replacing the specified **UserObjects** array element with the null object. This operator is equivalent to the following code:

```
userdict /UserObjects get
exch null put
```

undefineuserobject does not take any other actions, such as shrinking the **UserObjects** array. If *index* is not a valid index for the existing **UserObjects** array, a **rangecheck** error occurs. See Section 3.7.6, "User Objects."

There is no need to invoke **undefineuserobject** before invoking a **defineuserobject** operator that reuses the same index. The purpose of **undefineuserobject** is to eliminate references to objects that are no longer needed, which may enable the garbage collector to reclaim such objects.

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **defineuserobject, UserObjects**

unmatchedmark

(error)

The **]**, **>>**, **cleartomark**, **counttomark**, **setcacheparams**, or **setucacheparams** operator is seeking a mark object on the operand stack, but none is present.

unregistered *(error)*

An operator object has been executed for which the interpreter has no built-in action. This represents an internal malfunction in the PostScript interpreter and should not occur.

upath *bool* **upath** *userpath*



creates a new user path object equivalent to the current path in the graphics state. The current path itself is not disturbed. The result is an ordinary user path array, not an encoded user path. The *bool* operand specifies whether the user path should include a **ucache** operator as its first element.

upath creates a new executable array object of the appropriate length and fills it with the operands and operators needed to describe the current path. It is approximately equivalent to the following code:

```
[   exch {/ucache cvx} if
    pathbbox /setbbox cvx
    {/moveto cvx} {/lineto cvx} {/curveto cvx} {/closepath cvx} pathforall
] cvx
```

If the path ends with a **moveto** operation, **upath** adjusts the bounding box if necessary to enclose it. (Note that **pathbbox** disregards a trailing **moveto** operation.)

Since the current path's coordinates are maintained in device space, **upath** transforms them to user space while constructing the user path, using the inverse of the current transformation matrix. Applying **uappend** to the resulting user path will reproduce the original current path only if the same CTM is in effect as at the time the path was constructed.

If **charpath** was used to construct any portion of the current path from a font whose outlines are protected, **upath** is not allowed; its execution will produce an **invalidaccess** error (see **charpath**).

Errors: **invalidaccess, stackoverflow, typecheck, VMerror**
See Also: **uappend, ucache, pathforall**

usecmap *key* **usecmap** – *(**CIDInit** procedure set)*



specifies that the **CMap** resource instance identified by *key* is to be used; see Section 5.11.4, "CMap Dictionaries."

usefont *fontnum* **usefont** – *(**CIDInit** procedure set)*



specifies the font number of the font or CIDFont to which the subsequent CMap mapping operations apply; see Section 5.11.4, "CMap Dictionaries."

userdict – **userdict** *dict*

pushes the dictionary object **userdict** on the operand stack (see Section 3.7.5, "Standard and User-Defined Dictionaries"). **userdict** is not an operator; it is a name in **systemdict** associated with the dictionary object.

Errors: **stackoverflow**
See Also: **systemdict, globaldict, errordict**

UserObjects – **UserObjects** *array*



returns the current **UserObjects** array defined in **userdict**. **UserObjects** is not an operator; it is simply a name associated with an array in **userdict**. This array is created and managed by the operators **defineuserobject**, **undefineuserobject**, and **execuserobject**. It defines a mapping from small integers (used as array indices) to arbitrary objects (the elements of the array). See Section 3.7.6, "User Objects."

The **UserObjects** entry in **userdict** is present only if **defineuserobject** has been executed at least once. The length of the array depends on the index operands of all previous executions of **defineuserobject**.

Note that **defineuserobject**, **undefineuserobject**, and **execuserobject** operate on the value of **UserObjects** in **userdict**, without regard to the dictionaries currently on the dictionary stack. Defining **UserObjects** in some other dictionary on the dictionary stack changes the value returned by executing the name object **UserObjects**, but does not alter the behavior of the user object operators.

Although **UserObjects** is an ordinary array object, it should be manipulated only by the user object operators. Improper direct alteration of **UserObjects** can subsequently cause the user object operators to malfunction.

Errors: **stackoverflow, undefined**
See Also: **defineuserobject, undefineuserobject, execuserobject**

usertime – **usertime** *int*

returns the value of a clock that is incremented by 1 for every millisecond of execution by the PostScript interpreter. The value has no defined meaning in terms of calendar time or time of day; its only use is interval timing. The accuracy and stability of the clock depends on the environment in which the PostScript interpreter is running. As the time value becomes greater than the largest integer allowed in the implementation, it wraps to the smallest (most negative) integer.

In an interpreter that supports multiple execution contexts, the value returned by **usertime** reports execution time on behalf of the current context only.

Errors: **stackoverflow**
See Also: **realtime**

ustroke *userpath* **ustroke** –
userpath matrix **ustroke** –



interprets a user path definition (see Section 4.6, "User Paths") and strokes the resulting path as if by the **stroke** operator. The entire operation is effectively bracketed between **gsave** and **grestore**, so it has no lasting effect on the graphics state.

The first form of the operator is equivalent to the following code:

```
gsave                % Save graphics state
   newpath           % Clear current path
   uappend           % Interpret userpath
   stroke            % Stroke the path
grestore             % Restore graphics state
```

The second form concatenates *matrix* to the current transformation matrix after interpreting *userpath*, but before executing **stroke**. The resulting matrix affects the line width and the dash pattern, if any, but not the path itself. This form of **ustroke** is equivalent to the following code:

```
gsave                % Save graphics state
   newpath           % Clear current path
   exch uappend      % Interpret userpath
   concat            % Concatenate matrix to CTM
   stroke            % Stroke the path
grestore             % Restore graphics state
```

The main use of this second form is to compensate for variations in line width and dash pattern that occur if the CTM has been scaled by different amounts hor-

izontally and vertically. This is accomplished by defining *matrix* to be the inverse of the unequal scaling transformation.

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **stroke, uappend**

ustrokepath *userpath* **ustrokepath** –
userpath matrix **ustrokepath** –



interprets a user path definition (see Section 4.6, "User Paths") and replaces the current path with one enclosing the shape of the line that the **ustroke** operator would draw if applied to that user path. The resulting path is suitable for use as the implicit operand to a subsequent **fill**, **clip**, or **pathbbox** operation. In general, it is not suitable for **stroke**, as it may contain interior segments or disconnected subpaths produced by **ustrokepath**'s conversion from stroke to outline.

The first form of the operator is equivalent to the following code:

```
newpath                          % Clear current path
uappend                          % Interpret userpath
strokepath                       % Convert stroke to outline
```

The second form concatenates *matrix* to the current transformation matrix after interpreting *userpath*, but before executing **strokepath**. The CTM is restored to its original value afterward, so **ustrokepath** has no lasting effect on it. This form of **ustrokepath** is equivalent to the following code:

```
newpath                          % Clear current path
exch uappend                     % Interpret userpath
matrix currentmatrix exch        % Save CTM
    concat                       % Concatenate matrix to CTM
    strokepath                   % Convert stroke to outline
setmatrix                        % Restore original CTM
```

Errors: **invalidaccess, limitcheck, rangecheck, stackunderflow, typecheck**
See Also: **ustroke, strokepath**

version – **version** *string*

returns a string that identifies the version of the PostScript interpreter being used. This identification does not include information about the language features or the hardware or operating system environment in which the PostScript interpreter is running.

Errors: **stackoverflow**
See Also: **languagelevel, product, revision, serialnumber**

VMerror *(error)*

An error has occurred in the virtual memory (VM) machinery. The most likely problems are:

- An attempt to create a new composite object (string, array, dictionary, or packed array) would exhaust VM resources. Either the program's requirements exceed available capacity or, more likely, the program has failed to use the **save**-**restore** facility appropriately (see Section 3.7, "Memory Management").
- The interpreter has attempted to perform an operation that should be impossible due to access restrictions (for example, storing into **systemdict**, which is read-only). This represents an internal error in the interpreter.

The default handler for this error, unlike those for all other errors, does not snapshot the stacks.

vmreclaim *int* **vmreclaim** –



controls the garbage collection machinery as specified by *int*:

–2	Disables automatic collection in both local and global VM
–1	Disables automatic collection in local VM
0	Enables automatic collection
1	Performs immediate collection in local VM
2	Performs immediate collection in local and global VM

Garbage collection causes the memory occupied by the values of inaccessible objects to be reclaimed and made available for reuse. It does not have any effects that are visible to the PostScript program. There is normally no need to execute the **vmreclaim** operator, because garbage collection is invoked automatically when

necessary. However, there are a few situations in which this operator may be useful:

- In an interactive application that is temporarily idle, an immediate garbage collection can be invoked to put the idle time to good use. This defers the need to perform an automatic collection subsequently. In a context that is under the control of a job server (described in Section 3.7.7, "Job Execution Environment"), garbage collection is invoked automatically between jobs.
- For meaningful results when monitoring the VM consumption of a program, garbage collection must be invoked before **vmstatus** is executed.
- For repeatable results when measuring a program's execution time, automatic garbage collection must be disabled.

The negative values that disable garbage collection apply only to the current execution context; they do not prevent collection from occurring during execution of other contexts. Note that disabling garbage collection for too long may eventually cause a program to run out of memory and fail with a **VMerror**.

Executing **vmreclaim** with an operand of 0, –1, or –2 has the same effect as setting the **VMReclaim** user parameter to the same value by means of **setuserparams** (see Appendix C).

Errors: **rangecheck, stackunderflow, typecheck**
See Also: **setvmthreshold, setuserparams**

vmstatus – **vmstatus** *level used maximum*

returns three integers describing the state of the PostScript interpreter's virtual memory (VM). *level* is the current depth of **save** nesting—in other words, the number of **save** operations that have not been matched by a **restore** operation. *used* and *maximum* measure VM resources in units of 8-bit bytes; *used* is the number of bytes currently in use and *maximum* is the maximum available capacity.

VM consumption is monitored separately for local and global VM. The *used* and *maximum* values apply to either local or global VM according to the current VM allocation mode (see **setglobal**).

The *used* value is meaningful only immediately after a garbage collection has taken place (see **vmreclaim**). At other times, it may be too large because it includes memory occupied by objects that have become inaccessible, but have not yet been reclaimed.

The *maximum* value is an estimate of the maximum size to which the current VM (local or global) could grow, assuming that all other uses of available memory re-

main constant. Because that assumption is never valid in practice, there is some uncertainty about the *maximum* value. Also, in some environments (workstations, for instance), the PostScript interpreter can obtain more memory from the operating system. In this case, memory is essentially inexhaustible and the *maximum* value is meaningless—it is an extremely large number.

Errors: **stackoverflow**
See Also: **setuserparams**

wcheck

array **wcheck** *bool*
packedarray **wcheck** *false*
dict **wcheck** *bool*
file **wcheck** *bool*
string **wcheck** *bool*

tests whether the operand's access permits its value to be written explicitly by PostScript operators. It returns *true* if the operand's access is unlimited, or *false* otherwise.

Errors: **stackunderflow, typecheck**
See Also: **rcheck, readonly, executeonly, noaccess**

where

key **where** *dict true* *(if found)*
false *(if not found)*

determines which dictionary on the dictionary stack, if any, contains an entry whose key is *key*. **where** searches for *key* in each dictionary on the dictionary stack, starting with the topmost (current) dictionary. If *key* is found in some dictionary, **where** returns that dictionary object and the boolean value *true*; otherwise, **where** simply returns *false*.

Errors: **invalidaccess, stackoverflow, stackunderflow, typecheck**
See Also: **known, load, get**

widthshow

c_x c_y *char string* **widthshow** –

paints glyphs for the characters of *string* in a manner similar to **show**; however, while doing so, it adjusts the width of each occurrence of the character *char*'s glyph by adding c_x to its *x* width and c_y to its *y* width, thus modifying the spacing

between it and the next glyph. This operator enables fitting a string of text to a specific width by adjusting the width of the glyph for a specific character, such as the space character.

char is an integer used as a character code. For a base font, *char* is simply an integer in the range 0 to 255 that is compared to successive elements of *string*. For a composite font, *char* is compared to integers computed from the character mapping algorithm (see Section 5.10.1, "Character Mapping"). The font number, *f*, and character code, *c*, that are selected by the character mapping algorithm are combined into a single integer according to the **FMapType** value of the immediate parent of the selected base font. For **FMapType** values of 4 and 5, the integer value is $(f \times 128) + c$; for all other **FMapType** values (with one exception, noted below), it is $(f \times 256) + c$.

Example

```
/Helvetica findfont 12 scalefont setfont
14 60 moveto (Normal spacing) show
14 46 moveto 6 0 8#040 (Wide word spacing) widthshow
```

Normal spacing
Wide word spacing

An exception to the above occurs for an **FMapType** value of 9 if the character mapping algorithm selects a character name or a CID rather than a character code. In this case, *char* is compared to an integer formed from the entire input character code, interpreted high-order byte first. For example, if the CMap defines the mappings

```
0 usefont                   % Following mapping uses font number 0
  1 begincidchar
    <8140> 633              % Maps two-byte code <8140> to CID 633
  endcidchar

1 usefont                   % Following mapping uses font number 1
  2 beginbfchar
    <61> /a                 % Maps one-byte code <61> to character name /a
    <40> <A9>               % Maps one-byte code <40> to character code <A9>
  endbfchar
```

and *string* is <81406140>, *char* is compared to 33308 (16#8140), 97 (16#61), and 425 ((1 × 256) + 16#A9), in sequence.

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**

See Also: **show, ashow, awidthshow, kshow, xshow, yshow, xyshow, stringwidth**

write *file int* **write** –

appends a single character to the output file *file*. The *int* operand should be an integer in the range 0 to 255 representing a character code (values outside this range are reduced modulo 256). If *file* is not a valid output file or an error is encountered in writing to the file, an **ioerror** occurs.

As is the case for all operators that write to files, the output produced by **write** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, invoking **flushfile** is required.

Errors: **invalidaccess, ioerror, stackunderflow, typecheck**
See Also: **read, writehexstring, writestring, file**

writehexstring *file string* **writehexstring** –

writes all of the characters of *string* to *file* as hexadecimal digits. For each element of *string* (an integer in the range 0 to 255), **writehexstring** appends a two-digit hexadecimal number composed of the characters 0 to 9 and a through f. The example

```
(%stdout) (w) file (abz) writehexstring
```

writes the six characters 61627a to the standard output file.

See Section 3.8.4, "Filters," for more information about ASCII-encoded, binary data representations and how to deal with them.

As is the case for all operators that write to files, the output produced by **writehexstring** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, invoking **flushfile** is required.

Errors: **invalidaccess, ioerror, stackunderflow, typecheck**
See Also: **readhexstring, write, writestring, file, filter**

writeobject *file obj tag* **writeobject** –



writes a binary object sequence to *file*. Except for taking an explicit *file* operand, **writeobject** is identical to **printobject** in all respects.

As is the case for all operators that write to files, the output produced by **writeobject** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, invoking **flushfile** is required.

Errors: **invalidaccess, ioerror, limitcheck, rangecheck, stackunderflow, typecheck, undefined**
See Also: **printobject, setobjectformat**

writestring *file string* **writestring** –

writes the characters of *string* to *file*. **writestring** does not append a newline character or interpret the value of *string*, which can contain arbitrary binary data. However, the communication channel may usurp certain control characters or impose other restrictions; see Section 3.8, "File Input and Output."

As is the case for all operators that write to files, the output produced by **writestring** may accumulate in a buffer instead of being transmitted immediately. To ensure immediate transmission, invoking **flushfile** is required.

Errors: **invalidaccess, ioerror, stackunderflow, typecheck**
See Also: **readstring, write, writehexstring, file, filter**

xcheck *any* **xcheck** *bool*

tests whether the operand has the executable or the literal attribute, returning *true* if it is executable or *false* if it is literal. This has nothing to do with the object's access attribute (for example, execute-only). See Section 3.3.2, "Attributes of Objects."

Errors: **stackunderflow**
See Also: **cvx, cvlit**

xor *bool*$_1$ *bool*$_2$ **xor** *bool*$_3$
int$_1$ *int*$_2$ **xor** *int*$_3$

returns the logical "exclusive or" of the operands if they are boolean. If the operands are integers, **xor** returns the bitwise "exclusive or" of their binary representations.

Examples

```
true true xor    ⇒  false      % A complete truth table
true false xor   ⇒  true
false true xor   ⇒  true
false false xor  ⇒  false

7 3 xor          ⇒   4
12 3 xor         ⇒  15
```

Errors: **stackunderflow, typecheck**
See Also: **or, and, not**

xshow *string numarray* **xshow** –
string numstring **xshow** –



is similar to **xyshow**; however, for each glyph shown, **xshow** extracts only one number from *numarray* or *numstring*. It uses that number as the *x* displacement and the value 0 as the *y* displacement. In all other respects, **xshow** behaves the same as **xyshow**.

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **xyshow, show**

xyshow *string numarray* **xyshow** –
string numstring **xyshow** –



paints glyphs for the characters of *string* in a manner similar to **show**. After painting each glyph, it extracts two successive numbers from the array *numarray* or the encoded number string *numstring*. These two numbers, interpreted in user space, determine the position of the origin of the next glyph relative to the origin of the glyph just shown. The first number is the *x* displacement and the second number is the *y* displacement. In other words, the two numbers override the glyph's normal width.

If *numarray* or *numstring* is exhausted before all the characters of *string* have been shown, a **rangecheck** error occurs. See Section 5.1.4, "Glyph Positioning," for further information about **xyshow**, and Section 3.14.5, "Encoded Number Strings," for an explanation of the *numstring* operand.

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **xshow, yshow, show**

yshow *string numarray* **yshow** –
string numstring **yshow** –



is similar to **xyshow**; however, for each glyph shown, **yshow** extracts only one number from *numarray* or *numstring*. It uses that number as the *y* displacement and the value 0 as the *x* displacement. In all other respects, **yshow** behaves the same as **xyshow**.

Errors: **invalidaccess, invalidfont, nocurrentpoint, rangecheck, stackunderflow, typecheck**
See Also: **xyshow, show**

APPENDIX A

LanguageLevel Feature Summary

SINCE ITS INTRODUCTION IN 1985, the PostScript language has been extended several times to incorporate new operators and other features. This appendix summarizes these changes.

Extensions are organized into major groups called *LanguageLevels*, numbered 1, 2, and 3. A PostScript interpreter that is based on a particular LanguageLevel supports all features of that LanguageLevel and lower. It may also support some additional features that are not a standard part of its base LanguageLevel.

For an introduction to the LanguageLevel concept, see Section 1.2, "Evolution of the PostScript Language." For guidelines about dealing compatibly with different LanguageLevels and extensions, see Appendix D.

The grouping of features by LanguageLevel is definitive. On the other hand, the grouping of features by other criteria, such as product version numbers, describes only how those features have been introduced as extensions in Adobe PostScript implementations. Such grouping, while of historical significance, is not part of the PostScript language definition and does not necessarily apply to all products.

A.1 LanguageLevel 3 Features

The LanguageLevel 3 features listed in this section, along with the features documented in the second edition of this book, are present in all LanguageLevel 3 interpreters.

The following LanguageLevel 3 operators are defined in **systemdict**:

cliprestore
clipsave
composefont
currentsmoothness
findcolorrendering
setsmoothness
shfill

Additionally, a number of standard LanguageLevel 3 operators are defined in procedure sets (instances of the **ProcSet** resource category), rather than in **systemdict**; see Table A.1.

TABLE A.1 LanguageLevel 3 operators defined in procedure sets

PROCEDURE SET	OPERATORS		
BitmapFontInit	addglyph		
	removeall		
	removeglyphs		
CIDInit	beginbfchar	endbfchar	StartData
	beginbfrange	endbfrange	usecmap
	begincidchar	endcidchar	usefont
	begincidrange	endcidrange	
	begincmap	endcmap	
	begincodespacerange	endcodespacerange	
	beginnotdefchar	endnotdefchar	
	beginnotdefrange	endnotdefrange	
	beginrearrangedfont	endrearrangedfont	
	beginusematrix	endusematrix	
ColorRendering	GetHalftoneName		
	GetPageDeviceName		
	GetSubstituteCRD		
FontSetInit	StartData		
Trapping	currenttrapparams		
	settrapparams		
	settrapzone		

Resource categories new in LanguageLevel 3 are listed in Table A.2, and Table A.3 lists some additional standard instances of existing resource categories.

TABLE A.2 New resource categories

TYPE OF RESOURCE	NEW CATEGORIES	
Regular resource	**CIDFont**	**InkParams**
	CMap	**Localization**
	ControlLanguage	**OutputDevice**
	FontSet	**PDL**
	HWOptions	**TrapParams**
	IdiomSet	
Implicit resource	**FunctionType**	
	ShadingType	
	TrappingType	

TABLE A.3 New resource instances

CATEGORY	NEW INSTANCES
ColorSpaceFamily	**CIEBasedDEF** **CIEBasedDEFG** **DeviceN**
Filter	**FlateDecode** **FlateEncode** **ReusableStreamDecode** Also, all encoding filters (with the exception of **NullEncode**) have become optional in LanguageLevel 3.
FMapType	9
FontType	2, 9, 10, 11, 14, 32, 42
HalftoneType	6, 9, 10, 16, 100
ImageType	3, 4
PatternType	2

Table A.4 lists new page device parameters and interpreter parameters. These are the new parameters documented in this book; see also the *PostScript Language Reference Supplement* and product-specific documentation. Note that not all products necessarily support every parameter.

TABLE A.4 New page device and interpreter parameters

TYPE OF PARAMETER	NEW PARAMETERS	
Page device parameters	**DeferredMediaSelection**	**PageOffset**
	ImageShift	**ProcessColorModel**
	InsertSheet	**RollFedMedia**
	LeadingEdge	**SeparationColorNames**
	MaxSeparations	**SeparationOrder**
	MediaClass	**Trapping**
	MediaPosition	**TrappingDetails**
	OutputDevice	**TraySwitch**
	PageDeviceName	**UseCIEColor**
	In the **InputAttributes** subdictionary:	
	InsertSheet	**MediaClass**
	In the **Policies** subdictionary:	
	PageSize (value 7)	
User parameters	**AccurateScreens**	**JobName**
	HalftoneMode	**MaxSuperScreen**
	IdiomRecognition	
System parameters	**CurSourceList**	**MaxDisplayAndSourceList**
	CurStoredScreenCache	**MaxImageBuffer**
	FactoryDefaults	**MaxSourceList**
	FontResourceDir	**MaxStoredScreenCache**
	GenericResourceDir	**PageCount**
	GenericResourcePathSep	**PrinterName**
	LicenseID	**StartupMode**

Miscellaneous language changes include the following:

- **bind** performs idiom recognition.
- **copypage** no longer preserves page contents.
- All filters have a **CloseSource** or **CloseTarget** parameter, and the **LZWEncode** and **LZWDecode** filters have additional new parameters.
- Halftone dictionaries have a **HalftoneName** entry.
- The base for an **Indexed** color space can be **Separation** or **DeviceN**.

- Overprinting is applied to colorants of a composite page, not just to separations.
- **glyphshow** accepts an integer operand.

A number of LanguageLevel 3 features were first introduced as extensions to LanguageLevel 2. The following sections list the features introduced by each of these extensions and, when possible, indicate which versions of Adobe PostScript implementations support them.

A.1.1 Version 2017 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2017 and greater:

- **HalftoneType** category instances: 9, 100 (note that these halftone types are product-dependent)
- Page device parameters: **LeadingEdge**, **MediaClass**, **RollFedMedia**, **UseCIEColor**
- Overprinting applied to colorants of a composite page, not just separations

A.1.2 Version 2016 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2016 and greater:

- **BitmapFontInit** procedure set operators: **addglyph**, **removeall**, **removeglyphs**
- **ColorSpaceFamily** category instances: **CIEBasedDEF**, **CIEBasedDEFG**
- **FontType** category instance: 32

A.1.3 Version 2015 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2015 and greater:

- Operators: **composefont**, **findcolorrendering**
- **CIDInit** and **ColorRendering** procedure set operators (see Table A.1 on page 726)

- Regular resource categories: **CIDFont**, **CMap**, **ControlLanguage**, **Localization**, **PDL**
- **FMapType** category instance: 9
- **FontType** category instances: 9, 10, 11
- **HalftoneType** category instance: 10
- **HalftoneName** entry in halftone dictionary
- Page device parameter: **PageDeviceName**
- **glyphshow** integer operand
- **GlyphDirectory** entry in Type 42 font

A.1.4 Version 2014 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2014 and greater:

- Page device parameters: **DeferredMediaSelection**, **ImageShift**, **MediaPosition**

A.1.5 Version 2013 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2013 and greater:

- **FontType** category instance: 42
- Page device parameter: **PageOffset**

A.1.6 Version 2012 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2012 and greater:

- Page device parameters: **ProcessColorModel**, **SeparationColorNames**, **SeparationOrder**

A.1.7 Version 2011 Extensions

The following LanguageLevel 3 features are present in LanguageLevel 2 implementations version 2011 and greater:

- Regular resource categories: **HWOptions**, **OutputDevice**
- **HalftoneType** category instance: 6
- Page device parameters: **InsertSheet**, **OutputDevice**, **TraySwitch**
- **Policies** subdictionary, **PageSize** value: 7
- User parameters: **AccurateScreens**, **JobName**
- System parameters:

CurSourceList
CurStoredScreenCache
FactoryDefaults
FontResourceDir
GenericResourceDir
GenericResourcePathSep
LicenseID
MaxImageBuffer
MaxSourceList
MaxStoredScreenCache
PageCount
PrinterName
StartupMode

A.2 LanguageLevel 2 Features

The following LanguageLevel 2 features, along with the LanguageLevel 1 features documented in the original edition of this book, are present in all LanguageLevel 2 interpreters:

- << … >> syntax for constructing dictionary objects
- <~ … ~> syntax for ASCII base-85 string literals
- // syntax for immediately evaluated names
- Binary encodings
- Object type packedarray

- Operators:

<<
>>
arct
colorimage
cshow
currentblackgeneration
currentcacheparams
currentcmykcolor
currentcolor
currentcolorrendering
currentcolorscreen
currentcolorspace
currentcolortransfer
currentdevparams
currentglobal
currentgstate
currenthalftone
currentobjectformat
currentoverprint
currentpacking
currentpagedevice
currentshared
currentstrokeadjust
currentsystemparams
currentundercolorremoval
currentuserparams
defineresource
defineuserobject
deletefile
execform
execuserobject
filenameforall
fileposition
filter
findencoding
findresource
gcheck
globaldict
GlobalFontDirectory
glyphshow
gstate
ineofill
infill
instroke
inueofill
inufill
inustroke
ISOLatin1Encoding
languagelevel
makepattern
packedarray
printobject
product
realtime
rectclip
rectfill
rectstroke
renamefile
resourceforall
resourcestatus
revision
rootfont
scheck
selectfont
serialnumber
setbbox
setblackgeneration
setcachedevice2
setcacheparams
setcmykcolor
setcolor
setcolorrendering
setcolorscreen
setcolorspace
setcolortranfer
setdevparams
setfileposition
setglobal

setgstate
sethalftone
setobjectformat
setoverprint
setpacking
setpagedevice
setpattern
setshared
setstrokeadjust
setsystemparams
setucacheparams
setundercolorremoval
setuserparams
setvmthreshold
shareddict
SharedFontDirectory
startjob
uappend
ucache
ucachestatus
ueofill
ufill
undef
undefinefont
undefineresource
undefineuserobject
upath
UserObject
ustroke
ustrokepath
vmreclaim
writeobject
xshow
xyshow
yshow

A number of LanguageLevel 2 features were first introduced as extensions to LanguageLevel 1. These extensions include functionality for CMYK color specification, composite fonts, file system support, and various other features. The following sections list the features introduced by each of these extensions and, when possible, indicate which Adobe PostScript implementations support them.

A.2.1 CMYK Color Extensions

The following LanguageLevel 2 operators are present in certain LanguageLevel 1 products, principally color printers and imagesetters.

colorimage
currentblackgeneration
currentcmykcolor
currentcolorscreen
currentcolortransfer
currentundercolorremoval
setblackgeneration
setcmykcolor
setcolorscreen
setcolortransfer
setundercolorremoval

A.2.2 Composite Font Extensions

The following LanguageLevel 2 operators are present in certain LanguageLevel 1 printer products that are intended to support Asian languages:

cshow
findencoding
rootfont
setcachedevice2

A.2.3 File System Operators

The following LanguageLevel 2 operators are present in LanguageLevel 1 host-based interpreters and in printer products that have disks or cartridges:

deletefile
filenameforall
fileposition
renamefile
setfileposition

A.2.4 Version 25.0 Language Additions

The following LanguageLevel 2 features and operators are present in all LanguageLevel 1 implementations version 25.0 and greater:

- // syntax for immediately evaluated names
- Object type packedarray
- Operators:

  currentcacheparams
  currentpacking
  packedarray
  setcacheparams
  setpacking

A.2.5 Miscellaneous Language Additions

The following LanguageLevel 2 operators are present in some LanguageLevel 1 products that are not identifiable by specific versions or functions. (When **product** and **revision** are not present in **systemdict**, they are present in **statusdict** instead.)

ISOLatin1Encoding
product
realtime
revision
serialnumber

This book also incorporates a large number of corrections and clarifications about LanguageLevel 2 features. These errata were documented in Adobe Technical Note #5085, *Updates to the PostScript Language Reference Manual, Second Edition*, which underwent several revisions. The changes are not listed here.

A.3 Incompatibilities

In a few instances, changes have been introduced that, strictly speaking, are not forward-compatible from one LanguageLevel to the next. This section summarizes those changes.

- If the source of text is a string instead of a file, an occurrence of a string literal enclosed in parentheses is treated specially in LanguageLevel 1: the scanner returns a substring of the original string instead of allocating a new string, and it does not recognize \ escape sequences within the string literal. In LanguageLevels 2 and 3, the scanner operates in a consistent way for all sources of text.
- The **copy** operator, when applied to dictionaries, copies the source dictionary's attribute to the destination dictionary in LanguageLevel 1; it does not do so in LanguageLevels 2 and 3.
- The **copypage** operator preserves page contents in LanguageLevels 1 and 2, but erases the page in LanguageLevel 3. The use of **copypage** was discouraged even in LanguageLevel 2.
- The encoding filters that are a standard part of LanguageLevel 2, with the exception of **NullEncode**, have become optional in LanguageLevel 3 and may not be supported in some products.

APPENDIX B

Implementation Limits

THE POSTSCRIPT LANGUAGE does not inherently restrict the sizes or quantities of things described in the language, such as numbers, arrays, stacks, paths, and so forth. However, a PostScript interpreter running on a particular processor in a particular operating environment does have such limits, and cannot execute PostScript programs that exceed those limits. Attempting to perform an operation that would exceed one of the limits causes the error **limitcheck** (or **VMerror** if the operation exhausts virtual memory resources).

Because the PostScript interpreter has been designed to handle very complex page descriptions, all implementation limits are large enough that most PostScript page descriptions should never come close to exceeding them. On the other hand, the interpreter is *not* designed for unlimited general programming. Although the PostScript language makes no formal distinction between page descriptions and general programs, a PostScript interpreter residing in a printer is specifically optimized for its intended use: to produce raster output according to a fully specified graphical description generated by some external application program. For this reason, programs that are not page descriptions may well encounter some of the implementation limits.

Occurrence of a **limitcheck** error during the execution of a page description often points to an error in the PostScript program itself, such as unbounded recursion on one of the stacks. A **VMerror** typically indicates that the program is not using the **save** and **restore** operators properly.

B.1 Typical Limits

This section describes limits that are typical of PostScript implementations from Adobe Systems. These limits fall into two main classes:

- *Architectural limits*. The hardware on which the PostScript interpreter executes imposes certain constraints; for example, integers are usually represented in 32 bits, limiting the range of integer values that are allowed. Additionally, the design of the software imposes other constraints, such as a limit of 65,535 elements in an array or string.
- *Memory limits*. The amount of memory available to the PostScript interpreter limits the number of memory-consuming objects that the interpreter can hold simultaneously. Memory management is discussed below.

Table B.1 shows the typical architectural limits for most PostScript interpreters running on 32-bit machines. Although these limits are likely to remain constant across a wide variety of platforms, they do not necessarily apply to all PostScript implementations. In particular, the limits for real numbers in any implementation are those imposed by the native floating-point representation of the underlying hardware platform. The real-number limits shown in the table are based on the IEEE 754 standard for normalized single-precision floating-point arithmetic. (See the Bibliography for a reference to this document.) Not all implementations adhere to this standard, however; see product documentation for the exact limits in a particular implementation.

Memory limits cannot be characterized so precisely, because the amount of available memory and the ways in which it is allocated vary from one product to another. Nevertheless, it is possible to give some general information about memory limits that a complex page description is likely to encounter.

The PostScript interpreter requires memory for a variety of purposes, including:

- Virtual memory for the values of composite objects
- Stacks and other objects visible to a PostScript program
- Paths in the graphics state, including those saved by the **save**, **gsave**, **gstate**, and **currentgstate** operators

- Frame buffer or other internal representation of the raster memory for the current page
- Font cache, user path cache, form cache, pattern cache, and other internal data structures that save the results of expensive computations in order to avoid redundant work

TABLE B.1 Architectural limits

QUANTITY	LIMIT	DESCRIPTION
integer	2,147,483,647	Largest integer value. This value is equal to $2^{31} - 1$, and its internal representation is 16#7FFFFFFF. In most situations, an integer that would exceed this limit is automatically converted to a real value.
	−2,147,483,648	Smallest integer value. This value is equal to -2^{31}, and its internal representation is 16#80000000.
real	$\pm 10^{38}$	Largest and smallest real values (approximate).
	$\pm 10^{-38}$	Nonzero real values closest to 0 (approximate). Values closer than these are automatically converted to 0.
	8	Significant decimal digits of precision (approximate).
array	65,535	Maximum length of an array, in elements.
dictionary	65,535	Maximum capacity of a dictionary, in entries.
string	65,535	Maximum length of a string, in characters.
name	127	Maximum length of a name, in characters.
file name	100	Maximum length of a file name, including the %*device*% prefix.
save level	15	Maximum number of active **save** operations that have not yet been matched by a corresponding **restore**.
gsave level	31	Maximum number of active **gsave** operations. Each **save** operation also performs an implicit **gsave**.
clipsave level	31	Maximum number of active **clipsave** operations within a graphics state that have not yet been matched by a corresponding **cliprestore**.
XUID array	16	Maximum number of elements in an **XUID** (extended unique ID) array.
CID	65,535	Maximum value of a CID (character identifier).
separations	250	Maximum number of separations, colorants, or color components in **DeviceN** color space.

Different LanguageLevels have somewhat different conventions for allocating available memory among these uses. In LanguageLevel 1, there is usually a static allocation for each purpose—so much memory for stacks, so much for paths, and so on. If a PostScript program exceeds these static allocations, a **limitcheck** error occurs. Installing more memory in a LanguageLevel 1 product, if possible at all, usually increases the limit on available virtual memory but seldom affects any of the other limits.

In LanguageLevels 2 and 3, the allocation of memory is much more flexible. Memory is automatically reallocated from one use to another when necessary. When more memory is needed for a particular purpose, it can be taken away from memory allocated for other purposes if that memory is currently unused or if its use is nonessential (a cache, for instance). Installing more memory in a LanguageLevel 2 or LanguageLevel 3 product causes most implementation limits to increase.

Of course, the added flexibility in LanguageLevels 2 and 3 results in a loss of predictability. If a PostScript program consumes an unusually large amount of memory for a particular purpose, it may reduce other implementation limits *below* the corresponding limits in LanguageLevel 1.

In general, it is unwise for applications to generate page descriptions that operate near the implementation limits for resources. Such page descriptions cannot reasonably be included as components of larger page descriptions, because the combined resource requirements might exceed implementation limits.

Table B.2 gives memory limits that are typical of LanguageLevel 1 implementations. These are the smallest limits that are likely to be encountered in any product; many products have larger limits for some resources. LanguageLevel 2 and LanguageLevel 3 implementations have no fixed limits, though a program can establish certain artificial limits by means described in Appendix C.

There are other implementation limits on uses of memory that are not directly under the control of a PostScript program and are difficult to quantify. For example:

- Rendering extremely complex paths requires a substantial amount of memory, particularly when the **clip** operator is invoked.
- Halftone screens occupy an amount of memory that depends on the screen angle, frequency, and device resolution. Screens saved by **gsave** may occupy additional memory.
- High-resolution devices, such as imagesetters, represent the current page as a display list on the disk instead of a full pixel array in memory. If disk space is exhausted, a **limitcheck** error occurs.

TABLE B.2 Typical memory limits in LanguageLevel 1

QUANTITY	LIMIT	DESCRIPTION
userdict	200	Capacity of **userdict**. Note that **userdict** starts out with a few entries already defined.
FontDirectory	100	Capacity of **FontDirectory**, determining the maximum number of fonts that may be defined simultaneously.
operand stack	500	Maximum depth of the operand stack: number of elements that may be pushed on and not yet popped off. This also establishes a limit on the number of elements contained in all unfinished procedure definitions being processed by the PostScript language scanner, since the scanner uses the operand stack to accumulate them.
dictionary stack	20	Maximum depth of the dictionary stack.
execution stack	250	Maximum depth of the execution stack. Each procedure, file, or string whose execution has been suspended occupies one element of this stack. Also, control operators such as **for**, **repeat**, and **stopped** push a few additional elements on the stack to control their execution.
interpreter level	10	Maximum number of recursive invocations of the PostScript interpreter. Graphics operators that call PostScript procedures, such as **pathforall**, **show**, and **image**, invoke the interpreter recursively.
path	1500	Maximum number of points specified in all active path descriptions, including the current path, the clipping path, and paths saved by **save** and **gsave**.

dash	11	Maximum number of elements in a dash pattern: the maximum length of the array operand of the **setdash** operator.
VM	240,000	Maximum size of virtual memory, in bytes. Typically, this limit is influenced by the size of the imageable area for the current page, which requires memory in proportion to its area. Thus, increasing the page size reduces the VM limit. The current and maximum size of VM are reported by the **vmstatus** operator.
file	6	Maximum number of open files, including the standard input and output files. This limit is substantially larger in implementations that support named files.
image	3300	Maximum width of an image's source data, in samples per scan line. (Most implementations have a larger limit, but it varies from product to product.)

B.2 Virtual Memory Use

It is impossible to predict accurately how much virtual memory a program will consume, but it is possible to make a rough estimate. VM is occupied primarily by the values of composite objects. Simple objects do not consume VM, nor do composite objects that share the values of other objects. Some typical memory requirements are as follows:

- Array values are created and VM consumed when a program executes the **array**, **]**, and **matrix** operators. An array value occupies 8 bytes per element.
- When the PostScript language scanner encounters a procedure delimited by { … }, it creates either an array or a packed array, according to the current packing mode (see the description of the **setpacking** operator in Chapter 8). An array value occupies 8 bytes per element. A packed array value occupies 1 to 9 bytes per element, depending on each element's type and value; a typical average is 2.5 bytes per element.
- String values are created and VM consumed when a program executes the **string** operator and when the scanner encounters string literals delimited by (…), < … >, and <~ … ~>. A string value occupies 1 byte per element.
- Dictionary values are created by the **dict** and **>>** operators and by certain other operators that return collections of parameters as dictionaries. VM consumption is based on the dictionary's capacity (its **maxlength**), regardless of how full it currently is. A dictionary value occupies about 20 bytes per entry.

- Name objects consume VM at the time the scanner first encounters each distinct name. Computed names (generated by the **cvn** operator, for instance) consume VM on their first use as names. Repeated occurrences of a particular name require no additional storage. Each distinct name occupies about 40 bytes plus the number of characters in the name.
- The **save**-**restore** machinery consumes VM in proportion to the magnitude of the changes that must be undone by **restore**, but independently of the total size of VM. **restore** reclaims all local VM resources consumed since the corresponding **save**.
- Loading a Type 1 font program typically consumes 20,000 to 30,000 bytes of VM, depending on the size of the character set and the complexity of the glyphs. VM consumption of a font remains essentially constant, regardless of the number of ways in which its glyphs are scaled, rotated, or otherwise transformed.

APPENDIX C

Interpreter Parameters

IN GENERAL, THE FACILITIES described in this appendix are available only in LanguageLevels 2 and 3. A few special-purpose operators are explicitly identified as LanguageLevel 1 operators, and some features are unique to LanguageLevel 3.

There are various parameters to control the operation and behavior of the PostScript interpreter. Most of these have to do with allocation of memory and other resources for specific purposes; for example, there are parameters to control the maximum amount of memory to be used for virtual memory, the font cache, and halftone screens. Some parameters control the behavior of specific input/output and other devices.

A PostScript product is initially configured with interpreter parameter values that are appropriate for most applications. However, with suitable authorization, a PostScript program can alter the interpreter parameters to favor certain applications or to adapt the product to special requirements.

The interpreter parameters are divided into three categories:

- *User parameters* can be altered at will (within reasonable limits) by any PostScript program without special authorization. The **setuserparams** and **currentuserparams** operators manipulate user parameters. Alterations to user parameters are subject to **save** and **restore**.
- *System parameters*, in general, can be altered only by a program that presents a valid password. The **setsystemparams** and **currentsystemparams** operators manipulate system parameters. Alterations to system parameters have a permanent, systemwide effect, which may persist through restarts of the PostScript interpreter.

- *Device parameters* are similar to system parameters but apply to individual input/output or other devices. The **setdevparams** and **currentdevparams** operators manipulate device parameters. Alterations to device parameters are permanent and systemwide, and may persist through interpreter restarts.

The operators that manipulate interpreter parameters are described in Chapter 8; this appendix describes the individual parameters. Although these operators are a standard LanguageLevel 2 feature, the exact set of interpreter parameters recognized may vary from product to product. Not all products support all parameters, and some products may support additional parameters beyond those discussed here; in addition, the set of parameters supported by a given product may be subject to change over time. The parameters described here are typical of those supported by current PostScript products from Adobe.

Most of the user parameters establish temporary policies on matters such as whether to insert new items into caches. It is reasonable for a user (or a spooler program acting on the user's behalf) to alter user parameters when submitting jobs with unusual requirements. The system parameters, on the other hand, permanently alter the overall configuration of the product. A user application should never attempt to alter system or device parameters; only system management software should do so.

C.1 Properties of User and System Parameters

A program alters user or system parameters by invoking the **setuserparams** or **setsystemparams** operator, passing it a dictionary containing the names and new values of the parameters to be changed. Each user or system parameter is identified by a key, which is always a name object. The dictionary may also contain additional information; in particular, there can be an entry named **Password**, as discussed in Section C.1.2, "System Parameters."

The dictionary passed to **setuserparams** or **setsystemparams** is similar to the request dictionary used with the **setpagedevice** operator (see Section 6.1.1, "Page Device Dictionary"). It contains entries for one or more parameters whose values are to be changed; parameters not included in the dictionary are left undisturbed. The effects of such changes are cumulative over multiple invocations of **setuserparams** or **setsystemparams**; that is, the effect of setting a particular parameter persists through subsequent invocations of the operators until explicitly overridden or until the parameters are restored to some previous state by a

restore operation. This cumulative behavior allows individual parameters to be set in a modular fashion.

The detailed semantics of user and system parameters are implementation-dependent. For example, the effects of limits on the sizes of caches depend on how cached items are represented internally. Still, there are some guidelines that apply to interpreter parameters generally, as described in the subsections below.

C.1.1 User Parameters

The **setuserparams** operator sets user parameters; **currentuserparams** reads their current values. Alterations to user parameters are subject to **save** and **restore**—that is, **restore** resets all user parameters to their values at the time of the most recent **save** operation. In an interpreter that supports multiple execution contexts, user parameters are maintained separately for each context.

Usually, altering user parameters with **setuserparams** does not affect the behavior of PostScript programs, only their performance. For example, increasing the **MinFontCompress** parameter allows larger character glyphs to be stored as full pixel arrays. This increases the speed at which those glyphs can be shown, but at the cost of using font cache memory less efficiently.

In a few cases, however, user parameters affect implementation limits, as noted in the descriptions in Section C.3, "Details of User and System Parameters." For example, the **MaxScreenItem** parameter imposes an implementation limit on the size of a halftone screen.

In general, reducing the limit on the size of an individual cached item will not disturb any items that are already in the cache, even if they are larger than the new limit.

User parameters have default values that are implementation-dependent. In some implementations, these default values are supplied by correspondingly named system parameters that can be altered with **setsystemparams**. If an unencapsulated job changes a user parameter for which there is no corresponding system parameter, the new value becomes the default value for subsequent jobs. Changes made to a user parameter by an encapsulated job have no effect on the default value of that parameter for subsequent jobs.

C.1.2 System Parameters

The **setsystemparams** operator sets system parameters; **currentsystemparams** reads their current values. In general, permission to alter system parameters is controlled by a password: the dictionary passed to **setsystemparams** must contain an entry named **Password** whose value is equal to the system parameter password. The value supplied for **Password** may be a string or an integer; if an integer, it is converted to a string as if by the **cvs** operator. If the password supplied is incorrect, the operation is not allowed.

There are three circumstances under which **setsystemparams** does not require the dictionary to contain the system parameter password (and will ignore it if one is supplied):

- If **FactoryDefaults** is the only entry in the dictionary (or if a password is the only other entry)
- If the system parameter password has been set to the empty string
- During a system administrator job (as described in Section C.3.1, "Passwords")

Some system parameters can be set permanently—that is, in nonvolatile storage that survives restarts of the PostScript interpreter. This capability is implementation-dependent. No error occurs if parameters cannot be stored permanently. In some implementations, permanent parameter changes do not take effect until the next restart of the PostScript interpreter.

System parameters are global to the PostScript environment and are not maintained separately for each context in an interpreter that support multiple contexts. The initial value of system parameters when the device is turned on for the first time depends on the product implementation.

In general, the cache size parameters (for example, **MaxFontCache**) are simply limits; they do not represent memory dedicated to a specific use. Caches compete with each other for available memory. The main purpose of the limits is to prevent excessive memory from being devoted to one use, to the exclusion of other uses. Under some circumstances, memory in use by a cache may be unavailable for satisfying the needs of a PostScript program—for instance, to allocate new objects in virtual memory or to enlarge a stack.

Usually, reducing the size of a cache causes cached items to be discarded to make current consumption less than the new maximum. Sometimes, for implementation reasons, this operation must be deferred. Consequently, the current consumption for a cache may temporarily exceed the maximum.

Certain system parameters are read-only—that is, they are returned by **currentsystemparams**, but attempting to change one with **setsystemparams** has no effect. The read-only parameters report information such as current memory consumption. Certain other parameters, specifically **SystemParamsPassword** and **StartJobPassword**, are write-only; they can be set by **setsystemparams** but are not returned by **currentsystemparams**.

C.2 Defined User and System Parameters

The following tables summarize the commonly defined user and system parameters; for more details, see Section C.3, "Details of User and System Parameters." Additional parameters are described in the *PostScript Language Reference Supplement* and in product-specific documentation.

TABLE C.1 User parameters

KEY	TYPE	VALUE
AccurateScreens	boolean	A flag specifying whether to invoke an extremely precise (but computationally expensive) halftone algorithm during subsequent executions of **setscreen** and **setcolorscreen**.
HalftoneMode	integer	*(LanguageLevel 3)* A code controlling the behavior of subsequent halftone-setting operators: 0 Operators will behave as usual. 1 Operators may substitute a product-specific halftone. 2 Same as 1; in addition, if a product-specific halftone is substituted, further optimization may occur.
IdiomRecognition	boolean	*(LanguageLevel 3)* A flag specifying whether to enable procedure substitution during execution of the **bind** operator (see Section 3.12.1, "bind Operator").

JobName	string	The name of the current job. Status messages displayed during the remainder of the job may include a field consisting of the text of this string. Consequently, the string should consist of characters that can usefully be displayed as text; what this means depends on the product and environment, taking into account the language for which the product has been localized. The string should not contain a right bracket (]) or a semicolon (;), because these characters conflict with a very common syntax for presenting status messages. The string may be up to 100 characters long; if it exceeds this length, it is truncated.
MaxDictStack	integer	The maximum number of elements in the dictionary stack.
MaxExecStack	integer	The maximum number of elements in the execution stack.
MaxFontItem	integer	The maximum number of bytes occupied by the pixel array of a single glyph in the font cache.
MaxFormItem	integer	The maximum number of bytes occupied by a single cached form.
MaxLocalVM	integer	The maximum number of bytes occupied by values in local VM.
MaxOpStack	integer	The maximum number of elements in the operand stack.
MaxPatternItem	integer	The maximum number of bytes occupied by a single cached pattern.
MaxScreenItem	integer	The maximum number of bytes occupied by a single halftone screen.
MaxSuperScreen	integer	*(LanguageLevel 3)* The maximum number of pixels in a supercell. A value of 0 disables the use of supercells.
MaxUPathItem	integer	The maximum number of bytes occupied by a single cached user path.
MinFontCompress	integer	The threshold (in bytes) at which a cached glyph is stored in compressed form rather than as a full pixel array.
VMReclaim	integer	A code controlling the behavior of the garbage collector: 0 Enables automatic garbage collection −1 Disables it for local VM −2 Disables it for both local and global VM
VMThreshold	integer	The frequency of automatic garbage collection, which is triggered whenever this many bytes have been allocated since the previous collection.

TABLE C.2 System parameters

KEY	TYPE	VALUE
ByteOrder	boolean	*(Read-only)* The native (preferred) order of multiple-byte numbers in binary-encoded tokens. *false* indicates high-order byte first; *true* indicates low-order byte first.
BuildTime	integer	*(Read-only)* A time stamp identifying a specific build of the PostScript interpreter.
CurDisplayList	integer	*(Read-only)* The number of bytes currently occupied by display lists. A display list is an internal representation of the marks that have been painted on the current page or previous pages but have not yet been scan-converted into raster memory.
CurFontCache	integer	*(Read-only)* The number of bytes currently occupied by the font cache.
CurFormCache	integer	*(Read-only)* The number of bytes currently occupied by the form cache.
CurOutlineCache	integer	*(Read-only)* The number of bytes currently occupied by cached glyph descriptions for fonts whose definitions are kept on disk rather than in VM.
CurPatternCache	integer	*(Read-only)* The number of bytes currently occupied by the pattern cache.
CurScreenStorage	integer	*(Read-only)* The number of bytes currently occupied by all active halftone screens.
CurSourceList	integer	*(Read-only)* The number of bytes currently occupied by source lists. A source list holds the internal data representation of source data for sampled images and pixel arrays for uncached glyphs.
CurStoredScreenCache	integer	*(Read-only)* The number of bytes occupied by halftone screens that are cached on disk (or some other storage device).
CurUPathCache	integer	*(Read-only)* The number of bytes currently occupied by the user path cache.
FactoryDefaults	boolean	A flag that, if set to *true* immediately before the printer is turned off, causes all nonvolatile parameters to revert to their factory default values at the next power-on. The set of nonvolatile parameters is product-dependent. In most products, **PageCount** cannot be reset. If the job that sets **FactoryDefaults** to *true* is not the last job executed before power-off, the request is ignored; this reduces the chance that malicious jobs will attempt to perform this operation.

FontResourceDir	string	The pathname of the directory in which external font resources are located.
GenericResourceDir	string	The pathname of the directory in which generic external resources are located.
GenericResourcePathSep	string	The separator character in pathnames; used in conjunction with **GenericResourceDir** to determine where generic external resources are located.
LicenseID	string	*(Read-only)* An Adobe-assigned license identifier whose value is unique to each product.
MaxDisplayAndSourceList	integer	The maximum number of bytes occupied by display lists and source lists combined. This value should be greater than or equal to **MaxDisplayList** or **MaxSourceList** (whichever is larger).
MaxDisplayList	integer	The maximum number of bytes occupied by display lists, excluding those held in caches.
MaxFontCache	integer	The maximum number of bytes occupied by the font cache.
MaxFormCache	integer	The maximum number of bytes occupied by the form cache.
MaxImageBuffer	integer	The maximum number of bytes occupied by a single image buffer. An image buffer holds an internal data representation of the source data for a sampled image.
MaxOutlineCache	integer	The maximum number of bytes occupied by cached glyph descriptions for fonts whose definitions are kept on disk rather than in VM.
MaxPatternCache	integer	The maximum number of bytes occupied by the pattern cache.
MaxScreenStorage	integer	The maximum number of bytes occupied by all active halftone screens, including ones created by **setscreen** or **setcolorscreen** and saved by **gsave**, **gstate**, and **currentgstate**.
MaxSourceList	integer	The maximum number of bytes occupied by source lists.
MaxStoredScreenCache	integer	The maximum number of bytes occupied by halftone screens that are cached on disk (or some other storage device). Supplying a value that is negative or too large sets the maximum to the logical size of the device, or to an implementation-dependent value if the logical size is not known.
MaxUPathCache	integer	The maximum number of bytes occupied by the user path cache.
PageCount	integer	*(Read-only)* The number of pages that have been successfully processed since manufacture, counting the number of copies for each **showpage** (as specified by the value of **#copies** in the current dictionary stack or by the LanguageLevel 2 page device parameter

		NumCopies). Even pages not physically printed—because of manual feed timeout, job abort, or any other reason—are included in the count. The count will be accurate at the end of each job, but not necessarily after each **showpage**.
PrinterName	string	The name of the output device. The value of this parameter is often used as part of the name by which the device is identified on the network it is attached to. Consequently, it should conform to whatever syntax is appropriate for such names on the network. Setting **PrinterName** to a zero-length string causes the name to be set to the value of the **product** string in **systemdict**.
RealFormat	string	*(Read-only)* The native (preferred) representation for real numbers in binary-encoded tokens. This is either IEEE or the name of some specific machine architecture. The interpreter will always accept IEEE format, but it may process native-format real numbers more efficiently (see Section 3.14.4, "Number Representations").
Revision	integer	*(Read-only)* The current revision level of the product in which the PostScript interpreter is running; equal to the value of the **revision** entry in **systemdict**.
StartJobPassword	string	*(Write-only)* A password authorizing use of the **startjob** or **exitserver** operator.
StartupMode	integer	A code controlling execution of startup files during subsequent restarts of the interpreter. A startup file is a PostScript program that is invoked as an unencapsulated job; its purpose is to load initial definitions into VM. 0 Do not execute any startup file. 1 Execute the standard startup file, whose name is usually Sys/Start in any searchable file system. >1 Perform some other product-dependent startup action.
SystemParamsPassword	string	*(Read-only)* The system parameter password, which authorizes the use of the **setsystemparams** and **setdevparams** operators, as well as of the **startjob** and **exitserver** operators.

C.3 Details of User and System Parameters

The following sections give further details about user and system parameters.

C.3.1 Passwords

The password that controls the ability to change system parameters is itself a system parameter, **SystemParamsPassword**, which can be changed by **setsystemparams**. Another password, **StartJobPassword**, controls the ability to execute the **startjob** operator to alter the initial VM (see Section 3.7.7, "Job Execution Environment"). The two passwords are separate so that the system administrator can be permissive about granting access to **startjob** without compromising control over **setsystemparams**. On the other hand, should an overlap of **startjob** and **setsystemparams** permissions be desired, **SystemParamsPassword** rather than **StartJobPassword** may be passed to **startjob** to start a job that can invoke **setsystemparams** without presenting a password each time.

Note: *All references here to* ***startjob****, a LanguageLevel 2 operator, apply equally to the LanguageLevel 1* ***exitserver*** *operator.*

Which password is presented to **startjob** determines the type of unencapsulated job that is started. If the password is equal to **SystemParamsPassword**, a system administrator job is started; otherwise, if the password is equal to **StartJobPassword**, an ordinary unencapsulated job is started. A system administrator job not only may alter VM but may invoke **setsystemparams** (and **setdevparams**) without presenting a password each time. Also, LanguageLevel 1 compatibility operators that change system and device parameters may be executed during a system administrator job; see the *PostScript Language Reference Supplement*.

If an integer appears where a password is expected, it is automatically converted to a string, as if by the **cvs** operator. All characters of a password are significant, and password comparison is case-sensitive.

If a password is set to the empty (zero-length) string, password checking is disabled. If **SystemParamsPassword** has been set to the empty string, **setsystemparams** is always allowed, regardless of the **Password** value passed to it; furthermore, **startjob** is always allowed, regardless of the password presented to it, and it starts a system administrator job. Similarly, if **StartJobPassword** has been set to the empty string, **startjob** is always allowed, and it starts an ordinary unencapsulated job (unless **SystemParamsPassword** is also the empty string). When a PostScript interpreter is initially installed, both passwords are set to empty strings.

To change **SystemParamsPassword**, execute the following PostScript code:

```
<<    /Password (oldpassword)
      /SystemParamsPassword (newpassword)
>> setsystemparams
```

If the system parameter password is forgotten, there is still a way to reset it (along with other factory defaults): by passing a dictionary to **setsystemparams** in which **FactoryDefaults** is the only entry.

C.3.2 Font Cache

Two user parameters specify policies for inserting new items into the font cache. These parameters, **MaxFontItem** and **MinFontCompress**, control the behavior of the **setcachedevice** operator.

If the pixel array of a cached glyph, as determined from the bounding box passed to **setcachedevice**, would be larger than the size specified by **MaxFontItem**, the glyph will not be cached. If a glyph's pixel array is not larger than the size specified by **MaxFontItem**, the glyph will be cached (space permitting), as follows: If the pixel array is larger than or equal to the size specified by **MinFontCompress**, the glyph will be stored in a space-efficient, compressed representation in the cache; otherwise, it will be stored in a time-efficient, full-pixel-array representation. Compressed glyphs consume much less space in the font cache than do full pixel arrays (by factors of up to 40 in Adobe implementations), but they require more computation because they need to be reconstituted from the compressed representation each time they are needed. **MinFontCompress** controls the tradeoff between time and space. Some devices do not support compression of glyphs.

Setting **MinFontCompress** to 0 forces all glyphs to be compressed, permitting more glyphs to be stored in the cache but increasing the work required to print them. Setting it to a value greater than **MaxFontItem** disables compression altogether.

There are three convenience operators that control the same font cache parameters: **setcachelimit**, **setcacheparams**, and **currentcacheparams**. **setcachelimit** exists in all LanguageLevel 1 implementations; **setcacheparams** and **currentcacheparams** exist in most, but not all. The LanguageLevel 1 **cachestatus** opera-

tor returns some implementation-dependent information in addition to what is available from **currentsystemparams**.

The system parameter **MaxFontCache** specifies an overall limit on the size of the font cache, including both the device pixel arrays themselves and other overhead, such as cached metrics.

C.3.3 Other Caches

User paths, forms, and patterns all use caches that are controlled in similar ways. The user parameters **MaxUPathItem**, **MaxFormItem**, and **MaxPatternItem** specify limits on the sizes of individual items to be inserted into the respective caches. The system parameters **MaxUPathCache**, **MaxFormCache**, and **MaxPatternCache** specify overall limits on the sizes of the caches.

Two convenience operators, **setucacheparams** and **ucachestatus**, also deal with the user path cache parameters. These operators exist for compatibility with earlier PostScript implementations.

C.3.4 Halftone Screens

Certain parameters affect the storage for halftone screens and the behavior of subsequent halftone-setting operators.

Storage for Halftone Screens

Storage for halftone screens is managed somewhat differently than storage for caches. The halftone machinery must have enough storage to hold an expanded internal representation of the screen in use. It can use any excess storage to hold a cache of screens that are not in use.

The user parameter **MaxScreenItem** specifies the maximum amount of memory a single halftone screen can occupy. This is not a simple function of the size of a halftone cell; it is influenced by frequency, angle, device resolution, and quantization of raster memory. The **MaxScreenItem** parameter imposes an implementation limit on the size of screens that can be used. Furthermore, the user parameter **MaxSuperScreen** *(LanguageLevel 3)* specifies the maximum number of pixels in a supercell (see Section 7.4.8, “Supercells”).

Use of the **AccurateScreens** feature of halftone dictionaries (or the user parameter of the same name) substantially increases the memory requirement for a halftone screen. The highest accuracy is achieved only when sufficient memory is available. As a rule of thumb, **MaxScreenItem** should be at least $R \times D \times 5$, where R is the device resolution in pixels per inch and D is the diagonal length of the imageable area of device space in inches.

The system parameter **MaxScreenStorage** specifies an overall limit on the amount of memory for all active halftone screens. A screen is active if it is the current screen, if it has been saved on the graphics state stack or in a gstate object, or if marks were placed on the page while the screen was the current screen.

In systems with disk-based screens, **MaxScreenStorage** specifies how much memory is available to build and hold one screen. Since any other screens (those described above) will have been saved on disk, they do not consume memory and are thus ignored when the current screen storage amount is calculated.

In some implementations, internal representations of halftone screens are copied to files on disk (or some other storage device) for more efficient setup if the same screens are used again later. The system parameter **MaxStoredScreenCache** limits the amount of space that will be used for this purpose.

Halftone Setting

Two user parameters, **AccurateScreens** and **HalftoneMode**, control the behavior of subsequent halftone-setting operators.

AccurateScreens corresponds to the similarly named feature of halftone dictionaries, but applies only to **setscreen** and **setcolorscreen**; it does not affect the operation of **sethalftone**. If this parameter is *true*, an extremely precise (but computationally expensive) halftoning algorithm is enabled; see "Type 1 Halftone Dictionaries" on page 487 for details.

HalftoneMode *(LanguageLevel 3)* controls the behavior of subsequent executions of **setscreen**, **setcolorscreen**, and **sethalftone**. A nonzero value may alter the behavior of these operators as follows:

- A **HalftoneMode** of 1 may cause the operator to ignore its halftone operand and substitute a product-specific halftone. Whether this substitution actually

occurs is product-dependent. When a product-specific halftone is substituted, certain pages may print faster; this behavior is also product-dependent.

- A **HalftoneMode** of 2 has the same effect as a **HalftoneMode** of 1; in addition, if a product-specific halftone is substituted, further optimization may occur during image rendering. This optimization, which results in additional speed improvement at the expense of some degradation in image quality, is disabled for masked images, image masks, and images rotated at angles other than multiples of 90 degrees.

Halftone substitution is strictly internal to the interpreter; its effects are not visible at the PostScript language level. If halftone substitution occurs during a **sethalftone** operation, for example, a subsequent **currenthalftone** operation will return the halftone dictionary that was originally supplied as an operand to **sethalftone**, not the substituted halftone.

Note that **HalftoneMode** has no effect on the current halftone.

C.3.5 Virtual Memory and Stacks

The **MaxLocalVM** user parameter imposes a limit on the total amount of local virtual memory in use. Attempting to create a new composite object in local VM will fail (with a **VMerror**) if the VM would exceed its limit. There is no corresponding limit for global VM. The method for sharing global VM among multiple execution contexts does not provide a way to attribute VM consumption to a particular context.

Three other user parameters, **MaxOpStack**, **MaxDictStack**, and **MaxExecStack**, impose limits on the number of elements that can be pushed onto the operand, dictionary, and execution stacks. Attempting to exceed one of these limits will result in a **stackoverflow**, **dictstackoverflow**, or **execstackoverflow** error, respectively.

Normally, there are no effective limits on VM or stack allocation; that is, the default values of these user parameters are extremely large. VM and stacks can grow without limit, subject only to the total amount of memory available in the machine on which the PostScript interpreter is running. As VM and stack consumption increases, less memory is available for the font cache and other uses; this can degrade performance. The main use of the VM and stack limit parameters is to test the behavior of applications in limited memory.

Two user parameters, **VMReclaim** and **VMThreshold**, control the behavior of the garbage collector. Normally, garbage collection is triggered periodically and automatically to reclaim inaccessible objects in VM. It is sometimes useful to disable garbage collection temporarily—to obtain repeatable timing measurements, for instance.

Like all user parameters, the VM and stack parameters are maintained separately for each context in an interpreter that supports multiple execution contexts. In particular, if VM is shared among multiple contexts, the effects of a particular context's VM parameters apply only while that context is executing.

The **vmstatus**, **vmreclaim**, and **setvmthreshold** operators manipulate some of the VM parameters. **vmreclaim** can also be used to trigger immediate garbage collection.

C.3.6 Resource File Location

Resource instances are typically installed as named files, as discussed in Section 3.9.4, "Resources as Files." Three system parameters—**GenericResourceDir**, **GenericResourcePathSep**, and **FontResourceDir**—determine where files containing external resources are located under certain conditions.

GenericResourceDir and **GenericResourcePathSep** control the location of resources for the **Generic** resource category and all categories based on it (typically all regular resource categories except **Font**). The implementation of the **Generic** category concatenates the value of **GenericResourceDir** with the category name, the value of **GenericResourcePathSep**, and the resource name to get the external location of the resource. If, for example, **GenericResourceDir** and **GenericResourcePathSep** were Resource/ and /, respectively, the **AdobeLogo** resource of the **Pattern** category would be in Resource/Pattern/AdobeLogo.

GenericResourceDir must be an absolute pathname—that is, a pathname beginning at the root of the storage device. It must end with a pathname separator, as defined by **GenericResourcePathSep**. It should include a storage device (for example, %os%) if a single device is to be considered, or omit the device if all searchable devices are to be considered. If a device is dedicated to generically managed resources (for example, %GenericResource%) and may access resources through a network server or along a search path, **GenericResourceDir** should be set to that device.

Resource files are expected to be in subdirectories with names corresponding to category names. The resource file name should be the same as the name of the resource it defines. In the example above, the file named Resource/Pattern/AdobeLogo should contain a PostScript program that, when run, will define the **AdobeLogo** instance of the **Pattern** resource category.

For products with no external resources (and presumably no file systems), **GenericResourceDir** should be set to %null. This mechanism can also be used by site administrators to temporarily disable access to external resources.

FontResourceDir controls the location of external fonts, which are resources in LanguageLevels 2 and 3. The implementation of the **Font** resource category concatenates the value of **FontResourceDir** with the font name to get the external location of the font. For example, if **FontResourceDir** were Resource/Font/, the **Times-Roman** resource of the **Font** category would be in Resource/Font/Times-Roman. This parameter is provided separately from **GenericResourceDir** to allow backward compatibility with applications that expect fonts to be located under fonts/, in which case **FontResourceDir** should be set to fonts/.

C.4 Device Parameters

Each PostScript interpreter supports a collection of input/output and other devices, such as communication channels, disks, and cartridges. The standard file operators, described in Section 3.8, "File Input and Output," access these devices as files. Some devices have device-dependent parameters. In addition, there are some named sets of parameters that do not correspond to physical devices. Given a string identifying a device or other named parameter set, the **setdevparams** and **currentdevparams** operators access these parameters.

A device is identified by a string of the form %*device*, or %*device*%, which is a prefix of the %*device*%*file* syntax for named files in storage devices (see Section 3.8.2, "Named Files"). The available devices can be enumerated by invoking the **resourceforall** operator for the **IODevice** category (see Section 3.9, "Named Resources").

setdevparams is very similar to **setsystemparams**; the same restrictions apply. The names of parameter sets and the names and semantics of the parameters are product-dependent. They are not documented in this book, but rather in the *PostScript Language Reference Supplement* and in product-specific documentation.

APPENDIX D

Compatibility Strategies

AS DISCUSSED IN SECTION 1.2, "Evolution of the PostScript Language," the PostScript language has undergone several significant extensions in order to adapt to new technology and to incorporate new functionality and flexibility. While the PostScript language is designed to be a universal standard for device-independent page description, the reality is that there are different PostScript language implementations that have different sets of features. This appendix presents guidelines for taking advantage of language extensions while maintaining compatibility with all PostScript interpreters.

D.1 The LanguageLevel Approach

PostScript implementations are organized into *LanguageLevels*, of which three have been defined:

- LanguageLevel 1 interpreters implement all LanguageLevel 1 features, These features are documented in the first edition of this book. In the present (third) edition, LanguageLevel 1 consists of all features except those explicitly designated as LanguageLevel 2 or LanguageLevel 3.
- LanguageLevel 2 interpreters implement all LanguageLevel 1 and LanguageLevel 2 features. These features are documented in the second edition of this book. In the present edition, LanguageLevel 2 consists of all features except those explicitly designated as LanguageLevel 3.
- LanguageLevel 3 interpreters implement all features of all LanguageLevels. Those features that are available only in LanguageLevel 3 are summarized in Section 1.2, "Evolution of the PostScript Language."

Except as noted in Section A.3, "Incompatibilities," each LanguageLevel is forward-compatible with higher LanguageLevels. For example, applications that work with LanguageLevel 2 interpreters, using the language features documented in the second edition of this book, will also work with LanguageLevel 3 interpreters. Higher LanguageLevels are not, however, backward-compatible with lower LanguageLevels. Thus, PostScript programs that use LanguageLevel 3 features do not automatically work on LanguageLevel 2 or LanguageLevel 1 interpreters. Applications wishing to take advantage of new LanguageLevel features while remaining compatible with lower LanguageLevels must adopt one or more of the strategies described in Section D.3, "Compatibility Techniques."

In addition to the three standard LanguageLevels, there are several language *extensions*. An extension is a collection of language features that are not a standard part of the LanguageLevel supported by an implementation. For example, if a LanguageLevel 1 implementation includes CMYK color features, those features are an extension, since CMYK color is not a standard part of LanguageLevel 1. On the other hand, *all* LanguageLevel 2 (and LanguageLevel 3) implementations include CMYK color features, since CMYK color is a standard part of LanguageLevel 2.

Extensions exist because the PostScript language must evolve to support new technologies and new applications. When an extension is introduced, it is based on an existing LanguageLevel. Extensions that prove to be of general utility are candidates for inclusion as standard features in the next higher LanguageLevel. For example, many LanguageLevel 3 features originated as extensions to LanguageLevel 2. Appendix A describes how these extensions are organized.

The advantages of the LanguageLevel approach are clear. Organizing features into a small number of levels simplifies the choices that application software developers must make. In contrast, organizing them as independent extensions implemented in arbitrary combinations leads to an exponential increase in choices. An application using features of a given LanguageLevel is guaranteed to work with PostScript interpreters at that LanguageLevel or higher.

Although the LanguageLevel approach simplifies application programming, it is sometimes necessary for applications to depend on specific extensions for functional reasons. The following sections emphasize techniques for creating applications that are compatible with interpreters of different LanguageLevels, but many of these techniques are applicable when dealing with extensions as well.

D.2 When to Provide Compatibility

An application (or driver) must know what PostScript operators are available to it. Essentially, there are two different scenarios:

- The application is sending output to a specific PostScript interpreter, in which case it knows what the target interpreter is.
- The application is printing through a spooler or saving to a file, in which case it does not know what the target interpreter is.

In the first case, the application can generate a PostScript program appropriate for the target interpreter. The application simply needs to determine the LanguageLevel that the interpreter supports *before* generating the PostScript page description. There are two ways to do this:

- Consult a PPD file.
- Query the interpreter directly.

A *PostScript printer description* (PPD) file is a text file that can be read by an application to obtain information about a specific printer product. In the PPD file, the *LanguageLevel entry specifies the PostScript LanguageLevel that the product supports. (If the entry is absent, the product supports LanguageLevel 1.) For information on PPD files, refer to Adobe Technical Note #5003, *PostScript Printer Description File Format Specification*.

If there is a bidirectional communication channel between the application and the PostScript interpreter, the application can determine the interpreter's capabilities by sending it a query job. The following program queries the LanguageLevel of the interpreter:

```
%!PS-Adobe-3.0 Query
%%?BeginFeatureQuery: *LanguageLevel
    /languagelevel where
            {pop languagelevel}
            {1}
        ifelse
    (') print 3 string cvs print (') = flush
%%?EndFeatureQuery: Unknown
```

This query job returns an integer enclosed in double quotes (following PPD conventions for identifying features). The integer identifies the highest LanguageLevel supported by the interpreter. Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*, presents guidelines for constructing query jobs.

Checking for the existence of language extensions that are not part of a particular LanguageLevel is very similar. For example, some LanguageLevel 1 implementations have the CMYK color extension; if the application wants to use the CMYK color operators, it needs to find out whether the target interpreter supports them. This, too, can be tested either by consulting the product's PPD file (specifically, the *Extensions entry) or by sending a query job to the interpreter. For example:

```
%!PS-Adobe-3.0 Query
%%?BeginQuery: ColorExtensions
    /setcmykcolor where
            {pop true}
            {false}
        ifelse
    = flush
%%?EndQuery: false
```

This query job returns either *true* or *false,* indicating whether the **setcmykcolor** operator is available.

If an application is producing output not targeted to a particular interpreter, the strategy is entirely different. The application has the following options:

- Generate a page description using LanguageLevel 1 features only. The resulting program can be sent to any interpreter.

  This is the simplest method for producing fully portable output, and is entirely adequate for many applications. However, it sacrifices any improvements in functionality, performance, or programming convenience available through the use of LanguageLevel 2 or LanguageLevel 3 features.

- Generate a page description that uses LanguageLevel 2 or LanguageLevel 3 features unconditionally. The resulting program will execute correctly only when sent to an interpreter that supports the highest LanguageLevel of the features that the program uses.

This approach allows the application to take full advantage of higher-LanguageLevel features, but at the cost of incompatibility with lower-LanguageLevel interpreters. This makes the most sense when an application *must* use newer features to perform functions that are simply unavailable in lower LanguageLevels, such as device-independent CIE-based color specification in LanguageLevel 2. Especially in this case, the application should include the appropriate document structuring comments (as described in Technical Note #5001), so that a print manager or spooler can know that it must direct the page description to an interpreter supporting the required LanguageLevel.

- Generate a page description that uses higher-LanguageLevel features but provides for compatibility with lower-LanguageLevel interpreters. The resulting program can be sent to any interpreter that supports at least the lower LanguageLevel.

  This is the most desirable option, because the resulting page description is portable yet takes advantage of higher-LanguageLevel features when they are available. The idea behind this strategy is for the application to provide PostScript emulations, using lower-LanguageLevel features, of the higher-LanguageLevel features that the page description actually uses. When the program is executed, it determines which features the interpreter supports and installs the emulations only if necessary (see Section D.4, "Installing Emulations"). This strategy may not be the simplest or most efficient, but it takes best advantage of the features available in different interpreters.

D.3 Compatibility Techniques

It is not possible to emulate every feature of higher LanguageLevels in terms of lower-LanguageLevel features, but many features can be at least partially emulated. For example, LanguageLevel 2 user path operators with ordinary user paths as operands can be emulated easily in LanguageLevel 1, but those with encoded user paths as operands can be emulated only with great difficulty and probably with unacceptable cost in performance. The application must determine an appropriate tradeoff between the benefit of using a feature and the cost of providing emulation for that feature.

The following sections outline three compatibility techniques: complete emulation, partial emulation, and emulation in the application's PostScript driver.

D.3.1 Complete Emulation

Some features of higher LanguageLevels are sufficiently simple that they can be completely emulated in terms of lower-LanguageLevel features. For instance, the LanguageLevel 2 operator **selectfont** is defined as follows:

key scale **selectfont**
key matrix **selectfont**

This operator obtains a font whose name is *key,* transforms it according to *scale* or *matrix,* and establishes it as the current font dictionary in the graphics state. This is equivalent to executing **findfont**, **scalefont** (or **makefont**), and **setfont**. But **selectfont** is more than just a convenience operator: its implementation caches the scaled font dictionary for possible reuse, making it more efficient as well. Using it can significantly improve the performance of programs that switch fonts frequently.

The **selectfont** operator can be completely emulated in terms of LanguageLevel 1 features as follows:

```
/selectfont
   {   exch findfont exch
       dup type /arraytype eq
               {makefont}
               {scalefont}
           ifelse
       setfont
   } bind def
```

A program can then invoke this emulation if the **selectfont** operator is unavailable. Section D.4, "Installing Emulations," describes the recommended method for accomplishing this.

Note that this emulation of **selectfont** does not achieve the performance gain that the actual **selectfont** operator does. Although it is possible to write a PostScript emulation of **selectfont** that caches scaled font dictionaries, this is tricky and probably not worthwhile.

D.3.2 Partial Emulation

Not all forms of certain operators can be emulated efficiently. For example, the **rectfill** operator is defined as follows:

x y width height **rectfill**
numarray **rectfill**
numstring **rectfill**

It is straightforward to emulate the first form of **rectfill**, and, with a little more work, the *numarray* form as well. However, it is difficult to emulate the *numstring* form efficiently in terms of LanguageLevel 1 features. For this reason, Adobe recommends that applications avoid using the *numstring* form when compatibility with LanguageLevel 1 interpreters is required.

Note that an application can choose to eliminate unnecessary overhead by emulating only the form of an operator it actually uses. Example D.1 defines a procedure named *RF that is a partial emulation of the **rectfill** operator. (The reason for naming this emulation *RF and not rectfill is explained in Section D.4, "Installing Emulations.")

Example D.1

```
/BuildRectPath
   {  dup type dup /integertype eq exch /realtype eq or
            {  4 –2 roll moveto                    % Operands are: x y width height
               dup 0 exch rlineto
               exch 0 rlineto
               neg 0 exch rlineto
               closepath
            }
            {  dup length 4 sub 0 exch 4 exch      % Operand is: numarray
                     {  1 index exch 4 getinterval aload pop
                        BuildRectPath
                     }
                  for
               pop
            }
         ifelse
   } bind def
```

```
/*RF
   {  gsave
         newpath BuildRectPath fill
      grestore
   } bind def
```

This emulation, in addition to omitting the *numstring* case altogether, does not emulate the *numarray* case precisely. **rectfill** draws all rectangles counterclockwise in user space, whereas *RF draws a rectangle clockwise if its height or width is negative. This affects the insideness computation (see Section 4.5.3, "Insideness Testing") if the rectangles overlap.

D.3.3 Emulation in the Driver

When a LanguageLevel 2 or LanguageLevel 3 feature is too costly to emulate in terms of lower-LanguageLevel features, the alternative is to avoid using that feature at all but to redesign the application's PostScript driver to obtain the same effect in a more efficient way. This often requires the application to do more work, such as keeping track of information that a higher-LanguageLevel interpreter would maintain automatically.

For example, instead of using the **selectfont** operator, a driver can keep track of scaled font dictionaries it has referenced recently. When it detects that a given font dictionary is needed multiple times, it can generate PostScript commands to save the dictionary on first use and refer to the saved dictionary on later uses. This achieves approximately the same performance benefits as using **selectfont**, but at the cost of additional complexity in the driver.

A related technique is to implement a desired effect in different ways, depending on the LanguageLevel that the interpreter supports. For example, when drawing thin strokes, either of two different methods can be used, depending on the LanguageLevel, to ensure that the strokes will be of uniform thickness when rasterized (see Section 7.5.2, "Automatic Stroke Adjustment"):

- When running on a LanguageLevel 2 or LanguageLevel 3 interpreter, all that is necessary is to invoke the LanguageLevel 2 operator **setstrokeadjust** to enable automatic adjustment before drawing any strokes.
- When running on a LanguageLevel 1 interpreter, a similar effect can be achieved by using the **transform**, **round**, and **itransform** operators to "snap" all

coordinates explicitly to pixel boundaries in device space before presenting them to the path construction operators.

The prolog should define procedures that select one or the other of these methods for drawing strokes. The most efficient way to do this is to include two or more different sets of procedures in the prolog, but conditionally define only one of them, depending on the LanguageLevel of the interpreter on which the program is running.

D.3.4 Syntax Considerations

LanguageLevel 2 includes additions to the syntax of the PostScript language as well as to the set of available operators. These additions are:

- The << … >> notation for constructing dictionary objects
- The <~ … ~> notation for ASCII base-85 string literals
- The // notation for immediately evaluated names
- Binary encodings

Since these constructs are part of the syntax, they are parsed by the scanner, whether or not they are ever executed. If a program containing these constructs is sent to a LanguageLevel 1 interpreter, a **syntaxerror** will occur, even if the constructs appear only inside procedures that are to be executed conditionally according to LanguageLevel.

Consequently, a program that is intended to be compatible with LanguageLevel 1 interpreters must avoid using any of the constructs listed above. There are straightforward alternative methods for constructing dictionaries and strings, but none for indicating an immediately evaluated name.

LanguageLevel 3 introduces no additional syntax to the language.

D.4 Installing Emulations

When defining a PostScript emulation of an operator, it is important *not* to give the emulation the same name as the actual operator unless it is a complete emulation. This is because another page description included in the same job (an encapsulated file, for instance) may require a particular form of the operator that is

not emulated; when it encounters the emulation, an error will result. Note that in Example D.1 on page 767, the emulation of the **rectfill** operator is not complete and is not named rectfill.

Emulation of operators should be done conditionally, based on whether the operator already exists. For example, it does not make sense to define a procedure named selectfont if the real **selectfont** operator already exists. Conditional emulation can be performed in one of two ways:

- Use the **languagelevel** operator to determine whether to install emulations of all required features for a given LanguageLevel as a group.
- Use the **where** operator to determine whether to install emulations of individual operators. This is appropriate for those operators that are available as extensions to a lower LanguageLevel in some products (see Appendix A).

Example D.2 uses the first method to provide conditional emulation of the LanguageLevel 2 **selectfont** and **rectfill** operators. This example makes use of the *RF procedure defined in Example D.1.

Example D.2

```
/*SF                              % Complete selectfont emulation
   {  exch findfont exch
      dup type /arraytype eq
            {makefont}
            {scalefont}
         ifelse
      setfont
   } bind def

/languagelevel where               % Determine LanguageLevel of implementation
      {pop languagelevel}
      {1}
   ifelse
2 lt
      {  /SF /*SF load def         % Interpreter is LanguageLevel 1,
         /RF /*RF load def         % so use emulations defined above
      }
      {  /SF /selectfont load def  % Interpreter is LanguageLevel 2 or 3,
         /RF /rectfill load def    % so use LanguageLevel 2 operators
      }
   ifelse
```

The examples together define procedures named *SF and *RF to emulate the **selectfont** and **rectfill** operators, respectively. Then, based on the results of the **languagelevel** operator, Example D.2 binds either the emulations or the actual LanguageLevel 2 operators to short names—SF and RF—that can be used later in the page description. This approach has three noteworthy features:

- An actual operator will always be used, if available, in preference to an emulation.
- An emulation is never given the same name as an operator. Thus, embedded programs will not be fooled into believing that an operator is defined when it is not.
- The script of the page description can invoke operations using short names, such as SF and RF, without regard to whether those operations are performed by operators or by emulations.

In general, PostScript programs should not use the **version** operator to test for the availability of specific features. When a feature is introduced as an extension (rather than as part of a LanguageLevel), it may not be present in all products. To determine whether a feature is supported, it is usually best to use the **where** operator to check for the presence of an operator associated with the feature. For some features, it is better to use **resourcestatus** to query an implicit resource category, such as **FontType** or **HalftoneType**.

Although using the **where** operator to test for PostScript operators is appropriate, using it to test for application-defined procedures is not. Doing so can lead to trouble in the future if an operator of the same name happens to come into existence. The correct way to test for application-defined procedures is to look them up in the application's own dictionary with the **known** operator instead of the **where** operator.

APPENDIX E

Character Sets and Encoding Vectors

THIS APPENDIX DESCRIBES the character sets and encoding vectors of font programs that are found in a typical PostScript interpreter or that are available for downloading. While there is not a standard set of fonts that is required by the PostScript language, most PostScript products include software for 13 standard fonts from the Times*, Helvetica*, Courier, and Symbol families. Samples of these fonts appear in the first few sections below.

Following the font samples are tables documenting the entire character set for Adobe's standard Latin text fonts, expert fonts, and the Symbol font. For each character set, every character is shown along with its full name and octal character code (unencoded characters are indicated by —). This is followed by detailed tables of the encoding vectors normally associated with a font program using that character set. Table E.1 lists these encoding vectors.

TABLE E.1 Encoding vectors

ENCODING VECTOR	DESCRIPTION
StandardEncoding	Built-in standard encoding vector. The default encoding used in most regular Latin text fonts.
ISOLatin1Encoding	Built-in encoding vector that closely matches the ISO 8859-1 (Latin 1) standard. Encodes the characters used in most Western European languages.
CE	Central European encoding vector, matching Microsoft Windows code page 1250. Encodes the characters used in some Central European languages.

Expert	Encoding vector for "expert" fonts, which contain additional characters useful for sophisticated typography, such as small capitals, ligatures, and fractions.
ExpertSubset	Encoding vector for "expert" fonts that contains a subset of the expert character set.
Symbol	Encoding vector that is unique to the Symbol font.

StandardEncoding and **ISOLatin1Encoding** are names in **systemdict** and in the **Encoding** resource category associated with encoding array objects. The CE, Expert, ExpertSubset, and Symbol encoding vectors are provided in the individual font programs that use them; they are not defined as named encodings in the PostScript interpreter itself.

Adobe defines two standard character sets for regular Latin text fonts: an "original" set containing 229 characters and an "extended" set containing 315 characters. The original set includes all characters listed in **StandardEncoding**, **ISOLatin1Encoding**, and several other common encodings. The extended set additionally includes all characters listed in the CE encoding, as well as a number of other characters.

For more information on encoding vectors, see Section 5.3, "Character Encoding."

E.1 Times Family

In 1931, *The Times* of London commissioned Monotype corporation, under the direction of Stanley Morison, to design a newspaper typeface. Times New Roman® was the result. The Linotype version shown here is called Times Roman. It continues to be popular for both newspaper and business applications, such as reports and correspondence.

Times-Roman

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ‚ ˊ ˆ ¨ ` ˚ ~ ¯ ˘ ˙ ˇ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ 1 2 3

Times-Italic

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
*ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "*
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ‚ ˊ ˆ ¨ ` ˚ ~ ¯ ˘ ˙ ˇ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ 1 2 3

Times-Bold

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ‚ ˊ ˆ ¨ ` ˚ ~ ¯ ˘ ˙ ˇ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ 1 2 3

Times-BoldItalic

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ‚ ˊ ˆ ¨ ` ˚ ~ ¯ ˘ ˙ ˇ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ 1 2 3

E.2 Helvetica Family

One of the most popular typefaces of all time, Helvetica was designed by Max Miedinger in 1957 for the Hass foundry in Switzerland. The name is derived from *Helvetia,* the Swiss name for Switzerland. Helvetica's range of styles allows a variety of uses, including headlines, packaging, posters, and short text blocks such as captions.

Helvetica

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ˆ ˚ ¨ ` ˚ ~ ¯ ˘ · ˛ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ ¹ ² ³

Helvetica-Oblique

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
*ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "*
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ˆ ˚ ¨ ` ˚ ~ ¯ ˘ · ˛ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ ¹ ² ³

Helvetica-Bold

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ˆ ˚ ¨ ` ˚ ~ ¯ ˘ · ˛ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ ¹ ² ³

Helvetica-BoldOblique

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ˆ ˚ ¨ ` ˚ ~ ¯ ˘ · ˛ ˝ •
% ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ ⁄ · ¦ ¹ ² ³

E.3 Courier Family

Courier was originally designed as a typewriter face for IBM in 1952 by Howard Kettler. It is a monospaced, or fixed-pitch, font suitable for use in tabular material, program listings, or word processing.

Courier

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ´ ˆ ¨ ` ° ˜ ¯ ˘ · ˇ ˝ •
# % ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ / . ¦ ¹ ² ³
```

Courier-Oblique

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' '
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ´ ˆ ¨ ` ° ˜ ¯ ˘ · ˇ ˝ •
# % ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ / . ¦ ¹ ² ³
```

Courier-Bold

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' "
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ´ ˆ ¨ ` ° ˜ ¯ ˘ · ˇ ˝ •
# % ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ / . ¦ ¹ ² ³
```

Courier-BoldOblique

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z & 0 1 2 3 4 5 6 7 8 9
Æ Á Â Ä À Å Ã Ç Ð É Ê Ë È Í Î Ï Ì Ł Ñ Œ Ó Ô Ö Ò Õ Ø Š Þ Ú
Û Ü Ù Ÿ Ý Ž æ á â ä à å ã ç é ê ë è ð fi fl í î ï ì ı µ ł ñ œ ó ô ö ò õ ø š þ ß
ú û ü ù ÿ ý ž £ ¥ ƒ $ ¢ ¤ ™ © ® @ ª º † ‡ § ¶ * ! ¡ ? ¿ . , ; : ‘ ’ “ ” , „ … ' '
‹ › « » ( ) [ ] { } | / \ - – — _ ‚ ¸ ´ ˆ ¨ ` ° ˜ ¯ ˘ · ˇ ˝ •
# % ‰ ¼ ¾ ½ = − + × ~ < ± > ÷ ¬ ° ^ / . ¦ ¹ ² ³
```

E.4 Symbol

αβχδεφγηικλμνοπθρστυωξψζφϖϑς ∫ ⌠ { ⌈ ⌉ } [⌈ ⌉] (⌠ ⌡)
ABXΔEΦΓHIKΛMNOΠΘPΣTYΩΞΨZϒ ⎪ { } ⎪ ⎪ ⎪ ⎪
= ≠ ≡ ≈ ≅ < > ≤ ≥ ∧ ∨ ∴ − + ± × ÷ ¬ • · ⌡ ⌊ ⌋ ⌊ ⌋ ⌡ ⌠ ⌡)
∀ ∃ *f* ∂ ϶ ∩ ∪ ⊃ ⊂ ⊇ ⊆ ⊄ ∈ ∉ ∅ ⊗ ⊕ ∞ ∝
↔ ← — → ↑ ↓ ⇔ ⇐ ⇒ ⇑ ⇓ ° ′ ″ ↵ ℵ ℑ ℜ ℘
… ⟨ ◊ ® © ™ ® © ™ ∑ ∏ ⟩ ∠ ⊥ ∇ ♣ ♦ ♥ ♠
! # % * & 0 1 2 3 4 5 6 7 8 9 . , ; : _ ~ | ⎪ //√ ¯

Sample Uses

$$\varepsilon = \min_{x > 0} (x \mid (1 + x) \neq 1)$$

$$w(\xi' - \xi'') = \sum_{i=1}^{m} \mid C_i \cap \{ \xi' \} \mid \cdot \mid C_i \cap \{ \xi'' \} \mid \text{ if } \xi', \xi'' \in L \text{ and } \xi' \neq \xi''$$

$$\bigcup_{i=1}^{n} Z_i(t) \subseteq M$$

$$Z_i(t) \cap Z_j(t) = \varnothing \quad (i \neq j)$$

proposition	true if and only if
$(\forall u)\ s(p)$	$S \cap T_p' = \varnothing$
$(\exists u)\ s(p)$	$S \cap T_p \neq \varnothing$
$(\forall u)\ s(\sim p)$	$S \cap T_p = \varnothing$
$(\exists u)\ s(\sim p)$	$S \cap T_p' \neq \varnothing$
$\sim((\forall u)\ s(p))$	$S \cap T_p' \neq \varnothing$
$\sim((\exists u)\ s(p))$	$S \cap T_p = \varnothing$

$$kp^{k/2} t^{-1} I_k(at) \Leftrightarrow \left| \frac{s + \sqrt{s^2 - 4\lambda\mu}}{2\lambda} \right|^{-k}$$

E.5 Standard Latin Character Set

CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE	CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE
A	A	101	101	101	I	I	111	111	111
Æ	AE	341	306	—	Í	Iacute	—	315	315
Á	Aacute	—	301	301	Î	Icircumflex	—	316	316
Ă	Abreve[1]	—	—	303	Ï	Idieresis	—	317	—
Â	Acircumflex	—	302	302	İ	Idotaccent[1]	—	—	—
Ä	Adieresis	—	304	304	Ì	Igrave	—	314	—
À	Agrave	—	300	—	Ī	Imacron[1]	—	—	—
Ā	Amacron[1]	—	—	—	Į	Iogonek[1]	—	—	—
Ą	Aogonek[1]	—	—	245	J	J	112	112	112
Å	Aring	—	305	—	K	K	113	113	113
Ã	Atilde	—	303	—	Ķ	Kcommaaccent[1]	—	—	—
B	B	102	102	102	L	L	114	114	114
C	C	103	103	103	Ĺ	Lacute[1]	—	—	305
Ć	Cacute[1]	—	—	306	Ľ	Lcaron[1]	—	—	274
Č	Ccaron[1]	—	—	310	Ļ	Lcommaaccent[1]	—	—	—
Ç	Ccedilla	—	307	307	Ł	Lslash	350	—	243
D	D	104	104	104	M	M	115	115	115
Ď	Dcaron[1]	—	—	317	N	N	116	116	116
Đ	Dcroat[1]	—	—	320	Ń	Nacute[1]	—	—	321
Δ	Delta[1]	—	—	—	Ň	Ncaron[1]	—	—	322
E	E	105	105	105	Ņ	Ncommaaccent[1]	—	—	—
É	Eacute	—	311	311	Ñ	Ntilde	—	321	—
Ě	Ecaron[1]	—	—	314	O	O	117	117	117
Ê	Ecircumflex	—	312	—	Œ	OE	352	—	—
Ë	Edieresis	—	313	313	Ó	Oacute	—	323	323
Ė	Edotaccent[1]	—	—	—	Ô	Ocircumflex	—	324	324
È	Egrave	—	310	—	Ö	Odieresis	—	326	326
Ē	Emacron[1]	—	—	—	Ò	Ograve	—	322	—
Ę	Eogonek[1]	—	—	312	Ő	Ohungarumlaut[1]	—	—	325
Ð	Eth	—	320	—	Ō	Omacron[1]	—	—	—
F	F	106	106	106	Ø	Oslash	351	330	—
G	G	107	107	107	Õ	Otilde	—	325	—
Ğ	Gbreve[1]	—	—	—	P	P	120	120	120
Ģ	Gcommaaccent[1]	—	—	—	Q	Q	121	121	121
H	H	110	110	110	R	R	122	122	122

CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE	CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE
Ŕ	Racute[1]	—	—	300	æ	ae	361	346	—
Ř	Rcaron[1]	—	—	330	à	agrave	—	340	—
Ŗ	Rcommaaccent[1]	—	—	—	ā	amacron[1]	—	—	—
S	S	123	123	123	&	ampersand	046	046	046
Ś	Sacute[1]	—	—	214	ą	aogonek[1]	—	—	271
Š	Scaron	—	—	212	å	aring	—	345	—
Ş	Scedilla[1]	—	—	—	^	asciicircum	136	136	136
Ș	Scommaaccent[1]	—	—	252	~	asciitilde	176	176	176
T	T	124	124	124	*	asterisk	052	052	052
Ť	Tcaron[1]	—	—	215	@	at	100	100	100
Ţ	Tcommaaccent[1]	—	—	336	ã	atilde	—	343	—
Þ	Thorn	—	336	—	b	b	142	142	142
U	U	125	125	125	\	backslash	134	134	134
Ú	Uacute	—	332	332	|	bar	174	174	174
Û	Ucircumflex	—	333	—	{	braceleft	173	173	173
Ü	Udieresis	—	334	334	}	braceright	175	175	175
Ù	Ugrave	—	331	—	[	bracketleft	133	133	133
Ű	Uhungarumlaut[1]	—	—	333	]	bracketright	135	135	135
Ū	Umacron[1]	—	—	—	˘	breve[2]	306	226	242
Ų	Uogonek[1]	—	—	—	¦	brokenbar	—	246	246
Ů	Uring[1]	—	—	331	•	bullet	267	—	225
V	V	126	126	126	c	c	143	143	143
W	W	127	127	127	ć	cacute[1]	—	—	346
X	X	130	130	130	ˇ	caron	317	237	241
Y	Y	131	131	131	č	ccaron[1]	—	—	350
Ý	Yacute	—	335	335	ç	ccedilla	—	347	347
Ÿ	Ydieresis	—	—	—	¸	cedilla[2]	313	270	270
Z	Z	132	132	132	¢	cent	242	242	—
Ź	Zacute[1]	—	—	217	ˆ	circumflex	303	223	—
Ž	Zcaron	—	—	216	:	colon	072	072	072
Ż	Zdotaccent[1]	—	—	257	,	comma	054	054	054
a	a	141	141	141	¸	commaaccent[1]	—	—	—
á	aacute	—	341	341	©	copyright	—	251	251
ă	abreve[1]	—	—	343	¤	currency	250	244	244
â	acircumflex	—	342	342	d	d	144	144	144
´	acute[2]	302	264	264	†	dagger	262	—	206
ä	adieresis	—	344	344	‡	daggerdbl	263	—	207

CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE	CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE
ď	dcaron[1]	—	—	357	>	greater	076	076	076
đ	dcroat[1]	—	—	360	≥	greaterequal[1]	—	—	—
°	degree	—	260	260	«	guillemotleft[4]	253	253	253
¨	dieresis[2]	310	250	250	»	guillemotright[4]	273	273	273
÷	divide	—	367	367	‹	guilsinglleft	254	—	213
$	dollar	044	044	044	›	guilsinglright	255	—	233
˙	dotaccent	307	227	377	h	h	150	150	150
ı	dotlessi	365	220	—	˝	hungarumlaut	315	235	275
e	e	145	145	145	-	hyphen	055	255	055
é	eacute	—	351	351	i	i	151	151	151
ě	ecaron[1]	—	—	354	í	iacute	—	355	355
ê	ecircumflex	—	352	—	î	icircumflex	—	356	356
ë	edieresis	—	353	353	ï	idieresis	—	357	—
ė	edotaccent[1]	—	—	—	ì	igrave	—	354	—
è	egrave	—	350	—	ī	imacron[1]	—	—	—
8	eight	070	070	070	į	iogonek[1]	—	—	—
…	ellipsis	274	—	205	j	j	152	152	152
ē	emacron[1]	—	—	—	k	k	153	153	153
—	emdash	320	—	227	ķ	kcommaaccent[1]	—	—	—
–	endash	261	—	226	l	l	154	154	154
ę	eogonek[1]	—	—	352	Í	lacute[1]	—	—	345
=	equal	075	075	075	ľ	lcaron[1]	—	—	276
ð	eth	—	360	—	ļ	lcommaaccent[1]	—	—	—
!	exclam	041	041	041	<	less	074	074	074
¡	exclamdown	241	241	—	≤	lessequal[1]	—	—	—
f	f	146	146	146	¬	logicalnot	—	254	254
fi	fi	256	—	—	◊	lozenge[1]	—	—	—
5	five	065	065	065	ł	lslash	370	—	263
fl	fl	257	—	—	m	m	155	155	155
ƒ	florin	246	—	—	¯	macron[2]	305	257	—
4	four	064	064	064	–	minus	—	055	—
⁄	fraction	244	—	—	µ	mu	—	265	265
g	g	147	147	147	×	multiply	—	327	327
ğ	gbreve[1]	—	—	—	n	n	156	156	156
ģ	gcommaaccent[1]	—	—	—	ń	nacute[1]	—	—	361
ß	germandbls	373	337	337	ň	ncaron[1]	—	—	362
`	grave[3]	301	221	140	ņ	ncommaaccent[1]	—	—	—

CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE	CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE
9	nine	071	071	071	“	quotedblleft	252	—	223
≠	notequal[1]	—	—	—	”	quotedblright	272	—	224
ñ	ntilde	—	361	—	‘	quoteleft	140	140	221
#	numbersign	043	043	043	’	quoteright	047	047	222
o	o	157	157	157	‚	quotesinglbase	270	—	202
ó	oacute	—	363	363	'	quotesingle	251	—	047
ô	ocircumflex	—	364	364	r	r	162	162	162
ö	odieresis	—	366	366	ŕ	racute[1]	—	—	340
œ	oe	372	—	—	√	radical[1]	—	—	—
˛	ogonek	316	236	262	ř	rcaron[1]	—	—	370
ò	ograve	—	362	—	ŗ	rcommaaccent[1]	—	—	—
ő	ohungarumlaut[1]	—	—	365	®	registered	—	256	256
ō	omacron[1]	—	—	—	˚	ring	312	232	—
1	one	061	061	061	s	s	163	163	163
½	onehalf	—	275	—	ś	sacute[1]	—	—	234
¼	onequarter	—	274	—	š	scaron	—	—	232
¹	onesuperior	—	271	—	ş	scedilla[1]	—	—	—
ª	ordfeminine	343	252	—	ș	scommaaccent[1]	—	—	272
º	ordmasculine	353	272	—	§	section	247	247	247
ø	oslash	371	370	—	;	semicolon	073	073	073
õ	otilde	—	365	—	7	seven	067	067	067
p	p	160	160	160	6	six	066	066	066
¶	paragraph	266	266	266	/	slash	057	057	057
(	parenleft	050	050	050		space[5]	040	040	040
)	parenright	051	051	051	£	sterling	243	243	—
∂	partialdiff[1]	—	—	—	Σ	summation[1]	—	—	—
%	percent	045	045	045	t	t	164	164	164
.	period	056	056	056	ť	tcaron[1]	—	—	235
·	periodcentered	264	267	267	ţ	tcommaaccent[1]	—	—	376
‰	perthousand	275	—	211	þ	thorn	—	376	—
+	plus	053	053	053	3	three	063	063	063
±	plusminus	—	261	261	¾	threequarters	—	276	—
q	q	161	161	161	³	threesuperior	—	263	—
?	question	077	077	077	˜	tilde	304	224	—
¿	questiondown	277	277	—	™	trademark	—	—	231
"	quotedbl	042	042	042	2	two	062	062	062
„	quotedblbase	271	—	204	²	twosuperior	—	262	—

CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE	CHAR	NAME	CHAR CODE (OCTAL) STD	ISO	CE
u	u	165	165	165	w	w	167	167	167
ú	uacute	—	372	372	x	x	170	170	170
û	ucircumflex	—	373	—	y	y	171	171	171
ü	udieresis	—	374	374	ý	yacute	—	375	375
ù	ugrave	—	371	—	ÿ	ydieresis	—	377	—
ű	uhungarumlaut[1]	—	—	373	¥	yen	245	245	—
ū	umacron[1]	—	—	—	z	z	172	172	172
_	underscore	137	137	137	ź	zacute[1]	—	—	237
ų	uogonek[1]	—	—	—	ž	zcaron	—	—	236
ů	uring[1]	—	—	371	ż	zdotaccent[1]	—	—	277
v	v	166	166	166	0	zero	060	060	060

1. These characters are present in the extended (315-character) Latin character set, but not in the original (229-character) set.
2. In the ISO 8859-1 standard, character codes in the range 220 through 237 are unused. In the **ISOLatin1Encoding** encoding vector, these character codes are assigned to accent characters, some of which are duplicated in other parts of the encoding. This is for historical reasons only.
3. The **ISOLatin1Encoding** encoding vector deviates from the ISO 8859-1 standard in one respect: the character at position 140 is quoteleft, whereas the ISO standard specifies grave. A PostScript program needing to conform exactly to the ISO standard should create a modified encoding vector with this entry changed.
4. The character names guillemotleft and guillemotright are misspelled. The correct spelling for this punctuation character is "guillemet." However, the misspelled names are the ones actually used in the fonts and encodings containing these characters.
5. This character is also encoded as 240 in the ISOLatin1Encoding and CE encoding vectors. The meaning of code 240 is "nonbreaking space," but it is typographically the same as space.

E.6 StandardEncoding Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x		!	"	#	$	%	&	’
\05x	(	)	*	+	,	-	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	@	A	B	C	D	E	F	G
\11x	H	I	J	K	L	M	N	O
\12x	P	Q	R	S	T	U	V	W
\13x	X	Y	Z	[	\	]	^	_
\14x	‘	a	b	c	d	e	f	g
\15x	h	i	j	k	l	m	n	o
\16x	p	q	r	s	t	u	v	w
\17x	x	y	z	{	|	}	~	
\20x								
\21x								
\22x								
\23x								
\24x		¡	¢	£	⁄	¥	*ƒ*	§
\25x	¤	'	“	«	‹	›	fi	fl
\26x		–	†	‡	·		¶	•
\27x	‚	„	”	»	…	‰		¿
\30x		`	´	ˆ	˜	¯	˘	˙
\31x	¨		°	¸		˝	˛	ˇ
\32x	—							
\33x								
\34x		Æ		ª				
\35x	Ł	Ø	Œ	º				
\36x		æ				ı		
\37x	ł	ø	œ	ß				

E.7 ISOLatin1Encoding Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x		!	"	#	$	%	&	’
\05x	(	)	*	+	,	−	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	@	A	B	C	D	E	F	G
\11x	H	I	J	K	L	M	N	O
\12x	P	Q	R	S	T	U	V	W
\13x	X	Y	Z	[	\	]	^	_
\14x	‘	a	b	c	d	e	f	g
\15x	h	i	j	k	l	m	n	o
\16x	p	q	r	s	t	u	v	w
\17x	x	y	z	{	|	}	~	
\20x								
\21x								
\22x	ı	`	´	ˆ	˜	¯	˘	˙
\23x	¨		°	¸		˝	˛	ˇ
\24x		¡	¢	£	¤	¥	¦	§
\25x	¨	©	ª	«	¬	-	®	¯
\26x	º	±	2	3	´	µ	¶	·
\27x	¸	1	º	»	¼	½	¾	¿
\30x	À	Á	Â	Ã	Ä	Å	Æ	Ç
\31x	È	É	Ê	Ë	Ì	Í	Î	Ï
\32x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×
\33x	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
\34x	à	á	â	ã	ä	å	æ	ç
\35x	è	é	ê	ë	ì	í	î	ï
\36x	ð	ñ	ò	ó	ô	õ	ö	÷
\37x	ø	ù	ú	û	ü	ý	þ	ÿ

E.8 CE Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x		!	"	#	$	%	&	'
\05x	(	)	*	+	,	-	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	@	A	B	C	D	E	F	G
\11x	H	I	J	K	L	M	N	O
\12x	P	Q	R	S	T	U	V	W
\13x	X	Y	Z	[	\	]	^	_
\14x	`	a	b	c	d	e	f	g
\15x	h	i	j	k	l	m	n	o
\16x	p	q	r	s	t	u	v	w
\17x	x	y	z	{	|	}	~	
\20x			‚		„	…	†	‡
\21x		‰	Š	‹	Ś	Ť	Ž	Ź
\22x		‘	’	“	”	•	–	—
\23x		™	š	›	ś	ť	ž	ź
\24x		ˇ	˘	Ł	¤	Ą	¦	§
\25x	¨	©	Ş	«	¬	-	®	Ż
\26x	°	±	˛	ł	´	µ	¶	·
\27x	¸	ą	ş	»	Ľ	˝	ľ	ż
\30x	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç
\31x	Č	É	Ę	Ë	Ě	Í	Î	Ď
\32x	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×
\33x	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
\34x	ŕ	á	â	ă	ä	ĺ	ć	ç
\35x	č	é	ę	ë	ě	í	î	ď
\36x	đ	ń	ň	ó	ô	ő	ö	÷
\37x	ř	ů	ú	ű	ü	ý	ţ	˙

E.9 Expert Character Set

CHAR	NAME	CODE	CHAR	NAME	CODE
Æ	AEsmall	346	J	Jsmall	152
Á	Aacutesmall	341	K	Ksmall	153
Â	Acircumflexsmall	342	Ł	Lslashsmall	243
´	Acutesmall	047	L	Lsmall	154
Ä	Adieresissmall	344	¯	Macronsmall	257
À	Agravesmall	340	M	Msmall	155
Å	Aringsmall	345	N	Nsmall	156
A	Asmall	141	Ñ	Ntildesmall	361
Ã	Atildesmall	343	Œ	OEsmall	367
˘	Brevesmall	251	Ó	Oacutesmall	363
B	Bsmall	142	Ô	Ocircumflexsmall	364
ˇ	Caronsmall	252	Ö	Odieresissmall	366
Ç	Ccedillasmall	347	˛	Ogoneksmall	266
¸	Cedillasmall	270	Ò	Ogravesmall	362
ˆ	Circumflexsmall	136	Ø	Oslashsmall	370
C	Csmall	143	O	Osmall	157
¨	Dieresissmall	250	Õ	Otildesmall	365
˙	Dotaccentsmall	254	P	Psmall	160
D	Dsmall	144	Q	Qsmall	161
É	Eacutesmall	351	°	Ringsmall	267
Ê	Ecircumflexsmall	352	R	Rsmall	162
Ë	Edieresissmall	353	Š	Scaronsmall	246
È	Egravesmall	350	S	Ssmall	163
E	Esmall	145	Þ	Thornsmall	376
Ð	Ethsmall	360	˜	Tildesmall	176
F	Fsmall	146	T	Tsmall	164
`	Gravesmall	140	Ú	Uacutesmall	372
G	Gsmall	147	Û	Ucircumflexsmall	373
H	Hsmall	150	Ü	Udieresissmall	374
˝	Hungarumlautsmall	042	Ù	Ugravesmall	371
Í	Iacutesmall	355	U	Usmall	165
Î	Icircumflexsmall	356	V	Vsmall	166
Ï	Idieresissmall	357	W	Wsmall	167
Ì	Igravesmall	354	X	Xsmall	170
I	Ismall	151	Ý	Yacutesmall	375

CHAR	NAME	CODE	CHAR	NAME	CODE
Ÿ	Ydieresissmall	377	4	fouroldstyle	064
Y	Ysmall	171	⁴	foursuperior	314
Ž	Zcaronsmall	247	⁄	fraction	057
Z	Zsmall	172	-	hyphen	055
&	ampersandsmall	046	-	hypheninferior	263
ª	asuperior	101	⁻	hyphensuperior	137
ᵇ	bsuperior	102	i	isuperior	111
¢	centinferior	334	l	lsuperior	114
¢	centoldstyle	242	ᵐ	msuperior	115
¢	centsuperior	103	9	nineinferior	333
:	colon	072	9	nineoldstyle	071
₡	colonmonetary	173	⁹	ninesuperior	321
,	comma	054	ⁿ	nsuperior	116
,	commainferior	337	.	onedotenleader	053
’	commasuperior	074	⅛	oneeighth	300
$	dollarinferior	335	1	onefitted	174
$	dollaroldstyle	044	½	onehalf	275
$	dollarsuperior	045	₁	oneinferior	323
ᵈ	dsuperior	104	1	oneoldstyle	061
8	eightinferior	332	¼	onequarter	274
8	eightoldstyle	070	¹	onesuperior	311
⁸	eightsuperior	320	⅓	onethird	304
ᵉ	esuperior	105	º	osuperior	117
¡	exclamdownsmall	241	(	parenleftinferior	133
!	exclamsmall	041	(	parenleftsuperior	050
ff	ff	126	)	parenrightinferior	135
ffi	ffi	131	)	parenrightsuperior	051
ffl	ffl	132	.	period	056
fi	fi	127	.	periodinferior	336
–	figuredash	262	·	periodsuperior	076
⅝	fiveeighths	302	¿	questiondownsmall	277
5	fiveinferior	327	?	questionsmall	077
5	fiveoldstyle	065	ʳ	rsuperior	122
⁵	fivesuperior	315	Rp	rupiah	175
fl	fl	130	;	semicolon	073
₄	fourinferior	326	⅞	seveneighths	303

CHAR	NAME	CODE	CHAR	NAME	CODE
7	seveninferior	331	—	threequartersemdash	075
7	sevenoldstyle	067	3	threesuperior	313
7	sevensuperior	317	t	tsuperior	124
6	sixinferior	330	..	twodotenleader	052
6	sixoldstyle	066	2	twoinferior	324
6	sixsuperior	316	2	twooldstyle	062
	space	040	2	twosuperior	312
s	ssuperior	123	⅔	twothirds	305
⅜	threeeighths	301	0	zeroinferior	322
3	threeinferior	325	0	zerooldstyle	060
3	threeoldstyle	063	0	zerosuperior	310
¾	threequarters	276			

E.10 Expert Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x		!	″		$	$	&	´
\05x	(	)	..	.	,	-	.	⁄
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	’	—	·	?
\10x		a	b	¢	d	e		
\11x		i			l	m	n	o
\12x			r	s	t		ff	fi
\13x	fl	ffi	ffl	(		)	ˆ	-
\14x	`	A	B	C	D	E	F	G
\15x	H	I	J	K	L	M	N	O
\16x	P	Q	R	S	T	U	V	W
\17x	X	Y	Z	₡	1	Rp	˜	
\20x								
\21x								
\22x								
\23x								
\24x		¡	¢	Ł			Š	Ž
\25x	¨	˘	˚		·			˙
\26x			–	-			˛	°
\27x	¸				¼	½	¾	¿
\30x	⅛	⅜	⅝	⅞	⅓	⅔		
\31x	0	1	2	3	4	5	6	7
\32x	8	9	0	1	2	3	4	5
\33x	6	7	8	9	¢	$	.	,
\34x	À	Á	Â	Ã	Ä	Å	Æ	Ç
\35x	È	É	Ê	Ë	Ì	Í	Î	Ï
\36x	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Œ
\37x	Ø	Ù	Ú	Û	Ü	Ý	Þ	Ÿ

E.11 ExpertSubset Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x					$	$		
\05x	(	)	..	.	,	-	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	’	—	·	
\10x		a	b	¢	d	e		
\11x		i			l	m	n	o
\12x			r	s	t		ff	fi
\13x	fl	ffi	ffl	(		)		-
\14x								
\15x								
\16x								
\17x				₡	1	Rp		
\20x								
\21x								
\22x								
\23x								
\24x			¢					
\25x								
\26x			–	-				
\27x					¼	½	¾	
\30x	⅛	⅜	⅝	⅞	⅓	⅔		
\31x	0	1	2	3	4	5	6	7
\32x	8	9	0	1	2	3	4	5
\33x	6	7	8	9	¢	$	.	,
\34x								
\35x								
\36x								
\37x								

E.12 Symbol Character Set

CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE
A	Alpha	101	↔	arrowboth	253	⊗	circlemultiply	304
B	Beta	102	⇔	arrowdblboth	333	⊕	circleplus	305
X	Chi	103	⇓	arrowdbldown	337	♣	club	247
Δ	Delta	104	⇐	arrowdblleft	334	:	colon	072
E	Epsilon	105	⇒	arrowdblright	336	,	comma	054
H	Eta	110	⇑	arrowdblup	335	≅	congruent	100
€	Euro	240	↓	arrowdown	257	©	copyrightsans	343
Γ	Gamma	107	—	arrowhorizex	276	©	copyrightserif	323
ℑ	Ifraktur	301	←	arrowleft	254	°	degree	260
I	Iota	111	→	arrowright	256	δ	delta	144
K	Kappa	113	↑	arrowup	255	♦	diamond	250
Λ	Lambda	114	⏐	arrowvertex	275	÷	divide	270
M	Mu	115	*	asteriskmath	052	·	dotmath	327
N	Nu	116	|	bar	174	8	eight	070
Ω	Omega	127	β	beta	142	∈	element	316
O	Omicron	117	{	braceleft	173	…	ellipsis	274
Φ	Phi	106	}	braceright	175	∅	emptyset	306
Π	Pi	120	⎧	braceleftttp	354	ε	epsilon	145
Ψ	Psi	131	⎨	braceleftmid	355	=	equal	075
ℜ	Rfraktur	302	⎩	braceleftbt	356	≡	equivalence	272
P	Rho	122	⎫	bracerighttp	374	η	eta	150
Σ	Sigma	123	⎬	bracerightmid	375	!	exclam	041
T	Tau	124	⎭	bracerightbt	376	∃	existential	044
Θ	Theta	121	⏐	braceex	357	5	five	065
Y	Upsilon	125	[	bracketleft	133	*f*	florin	246
ϒ	Upsilon1	241	]	bracketright	135	4	four	064
Ξ	Xi	130	⎡	bracketlefttp	351	⁄	fraction	244
Z	Zeta	132	⎢	bracketleftex	352	γ	gamma	147
ℵ	aleph	300	⎣	bracketleftbt	353	∇	gradient	321
α	alpha	141	⎤	bracketrighttp	371	>	greater	076
&	ampersand	046	⎥	bracketrightex	372	≥	greaterequal	263
∠	angle	320	⎦	bracketrightbt	373	♥	heart	251
⟨	angleleft	341	•	bullet	267	∞	infinity	245
⟩	angleright	361	↵	carriagereturn	277	∫	integral	362
≈	approxequal	273	χ	chi	143	⌠	integraltp	363

CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE
⌡	integralex	364	⎛	parenlefttp	346	″	second	262
⌡	integralbt	365	⎜	parenleftex	347	;	semicolon	073
∩	intersection	307	⎝	parenleftbt	350	7	seven	067
ι	iota	151	⎞	parenrighttp	366	σ	sigma	163
κ	kappa	153	⎟	parenrightex	367	ς	sigma1	126
λ	lambda	154	⎠	parenrightbt	370	~	similar	176
<	less	074	∂	partialdiff	266	6	six	066
≤	lessequal	243	%	percent	045	/	slash	057
∧	logicaland	331	.	period	056		space	040
¬	logicalnot	330	⊥	perpendicular	136	♠	spade	252
∨	logicalor	332	ϕ	phi	146	∍	suchthat	047
◊	lozenge	340	φ	phi1	152	∑	summation	345
–	minus	055	π	pi	160	τ	tau	164
′	minute	242	+	plus	053	∴	therefore	134
μ	mu	155	±	plusminus	261	θ	theta	161
×	multiply	264	∏	product	325	ϑ	theta1	112
9	nine	071	⊂	propersubset	314	3	three	063
∉	notelement	317	⊃	propersuperset	311	™	trademarksans	344
≠	notequal	271	∝	proportional	265	™	trademarkserif	324
⊄	notsubset	313	ψ	psi	171	2	two	062
ν	nu	156	?	question	077	_	underscore	137
#	numbersign	043	√	radical	326	∪	union	310
ω	omega	167		radicalex	140	∀	universal	042
ϖ	omega1	166	⊆	reflexsubset	315	υ	upsilon	165
o	omicron	157	⊇	reflexsuperset	312	℘	weierstrass	303
1	one	061	®	registersans	342	ξ	xi	170
(	parenleft	050	®	registerserif	322	0	zero	060
)	parenright	051	ρ	rho	162	ζ	zeta	172

E.13 Symbol Encoding Vector

octal	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
\00x								
\01x								
\02x								
\03x								
\04x		!	∀	#	∃	%	&	э
\05x	(	)	*	+	,	−	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	≅	Α	Β	Χ	Δ	Ε	Φ	Γ
\11x	Η	Ι	ϑ	Κ	Λ	Μ	Ν	Ο
\12x	Π	Θ	Ρ	Σ	Τ	Υ	ς	Ω
\13x	Ξ	Ψ	Ζ	[	∴	]	⊥	_
\14x	‾	α	β	χ	δ	ε	ϕ	γ
\15x	η	ι	φ	κ	λ	μ	ν	ο
\16x	π	θ	ρ	σ	τ	υ	ϖ	ω
\17x	ξ	ψ	ζ	{	|	}	~	
\20x								
\21x								
\22x								
\23x								
\24x	€	ϒ	′	≤	⁄	∞	*f*	♣
\25x	♦	♥	♠	↔	←	↑	→	↓
\26x	°	±	″	≥	×	∝	∂	•
\27x	÷	≠	≡	≈	…	|	—	↵
\30x	ℵ	ℑ	ℜ	℘	⊗	⊕	∅	∩
\31x	∪	⊃	⊇	⊄	⊂	⊆	∈	∉
\32x	∠	∇	®	©	™	∏	√	·
\33x	¬	∧	∨	⇔	⇐	⇑	⇒	⇓
\34x	◊	〈	®	©	™	∑	⎛	⎜
\35x	⎝	⎡	⎜	⎣	⎢	⎧	⎩	⎜
\36x		〉	⌠	⌡	⎜	⎭	⎫	⎜
\37x	⎭	⎤	⎜	⎦	⎥	⎨	⎭	

APPENDIX F

System Name Encodings

INDEX	NAME	INDEX	NAME	INDEX	NAME
0	abs	23	concat	46	cvn
1	add	24	concatmatrix	47	cvr
2	aload	25	copy	48	cvrs
3	anchorsearch	26	count	49	cvs
4	and	27	counttomark	50	cvx
5	arc	28	currentcmykcolor	51	def
6	arcn	29	currentdash	52	defineusername
7	arct	30	currentdict	53	dict
8	arcto	31	currentfile	54	div
9	array	32	currentfont	55	dtransform
10	ashow	33	currentgray	56	dup
11	astore	34	currentgstate	57	end
12	awidthshow	35	currenthsbcolor	58	eoclip
13	begin	36	currentlinecap	59	eofill
14	bind	37	currentlinejoin	60	eoviewclip
15	bitshift	38	currentlinewidth	61	eq
16	ceiling	39	currentmatrix	62	exch
17	charpath	40	currentpoint	63	exec
18	clear	41	currentrgbcolor	64	exit
19	cleartomark	42	currentshared	65	file
20	clip	43	curveto	66	fill
21	clippath	44	cvi	67	findfont
22	closepath	45	cvlit	68	flattenpath

69	floor	101	loop	133	rlineto
70	flush	102	lt	134	rmoveto
71	flushfile	103	makefont	135	roll
72	for	104	matrix	136	rotate
73	forall	105	maxlength	137	round
74	ge	106	mod	138	save
75	get	107	moveto	139	scale
76	getinterval	108	mul	140	scalefont
77	grestore	109	ne	141	search
78	gsave	110	neg	142	selectfont
79	gstate	111	newpath	143	setbbox
80	gt	112	not	144	setcachedevice
81	identmatrix	113	null	145	setcachedevice2
82	idiv	114	or	146	setcharwidth
83	idtransform	115	pathbbox	147	setcmykcolor
84	if	116	pathforall	148	setdash
85	ifelse	117	pop	149	setfont
86	image	118	print	150	setgray
87	imagemask	119	printobject	151	setgstate
88	index	120	put	152	sethsbcolor
89	ineofill	121	putinterval	153	setlinecap
90	infill	122	rcurveto	154	setlinejoin
91	initviewclip	123	read	155	setlinewidth
92	inueofill	124	readhexstring	156	setmatrix
93	inufill	125	readline	157	setrgbcolor
94	invertmatrix	126	readstring	158	setshared
95	itransform	127	rectclip	159	shareddict
96	known	128	rectfill	160	show
97	le	129	rectstroke	161	showpage
98	length	130	rectviewclip	162	stop
99	lineto	131	repeat	163	stopped
100	load	132	restore	164	store

165	string	197	FontDirectory	259	StandardEncoding
166	stringwidth	198	SharedFontDirectory	260	[
167	stroke	199	Courier	261	]
168	strokepath	200	Courier-Bold	262	atan
169	sub	201	Courier-BoldOblique	263	banddevice
170	systemdict	202	Courier-Oblique	264	bytesavailable
171	token	203	Helvetica	265	cachestatus
172	transform	204	Helvetica-Bold	266	closefile
173	translate	205	Helvetica-BoldOblique	267	colorimage
174	truncate	206	Helvetica-Oblique	268	condition
175	type	207	Symbol	269	copypage
176	uappend	208	Times-Bold	270	cos
177	ucache	209	Times-BoldItalic	271	countdictstack
178	ueofill	210	Times-Italic	272	countexecstack
179	ufill	211	Times-Roman	273	cshow
180	undef	212	execuserobject	274	currentblackgeneration
181	upath	213	currentcolor	275	currentcacheparams
182	userdict	214	currentcolorspace	276	currentcolorscreen
183	ustroke	215	currentglobal	277	currentcolortransfer
184	viewclip	216	execform	278	currentcontext
185	viewclippath	217	filter	279	currentflat
186	where	218	findresource	280	currenthalftone
187	widthshow	219	globaldict	281	currenthalftonephase
188	write	220	makepattern	282	currentmiterlimit
189	writehexstring	221	setcolor	283	currentobjectformat
190	writeobject	222	setcolorspace	284	currentpacking
191	writestring	223	setglobal	285	currentscreen
192	wtranslation	224	setpagedevice	286	currentstrokeadjust
193	xor	225	setpattern	287	currenttransfer
194	xshow	256	=	288	currentundercolorremoval
195	xyshow	257	==	289	defaultmatrix
196	yshow	258	ISOLatin1Encoding	290	definefont

291	deletefile	323	packedarray	355	sqrt
292	detach	324	quit	356	srand
293	deviceinfo	325	rand	357	stack
294	dictstack	326	rcheck	358	status
295	echo	327	readonly	359	statusdict
296	erasepage	328	realtime	360	true
297	errordict	329	renamefile	361	ucachestatus
298	execstack	330	renderbands	362	undefinefont
299	executeonly	331	resetfile	363	usertime
300	exp	332	reversepath	364	ustrokepath
301	false	333	rootfont	365	version
302	filenameforall	334	rrand	366	vmreclaim
303	fileposition	335	run	367	vmstatus
304	fork	336	scheck	368	wait
305	framedevice	337	setblackgeneration	369	wcheck
306	grestoreall	338	setcachelimit	370	xcheck
307	handleerror	339	setcacheparams	371	yield
308	initclip	340	setcolorscreen	372	defineuserobject
309	initgraphics	341	setcolortransfer	373	undefineuserobject
310	initmatrix	342	setfileposition	374	UserObjects
311	instroke	343	setflat	375	cleardictstack
312	inustroke	344	sethalftone	376	A
313	join	345	sethalftonephase	377	B
314	kshow	346	setmiterlimit	378	C
315	ln	347	setobjectformat	379	D
316	lock	348	setpacking	380	E
317	log	349	setscreen	381	F
318	mark	350	setstrokeadjust	382	G
319	monitor	351	settransfer	383	H
320	noaccess	352	setucacheparams	384	I
321	notify	353	setundercolorremoval	385	J
322	nulldevice	354	sin	386	K

387	L	419	r	451	setsystemparams
388	M	420	s	452	setuserparams
389	N	421	t	453	startjob
390	O	422	u	454	undefineresource
391	P	423	v	455	GlobalFontDirectory
392	Q	424	w	456	ASCII85Decode
393	R	425	x	457	ASCII85Encode
394	S	426	y	458	ASCIIHexDecode
395	T	427	z	459	ASCIIHexEncode
396	U	428	setvmthreshold	460	CCITTFaxDecode
397	V	429	<<	461	CCITTFaxEncode
398	W	430	>>	462	DCTDecode
399	X	431	currentcolorrendering	463	DCTEncode
400	Y	432	currentdevparams	464	LZWDecode
401	Z	433	currentoverprint	465	LZWEncode
402	a	434	currentpagedevice	466	NullEncode
403	b	435	currentsystemparams	467	RunLengthDecode
404	c	436	currentuserparams	468	RunLengthEncode
405	d	437	defineresource	469	SubFileDecode
406	e	438	findencoding	470	CIEBasedA
407	f	439	gcheck	471	CIEBasedABC
408	g	440	glyphshow	472	DeviceCMYK
409	h	441	languagelevel	473	DeviceGray
410	i	442	product	474	DeviceRGB
411	j	443	pstack	475	Indexed
412	k	444	resourceforall	476	Pattern
413	l	445	resourcestatus	477	Separation
414	m	446	revision	478	CIEBasedDEF
415	n	447	serialnumber	479	CIEBasedDEFG
416	o	448	setcolorrendering	480	DeviceN
417	p	449	setdevparams		
418	q	450	setoverprint		

APPENDIX G

Operator Usage Guidelines

IF NOT USED PROPERLY, some PostScript operators can cause unintended side effects, render a document device-dependent, or inhibit postprocessing of a document. The guidelines in this appendix will help ensure the proper use of those operators. These guidelines apply to regular page descriptions and encapsulated PostScript (EPS) files, as described below. In addition, most of the EPS guidelines also apply to the definition of a **PaintProc** procedure in a form or pattern dictionary, a **BuildGlyph** procedure in a Type 3 base font or Type 1 CIDFont, and a **CharStrings** procedure in any font format that allows glyph descriptions to be replaced by PostScript procedures.

As discussed in Section 2.4, "Using the PostScript Language," the primary use of the PostScript language is to represent a page description, which is a device-independent representation of the appearance of pages that are to be viewed or printed. A page description not only is a valid PostScript program but also conforms to certain structuring conventions and usage guidelines. These guidelines help to ensure device independence and facilitate postprocessing of a page description by other applications.

There are two main classes of page description:

- A *regular page description* is a PostScript program produced by a document composition program—for example, a word processor or page-layout program. Typically, the PostScript program produces several pages, uses a number of fonts and other resources, and activates some printer-specific features such as paper trays or other physical requirements. A regular page description does not normally query the printer, perform calibration functions, cause VM to be permanently modified, or produce color separations.

PostScript programs that have the notation %!PS-Adobe-3.0 as the first line of the file are considered to be regular page descriptions that conform to the document structuring conventions (DSC) version 3.0. See Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*, for more information about DSC conformance and conventions. There is no requirement that a regular page description conform to DSC, but it is strongly recommended.

- An *encapsulated PostScript (EPS) file* is a PostScript program describing at most a single page in a form that can be imported by other applications to embed within a containing document. EPS files follow specific guidelines and have a particular structure that is further described in Adobe Technical Note #5002, *Encapsulated PostScript File Format Specification*. In particular, they must be device-independent and must not invoke printer-specific operators.

Table G.1 summarizes the use of specific operators in either a regular page description or an EPS file; further details are provided for each of these operators following the table. "No" in the table indicates that the operator should not be used; alternate suggestions, if any, are given in the details for that operator. "Careful" indicates that the operator can be used if appropriate precautions are taken. The rationale for the guidelines and the precautions that should be taken are provided in the details for each operator. Note that the guidelines are not enforced; however, failing to observe the guidelines may result in incorrect output.

TABLE G.1 Guidelines summary

OPERATOR	REGULAR PAGE DESCRIPTION	EPS FILE
banddevice	No	No
clear	Careful	No
cleardictstack	No	No
copypage	No	No
erasepage	Careful	No
executive	No	No
exitserver	No	No
framedevice	No	No
grestoreall	Careful	Careful
initclip	Careful	No

initgraphics	Careful	No
initmatrix	Careful	No
nulldevice	Careful	Careful
quit	No	No
renderbands	No	No
setblackgeneration	Careful	Careful
setcolorrendering	Careful	Careful
setcolorscreen	Careful	Careful
setcolortransfer	Careful	Careful
setflat	Careful	Careful
setglobal	Careful	Careful
setgstate	Careful	Careful
sethalftone	Careful	Careful
setmatrix	Careful	Careful
setoverprint	Careful	Careful
setpagedevice	Careful	No
setscreen	Careful	Careful
setshared	Careful	Careful
setsmoothness	Careful	Careful
settransfer	Careful	Careful
setundercolorremoval	Careful	Careful
startjob	No	No
undefinefont	Careful	Careful
undefineresource	Careful	Careful
statusdict operators	Careful	No
userdict imageable area operators	Careful	No

banddevice Obsolete LanguageLevel 1 device setup operator. It should never be used in a page description.

clear Disrupts nesting of included documents and EPS files. Instead of using **clear**, the application should keep track of which items have been placed on the operand stack and clean up the stack intelligently. If it is necessary to perform the equivalent of a **clear** operation, a count of the objects on the operand stack can be saved at the beginning of the document:

```
count /numstack exch def
```

When it is time to remove all objects the document has left on the operand stack, the following code should be executed:

```
count numstack sub
    {pop}
  repeat
```

cleardictstack Disrupts nesting of included documents and layering of document prologs. Instead of using **cleardictstack**, the application should keep track of which dictionaries have been used and clean up the stack intelligently. If it is necessary to perform the equivalent of a **cleardictstack** operation, a count of the dictionaries present on the stack can be saved at the beginning of the document:

```
/numdict countdictstack def
```

When it is time to remove all dictionaries the document has left on the dictionary stack, the following code should be executed:

```
countdictstack numdict sub
    {end}
  repeat
```

copypage Disrupts operations that depend on page independence. **copypage** is primarily used for debugging and should not appear in a page description. Furthermore, in LanguageLevel 3 **copypage** no longer preserves the page's contents. For multiple copies of a document, use the **#copies** convention or the **NumCopies** page device parameter. The **copypage** operator should not be used to simulate forms functionality; use the **execform** operator (see Section 4.7, "Forms").

erasepage Disrupts nesting of included documents. Normally, it is unnecessary to erase the page explicitly; a program can assume that the page is already erased. However, if necessary, the interior of the current clipping path can be erased with the following code:

```
gsave
    clippath
    1 setgray
    fill
grestore
```

executive Invokes an interactive session. It should never be used in a page description.

exitserver See **startjob**.

framedevice Obsolete LanguageLevel 1 device setup operator. It should never be used in a page description.

grestoreall Discards any graphics state previously established by the document and disrupts nesting of included documents. Instead of **grestoreall**, **gsave** and **grestore** operators should be used in properly balanced pairs. However, **grestoreall** is acceptable if used within a **save**-**restore** pair.

initclip Disrupts nesting of included documents. If the current clipping path in the document must be changed, calls to clipping operators should be surrounded by a **save**-**restore** or **gsave**-**grestore** pair.

initgraphics Disrupts nesting of included documents. If a document requires its graphics state to be initialized, the graphics state should be set explicitly with operators such as **setgray** and **setlinewidth**, surrounded by a **save**-**restore** or **gsave**-**grestore** pair.

initmatrix Disrupts nesting of included documents. If a document requires its CTM to be initialized, the current CTM should be modified (using the **concat** operator), surrounded by a **save**-**restore** or **gsave**-**grestore** pair so that the current CTM is preserved.

nulldevice Installs the "null device" as the current output device. This device produces no physical output, but behaves like a normal device—in other words, the current point is moved, the font machinery is invoked, and so

on. If used carefully, it can be helpful when performing color separations, where knockout control and overprinting are needed. A **save**-**restore** or **gsave**-**grestore** pair around this operator is recommended.

quit Terminates the operation of the interpreter; the document will not be printed. This operator should never be used in a page description.

renderbands Obsolete LanguageLevel 1 device setup operator. It should never be used in a page description.

setblackgeneration See **settransfer**.

setcolorrendering Should be used only in conjunction with **findcolorrendering** to establish the proper color rendering dictionary for a specified rendering intent. Embedding a color rendering dictionary will cause a page description to be device-dependent.

setcolorscreen See **sethalftone**.

setcolortransfer See **settransfer**.

setflat Should be used with caution, since its effects are device-dependent.

setglobal If used improperly, can disrupt page independence and nesting of included documents. In global VM allocation mode, the values of new composite objects are allocated in global VM. Creation and modification of global objects are unaffected by the **save** and **restore** operators.

There are proper uses of **setglobal** and global VM that do not violate page independence or EPS embedding. Global VM can be used to hold data that should be unaffected by **save** and **restore** operators occurring within a page. It can also be used for read-only resources that are loaded by the **findresource** operator on one page and are available for access (also by **findresource**) on subsequent pages.

setgstate Disrupts page independence and nesting of included documents. Proper use of **setgstate** involves resetting a previously obtained graphics state from the **currentgstate** operator. To ensure page independence, the use of

setgstate must not impose a graphics state defined in another page in the document. That is, it should impose a graphics state that is local to that page only. The following example illustrates a proper use of **setgstate**:

```
/oldstate gstate def
    306  3 92  translate
    135  rotate
    5  5  scale
    10 setlinewidth
    … Draw objects in the transformed coordinate system …
oldstate setgstate
    … Draw more objects in the original coordinate system …
```

To obtain a similar effect in a page description, a **save**-**restore** or **gsave**-**grestore** pair should be used instead.

sethalftone Should not normally appear in a page description; it can cause problems if a postprocessor attempts to perform color separations. However, it is appropriate for a system administrator to use **sethalftone** to establish default screening values for the device. The use of **sethalftone** in a page description is device-dependent; the results will vary from one device to another.

sethalftone should not be used to create patterns; the resulting patterns will vary depending on the resolution of the output device. Also, patterns defined by **sethalftone** cannot be color-separated and will appear only on devices that support halftoning. Patterns should be created with the **setpattern** operator or by defining them as characters in a special font.

setmatrix Should be used with a matrix that was previously obtained using the **currentmatrix** operator or its equivalent. It can be used for drawing objects such as ovals, as in the following example:

```
matrix currentmatrix
rx ry scale
0  0  1  0  360  arc
setmatrix
stroke
```

This example ensures that the oval is drawn with an even stroke. However, **setmatrix** should not be used to perform such operations as flipping the coordinate axes. Instead, the **concat** operator should be used to concatenate to the CTM. Ordinarily, PostScript programs should modify the CTM (by using the **translate**, **scale**, **rotate**, and **concat** operators) rather than replace it.

setoverprint Should be used with caution, since its effects are device-dependent.

setpagedevice Can be used to set printer-specific features in a device-independent way. **setpagedevice** establishes a new device, implicitly performing the equivalent of an **initgraphics** operation and an **erasepage** operation. **setpagedevice** must not be used inside an EPS file, as it will erase the entire page in which the EPS file is included. However, in a document page description this operator is often useful in the document setup or page setup section. Documents that want to promote device independence and receive printer rerouting services from a document manager must enclose calls to **setpagedevice** with %%Begin(End)Feature: comments (see the section on requirement body comments in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*).

When using **setpagedevice** at the page level, a program should save the current page device and reestablish it at the end of the page to maintain page independence.

setscreen See **sethalftone**.

setshared See **setglobal**.

setsmoothness Should be used with caution, since its effects are device-dependent.

settransfer Device setup code may have installed a transfer function that is tuned precisely to the device characteristics, and wholesale replacement of the current transfer function can remove any calibration already in place.

Instead, the current transfer function should be *modified*. In the following example, a negative transfer function is concatenated to the current transfer function:

```
[   {1 exch sub} /exec load
    currenttransfer /exec load
] cvx settransfer
```

Even when performed this way, this operation has a device-dependent effect.

setundercolorremoval See **settransfer**.

startjob Should be used only by PostScript programs that perform system administration functions, such as downloading a font program as part of an unencapsulated job to alter initial VM. During execution of an unencapsulated job, VM is not protected. Also, VM resources that the program consumes remain in use until the printer is power-cycled. A program that does call **startjob** should use the %!PS-Adobe-3.0 ExitServer comment (see Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*).

undefinefont See **undefineresource**.

undefineresource Improper use of **undefineresource** can disrupt document manager processing of a document. For example, if the document manager were to perform resource optimization on the document and move the resource within the document file, the **undefineresource** operator could cause that resource to be unavailable for portions of the document.

statusdict operators There are operators defined in the **statusdict** dictionary that are likely to be highly device-dependent—that is, some interpreters will have these operators defined and others will not. These operators must not be used in EPS files. Examples of such operators include, but are not limited to, **setsccbatch**, **duplexmode**, **setpapertray**, **tumble**, and **setmargins**. Documents wanting to promote device independence and receive printer rerouting services from a document manager must enclose the calls to these operators with %%Begin(End)Feature: comments (see the section on requirement body comments in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*).

userdict operators

There are some operators defined in the **userdict** dictionary that cause an imageable region to be defined. Examples of such operators include, but are not limited to, **a4**, **a4small**, **b5**, **ledger**, **legal**, **letter**, **lettersmall**, and **note**. These operators perform the equivalent of an **initgraphics** operation and an **erasepage** operation. They must not be used in an EPS file, as they will erase the entire page that includes the EPS file. However, in a document page description, they are valid in the document setup and page setup sections.

The use of these operators is device-dependent: some interpreters will have them defined and others will not. Documents wanting to promote device independence and receive printer rerouting services from a document manager must enclose the calls to these operators with %%Begin(End)Feature: comments (see the section on requirement body comments in Adobe Technical Note #5001, *PostScript Language Document Structuring Conventions Specification*).

Bibliography

SOME DOCUMENTS LISTED IN THIS BIBLIOGRAPHY are indicated as being available on the Adobe Developer Relations site on the World Wide Web. This site is located at

<http://www.adobe.com/supportservice/devrelations/>

Document version numbers and dates given in this Bibliography are the latest at the time of publication; more recent versions may be found on the Web site.

Other documents are indicated here as being available through the Adobe Developers Association. Developers registered with the ADA receive software development kits, regular electronic mailings, e-mail and telephone support, discounted Adobe software products, and a "members only" Web site containing documentation and sample code not available publicly. For information about joining the ADA, visit the Adobe Developer Relations Web site, send e-mail to ada@adobe.com, or write to

Printing Technologies Developer Support
c/o Adobe Developers Association
345 Park Avenue
San Jose, CA 95110-2704

Resources from Adobe Systems Incorporated

Adobe Type 1 Font Format, Addison-Wesley, Reading, MA, 1990. Explains the internal organization of a PostScript language Type 1 font program. This document is available on the Adobe Developer Relations Web site. Also see Adobe Technical Note #5015, *Type 1 Font Format Supplement*.

Display PostScript System. A suite of manuals containing reference documentation for the Display PostScript system. These documents are available on the Adobe Developer Relations Web site.

Portable Document Format Reference Manual Version 1.2, 27 November 1996. Describes PDF, the native file format of the Adobe Acrobat family of products, and offers suggestions for producing efficient PDF files. This document is available on the Adobe Developer Relations Web site.

PostScript Language Program Design, Addison-Wesley, Reading, MA, 1988. Though this edition of this book describes LanguageLevel 1 only, it is still useful for programmers interested in the effective and efficient design of PostScript programs and printer drivers.

PostScript Language Reference Supplement. A new Supplement is published with each major release of Adobe PostScript software. The Supplements for versions 2011 through the latest version are available on the Adobe Developer Relations Web site. The latest version at the time of publication, *Supplement: PostScript Language Reference Manual (LanguageLevel 3 Specification and Adobe PostScript 3 Version 3010 Product Supplement)*, describes PostScript language extensions that are available in the version 3010 release of Adobe PostScript software.

PostScript Language Tutorial and Cookbook, Addison-Wesley, Reading, MA, 1985. Illustrates the many capabilities of the PostScript language through examples. This edition of this book describes LanguageLevel 1 only and includes some recipes that are no longer recommended by Adobe (in particular, the pattern fill example); nevertheless, it can be a useful learning tool for those who are new to the PostScript language.

Tag Image File Format Specification, Revision 6.0, 3 June 1992. The so-called TIFF standard. Several PostScript language filters use encoding schemes similar to ones included in TIFF. Also, the optional screen preview portion of an EPS file can be in TIFF format. This document is available on the Adobe Developer Relations Web site.

Technical Notes. The following Technical Notes are available on the Adobe Developer Relations Web site:

- *Adobe CMap and CID Font Files Specification Version 1.0*, Technical Note #5014
- *Adobe Communications Protocols Specification*, Technical Note #5009
- *Adobe Font Metrics File Format Specification Version 4.1*, Technical Note #5004
- *CID-Keyed Font Technology Overview*, Technical Note #5092

- *Color Separation Conventions for PostScript Language Programs*, Technical Note #5044
- *The Compact Font Format Specification*, Technical Note #5176
- *Encapsulated PostScript File Format Specification Version 3.0*, Technical Note #5002
- *PostScript Language Document Structuring Conventions Specification Version 3.0*, Technical Note #5001
- *PostScript Printer Description File Format Specification*, Technical Note #5003
- *Type 1 Font Format Supplement,* Technical Note #5015
- *The Type 2 Charstring Format*, Technical Note #5177
- *The Type 42 Font Format Specification*, Technical Note #5012
- *Updates to the PostScript Language Reference Manual, Second Edition*, Technical Note #5085

Other Resources

Fairchild, M., *Color Appearance Models*, Addison-Wesley, Reading, MA, 1997. Covers color vision, basic colorimetry, color appearance models, cross-media color reproduction, and the current CIE standards activities. Updates, software, and color appearance data are available at <http://www.cis.rit.edu/people/faculty/fairchild/CAM.html>.

Farin, G., *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide,* 4th ed., Academic Press, San Diego, 1997. Includes chapters on Coons patches.

Field, G. G., *Color and its Reproduction*, Graphic Arts Technical Foundation, Pittsburgh, 1988. Includes information on trapping algorithms and techniques.

Foley, J. et al., *Computer Graphics: Principles and Practice,* Addison-Wesley, Reading, MA, 1996. (First edition was Foley, J. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.) Covers many graphics-related topics, including a thorough treatment of the mathematics of Bézier cubics and Gouraud shadings.

Hunt, R. W. G., *The Reproduction of Colour*, 5th ed., Fisher Books, England, 1996. A comprehensive general reference on color reproduction; includes an introduction to the CIE system.

Institute of Electrical and Electronics Engineers, Inc. (IEEE), *Standard 754-1985 for Binary Floating-Point Arithmetic*, 1985. May be ordered from IEEE at <http://www.ieee.com>.

International Color Consortium, *ICC Profile Format Specification*. This specification and related documents are available at <http://www.color.org>.

International Electrotechnical Commission (IEC), *Colour measurement and management in multimedia systems and equipment, Part 2: Default RGB color space—sRGB,* 9 January 1998. Available at <http://www.srgb.com/sRGBstandard.pdf>.

International Standards Organization, ISO/IEC 10918-1, *Digital Compression and Coding of Continuous-Tone Still Images*. Informally known as the JPEG standard, for the Joint Photographic Experts Group (the organization that developed the standard). May be ordered from the American National Standards Institute at <http://web.ansi.org>.

International Telecommunication Union (ITU), Recommendations T.4 and T.6. These standards for Group 3 and Group 4 facsimile encoding (which replace those formerly provided in the CCITT *Blue Book*, Volume VII.3) may be ordered from ITU at <http://www.itu.ch>.

Internet Engineering Task Force (IETF) Requests for Comments (RFCs) 1950, *ZLIB Compressed Data Format Specification Version 3.3;* 1951, *DEFLATE Compressed Data Format Specification Version 1.3*; and 2083, *PNG (Portable Network Graphics) Specification Version 1.0*. Available through the RFC Editor home page at <http://www.rfc-editor.org>.

Pennebaker, W., and Mitchell, J., *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1992.

Smith, A., "Color Gamut Transform Pairs," *Computer Graphics* (ACM SIGGRAPH), Volume 12, Number 3, August 1978. Explanation of color conversions between RGB, HSB, and gray levels. In this article, HSB is referred to as hue-saturation-value, with conversions performed according to the "hexcone" model.

Stokes, M., et al., *A Standard Default Color Space for the Internet—sRGB,* 5 November 1996. Available at <http://www.w3.org/Graphics/Color/sRGB> or <http://www.color.org/sRGB.html>.

Warnock, J. and Wyatt, D., "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics* (ACM SIGGRAPH), Volume 16, Number 3, July 1982. Technical background for the Adobe imaging model.

Wyszecki, G. and Styles, W., *Color Science: Concepts and Methods, Quantitative Data and Formulae*, 2nd ed., John Wiley and Sons, New York, 1982. A detailed reference on color theory.

INDEX

> (angle bracket, right)
  as EOD indication (**ASCIIHexDecode**) 130
<> (angle brackets)
  as hexadecimal string delimiter 30
  special syntactic treatment of 27
<<>> (angle brackets, double)
  compared with other uses of angle brackets 33
  literal dictionary object created by 36
<< (angle brackets, double left) operator
  dictionary constructor role 33, 54
  mark objects created by 44, 525
>> (angle brackets, double right) operator
  as dictionary constructor 33, 525
  dictionary constructor role 54
@ (at sign)
  as internal PostScript operator prefix 43
\ (backslash)
  as escape character in text strings 29
\\ (backslash, double)
  as ASCII encoding for backslash character *(list)* 29
\b (backslash b)
  as ASCII encoding for backspace character *(list)* 29
\f (backslash f)
  as ASCII encoding for form feed *(list)* 29
\((backslash left parenthesis)
  as ASCII encoding for left parenthesis character *(list)* 29
\n (backslash n)
  as ASCII encoding for line feed *(list)* 29
\) (backslash right parenthesis)
  as ASCII encoding for right parenthesis character *(list)* 29
\r (backslash r)
  as ASCII encoding for carriage return *(list)* 29
\t (backslash t)
  as ASCII encoding for tab character *(list)* 29
{} (curly brackets)
  executable arrays enclosed by 36
  as procedure delimiter 32
  special syntactic treatment of 27
= (equals) operator 87, 526
  == operator compared with 87
  printing operand stack contents using **stack** 87
== (equals, double) operator 87, 526
  = operator compared with 87
  printing operand stack contents using **pstack** 87
() (parentheses)
  ASCII encoding *(list)* 29
  as literal text string delimiters 29
((parenthesis, left)
  special syntactic treatment of 27
) (parenthesis, right)
  special syntactic treatment of 27
% (percent sign)
  comments initiated by 27
  special syntactic treatment of 27
/ (slash)
  as literal name delimiter 31
  special syntactic treatment of 27
  as valid literal name 31
// (slash, double)
  as immediately evaluated name delimiter 31
//name syntax
  immediately evaluated name 121–123, 167
    specifying in a binary object sequence 169
[] (square brackets) operator
  creating arrays with 54
  literal array object created by 36
[(square bracket, left) operator
  array construction role 32
  mark objects created by 44, 524
  special syntactic treatment of 27
] (square bracket. right) operator
  See also array(s)
  as array constructor 32, 524
  special syntactic treatment of 27
~ (tilde)
  as ASCII base-85 string delimiter 31
~> (tilde angle bracket)
  as EOD indication
    ASCII85Decode 131
    ASCII85Encode 131
<~~> (tilde angle brackets)
  as ASCII base-85 strings delimiter 31

A

abs operator 52, 527
absolute value calculation
  *See **abs** operator*
AbsoluteColorimetric rendering intent 470
accents
  See character(s)
access
  attributes 37
    setting (**executeonly**) 584
    setting (**noaccess**) 628
    setting (**readonly**) 640
    testing read (**rcheck**) 638
    testing write (**wcheck**) 718
  conventions, for files 79
  execute-only, as object attribute 37
  invalid
    errors (**invalidaccess**) 60, 80, 616
    errors (**invalidfileaccess**) 80, 617
  as object attribute, categories of 37
  read-only, as object attribute 37
  strings, for files *(table)* 79
  unlimited, as object attribute 37
accessing
  See also retrieving
  dictionaries, methods for 41
  fonts
    CFF and Chameleon, in a PostScript program 345–346
    findfont 315, 323
    metrics 332
  stacks
    dictionary 46
    execution 46
    operand 46
    operand, as if it were an array (**index**) 52, 610
AccurateScreens user parameter
  as halftone setting user parameter 757
  halftone-setting operator impact 482
  as type 1 halftone dictionary entry *(table)* 487
  value *(table)* 749
ACFM (Adobe composite font metrics) files 333
achromatic color spaces 228–231
ActualAngle entry
  as type 1 halftone dictionary entry *(table)* 488
ActualFrequency entry
  as type 1 halftone dictionary entry *(table)* 487
add operator 52, 527
  as a function *(example)* 106
  immediate execution use 47
addglyph operator 528
  Type 4 CIDFont use 381
adding
  glyphs 351–352
  numbers (**add**) 527
additive color
  devices, render color space use 461
  models (RGB and HSB) 217
adjusting
  pages 437
  strokes, automatically 503–504
AdvanceDistance page device parameter 413
AdvanceMedia page device parameter 412
AFM (Adobe font metrics) files 333
algorithms
  See also guidelines; rules
  ASCII base-85 131
  bit-oriented encoding
    *See **CCITTFaxEncode** filter*
  CIE-based color conversion to device color 459
  device color space conversion 474
  font mapping
    composite fonts 358–362
    FMapType selection of 358–360
    FMapType *(table)* 92
    nesting rules 363–364
    (table) 360
  image compression
    See DCT (discrete cosine transform)
  image interpolation 301
  LZW (Lempel-Ziv-Welch) adaptive compression method
    *See **FlateDecode** filter; **FlateEncode** filter; **LZWDecode** filter; **LZWEncode** filter*
  mapping, CMap 388–389
  PNG predictor 139
  run-length encodin
    *See **RunLengthDecode** filter; **RunLengthEncode** filter*
  scan conversion 501, 503
aliasing
  causes, in color transitions 263
  preventing, in shading patterns, **AntiAlias** *(table)* 262
All colorant name
  Separation color space use 243
allocation
  VM
    by composite object constructors 58
    policies (**Font**) 105
    retrieving (**currentglobal**) 557
    setglobal control of 60
    setting (**setvmthreshold**) 688
aload operator 54, 528
anchorsearch operator 55, 529
and operator 55, 529

angle bracket, right (>)
as EOD indication (**ASCIIHexDecode**) 130
angle brackets (<>)
in ASCII base-85 strings delimiters 31
as hexadecimal string delimiter 30
special syntactic treatment of 27
angle brackets, double (<<>>) operators
compared with other uses of angle brackets 33
literal dictionary object created by 36
angle brackets, double left (<<) operator
dictionary constructor role 33, 54
mark objects created by 525
angle brackets, double right (>>) operator
as dictionary constructor 33, 525
dictionary constructor role 54
Angle entry
as type 1 halftone dictionary entry *(table)* 487
AntiAlias entry
as shading dictionary entry *(table)* 262
appending
characters (**write**) 720
curves
curveto 564
rcurveto 638
glyphs (**charpath**) 319
lines
lineto 622
rlineto 650
user paths (**uappend**) 706
applications
PostScript interpreter interactions 15–22
userdict and **globaldict** dictionaries use by 66
arc
adding to current path
arc 191, 530–531
arcn 191, 531–532
arct 191, 532–533
arcto 191, 534
arc operator 177, 191, 530–531
operation code for encoded user paths *(table)* 201
user path 198
structuring 199
architecture
CID-keyed font 364
limits *(table)* 739
arcn operator 191, 531–532
operation code for encoded user paths *(table)* 201
user path 198
structuring 199
arct operator 191, 198, 532–533
operation code for encoded user paths *(table)* 201
arcto operator 191, 534
reasons why it is not a user path operator 198
areas
painting (**fill**) 177
path-enclosed
even-odd rule 196
nonzero winding number rule 195
arithmetic
See also mathematics
addition (**add**) 527
division
div 574
idiv 605
errors (**undefinedresult**) 709
multiplication (**mul**) 627
negation (**neg**) 628
operators
immediate execution use *(example)* 46–47
(list) 508
overview 52
remainder (**mod**) 627
square root (**sqrt**) 692
subtraction (**sub**) 701
array operator 534
as composite object constructor 58
creating arrays with 53
array(s) 38
See also matrix
accessing operand stack as if it were (**index**) 52, 610
binary object sequence encoding 166
as composite object *(table)* 34
constructing
counttomark 551
square brackets use 32
copying 39
copy 53, 548
creating
[] 54, 524
array 53, 534
elements
loading onto the operand stack (**aload**) 54, 528
retrieving (**get**) 53, 598
retrieving the number of (**length**) 53, 621
storing objects into (**astore**) 54
storing (**put**) 53, 635
storing the operand stack into (**astore**) 535
encoding 94
executable
building user paths as 204
execution handling 50
interpreter handling of *(example)* 48
semantics of 36

array(s) *(continued)*
executable *(continued)*
with immediate execution, binary object sequence as 163
expanding values of (==) 87
heterogeneous composition permitted in 38
homogeneous number
as binary tokens 161–162
encoded number string use 171
literal array object 162
mapping a procedure over (**forall**) 53, 597
operators
(list) 509
overview 52–55
that apply only to 54
packed
See packed arrays
procedure
objects as 33
semantics compared with 32
strings compared with 39
subintervals of
creating objects that share (**getinterval**) 53, 599
overwriting (**putinterval**) 53, 636
syntax 32
threshold 489–497
type 3 halftone dictionaries 490
type 6 halftone dictionaries 491–492
type 10 halftone dictionaries 492–495
type 16 halftone dictionaries 495–497
ASCII
See also text
base-85
decoding binary data from (**ASCII85Decode**) 130
decoding binary data from (**ASCII85Decode**) *(overview table)* 85
encoding 31
encoding, algorithm description 131
encoding binary data as (**ASCII85Encode**) 131
encoding binary data as (**ASCII85Encode**) *(overview table)* 85
decoding, filters, overview 84
encoding
arrays 32
comments 27
dictionaries 33
filters, overview 84
names 31
numbers 28
procedures 32–33
standard character set use 26–34
strings 29
strings, ASCII base-85 31
strings, ASCII base-85 algorithm description 131
ASCII *(continued)*
encoding *(continued)*
strings, hexadecimal 30–31
strings, hexadecimal, reading from an input file (**readhexstring**) 73
strings, hexadecimal, writing to an output file (**writehexstring**) 73
strings, literal text 29–30
hexadecimal
decoding binary data from (**ASCIIHexDecode**) *(overview table)* 85
encoding binary data as (**ASCIIHexEncode**) *(overview table)* 85
as PostScript language encoding 25
tokens, binary tokens compared with 158
ASCII85Decode filter 84, 85, 98, 130
ASCII85Encode filter 85, 131–132
ASCIIHexDecode filter 82, 85, 130
ASCIIHexEncode filter 82, 85, 130
ashow operator 320, 534–535
Asian character sets
CID-keyed font use 364–390
associating keys with dictionary values
def 54, 568
astore operator 54, 535
ostack array relationship to *(table)* 116
AsyncRead entry
precautions 154
as **ReusableStreamDecode** dictionary entry *(table)* 155
at sign (@) 43
atan operator 52, 535–536
atob utility
ASCII base-85 encoding compared with 132
atomic objects
See simple objects
attribute(s)
access 37
file objects 79
of composite objects 37
setting (**executeonly**) 584
setting (**noaccess**) 628
setting (**readonly**) 640
testing read (**rcheck**) 638
testing write (**wcheck**) 718
executable 36
execution of objects with 50–51
precautions 36
setting (**cvx**) 568
testing for (**xcheck**) 721
of an image 289
literal 36
handling objects with 50

attribute(s) *(continued)*
literal *(continued)*
setting (**cvlit**) 566
testing for (**xcheck**) 721
matching requests with, in media selection 403–407
of objects 35–37
operators
(list) 512
overview 56
awidthshow operator 320, 536
axial shading pattern 261, 266–268

B

Background array
as shading dictionary entry *(table)* 262
backing up
characters, control characters for interactive executive use *(table)* 21
backslash (\)
as escape character in text strings 29
backslash, double (\\)
as ASCII encoding for backslash character *(list)* 29
backslash b (\b)
as ASCII encoding for backspace character *(list)* 29
backslash f (\f)
as ASCII encoding for form feed *(list)* 29
backslash left parenthesis (\()
as ASCII encoding for left parenthesis character *(list)* 29
backslash n (\n)
as ASCII encoding for line feed *(list)* 29
backslash r (\r)
as ASCII encoding for carriage return *(list)* 29
backslash right parenthesis (\))
as ASCII encoding for right parenthesis character *(list)* 29
backslash t (\t)
as ASCII encoding for tab character *(list)* 29
backspace (BS)
interactive executive use *(table)* 21
backspace character
\b as ASCII encoding *(list)* 29
banddevice operator
usage guidelines 804
bandwidth
as scarce resource, binary tokens suitable for 156
base fonts 321, 357
See also font(s)
character encoding scheme 328–330
dictionary entries common to all *(table)* 325
base-85 ASCII
decoding binary data from (**ASCII85Decode**) *(overview table)* 85, 130
encoding 31
algorithm description 131
binary data as (**ASCII85Encode**) 85, 131
BBox array
See also bounding box
as form dictionary entry *(table)* 208
as shading dictionary entry *(table)* 262
as type 1 pattern dictionary entry *(table)* 251
begin operator 54, 536
defineresource use of 100
dictionary, stack modifiable by 46
beginbfchar operator 384, 537
beginbfrange operator 384, 537
begincidchar operator 385, 537
begincidrange operator 385, 537
begincmap operator 384, 537
begincodespacerange operator 384, 537
beginnotdefchar operator 385, 538
beginnotdefrange operator 385, 538
BeginPage page device parameter
device initialization *(table)* 427
operations 427–428
beginrearrangedfont operator 384, 538
beginusematrix operator 385, 538
behavior
communication channels 76
of objects, attributes impact on 35
Bernstein polynomials 285
Bézier
control points, type 6 shading pattern data source use 280
curves
adding segment to current path (**arcto**) 191
appending to current path (**curveto**) 564
appending to current path (**rcurveto**) 638
in Coons patch meshes 277
in tensor-product patch meshes 284
bilevel picture encoding 145
bilinear interpolation 111
binary
encoding, details 156–174
object sequences 163–168
binary tokens compared with 158, 169
encoded system names in 168–169
setting format (**setobjectformat**) 677
writing to standard output (**printobject**) 634
writing (**writeobject**) 721
radix number representation of 28
representation of integers, shifting bits in (**bitshift**) 539

binary *(continued)*
tokens 156, 158–162
ASCII tokens compared with 158
binary object sequences compared with 169
encoded number strings as 171–172
encoded system names in 168–169
interpretation *(table)* 158–159
binary data
ASCII base-85 strings encoding of 31
ASCII-encoded hexadecimal strings
reading from an input file (**readhexstring**) 73
writing to an output file (**writehexstring**) 73
decoding
ASCII85Decode *(overview table)* 85
ASCIIHexDecode *(overview table)* 85
encoding
ASCII85Encode *(overview table)* 85
ASCIIHexEncode *(overview table)* 85
of font sets 343
fonts (**StartData**) 694
handling of 76
producing ASCII base-85 encoded data from **ASCII85Encode** 131
producing from ASCII base-85 encoded data **ASCII85Decode** 130
representation
ASCII encoding/decoding filters use for 84
ASCII-encoded hexadecimal strings 30–31
binary entry
as **$error** dictionary entry *(table)* 117
bind operator 538–539
benefits of using 118
early binding of names with 118–121
eligibility requirements for use of 119
IdiomSet resource category use 97
immediate evaluation of names by 122
user path use 199
binding 117
late 117
names
bind 538–539
early binding of (**bind**) 117–123, 118–121, 538–539
bit-oriented encoding
CCITT fax standard as, implications for **LowBitFirst** key 146
data compression with **CCITTFaxEncode** *(table)* 86
data decompression with **CCITTFaxDecode** *(table)* 86
LZW as, implications for **LowBitFirst** key 136
BitmapFontInit procedure set
LanguageLevel 3 operators defined in *(table)* 726
as standard procedure set in LanguageLevel 3 *(table)* 96
Type 4 CIDFont operators defined in 381
bitmaps (glyph)
in a Type 4 CIDFont 379–382
BitmapFontInit *(table)* 96
stencil masking use for painting 303
bitshift operator 55, 539
BitsPerComponent entry
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
as image data dictionary entry *(table)* 306
as image mask dictionary entry *(table)* 306
as LZW dictionary entry *(table)* 134
as predictor-related entry in LZW and Flate filter dictionaries *(table)* 141
as type 1 image dictionary entry *(table)* 298
as type 4 image dictionary entry *(table)* 308
as type 4 shading dictionary entry *(table)* 270
as type 5 shading dictionary entry *(table)* 275
as type 6 shading dictionary entry *(table)* 279
BitsPerCoordinate entry
as type 4 shading dictionary entry *(table)* 270
as type 5 shading dictionary entry *(table)* 275
as type 6 shading dictionary entry *(table)* 279
BitsPerFlag entry
as type 4 shading dictionary entry *(table)* 270
as type 6 shading dictionary entry *(table)* 279
BitsPerSample entry
(example) 111
as type 0 function dictionary entry *(table)* 109
bitwise
and operation (**and**) 529
bit shifting (**bitshift**) 539
exclusive or operation (**xor**) 722
inclusive or operation (**or**) 631
not operation (**not**) 629
operators
(list) 511
overview 55
black
diffuse black point (**BlackPoint**) *(table)* 224, 230, 465
generation
See black generation
pixels, encoding (**BlackIs1**) *(table)* 145
trapping rule 451–452
black generation 476–477
as graphics state parameter *(table)* 180
retrieving function (**currentblackgeneration**) 552
setting function (**setblackgeneration**) 476, 658
usage guidelines (**setblackgeneration**) 806
BlackColorLimit entry
Trapping dictionary, entries *(table)* 448
BlackDensityLimit entry
Trapping dictionary, entries *(table)* 448

BlackIs1 entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
BlackPoint array
as **CIEBasedA** dictionary entry *(table)* 230
as **CIEBasedABC** dictionary entry *(table)* 224
as type 1 CIE-based CRD entry *(table)* 465
BlackWidth entry
Trapping dictionary, entries *(table)* 448
blend circles
radial shading color blend based on 268–270
boolean
and operator (**and**) 529
binary object sequence encoding 166
exclusive or operator (**xor**) 722
false value (**false**) 587
inclusive or operation (**or**) 631
not operator (**not**) 629
objects 38
operators
(list) 511
overview 55
as simple object *(table)* 34
true value (**true**) 704
bounding box
See also clipping
BuildGlyph procedure use 339
current path, retrieving (**pathbox**) 631
fonts
accessing (**FontBBox**) 333
FontBBox *(table)* 325, 370
form (**BBox**) *(table)* 208
glyph
FontBBox 331
setting (**setcachedevice**) 659
setting (**setcachedevice2**) 659
page device (**ImagingBBox**) 414
retrieving, for current path (**pathbox**) 631
setting
operation code for encoded user paths (**setbox**) 201
setbox 198, 657
ucache required to precede *(example)* (**setbox**) 202–203
user path structuring (**setbox**) 199
shading pattern (**BBox**) *(table)* 262
tiling pattern (**BBox**) *(table)* 251
user path, encapsulated with the path 198
Bounds array
as type 3 function dictionary entry *(table)* 114
brackets
angle (<>)
special syntactic treatment of 27
use in delimiting ASCII base-85 strings 31
brackets *(continued)*
angle brackets
double (<<>>) operators 33, 36
double left (<<) operator 33, 54, 525
double right (>>) operator 33, 54, 525
curly ({}), as procedure delimiters 27, 32, 36
square bracket
left ([) operator 27, 32, 44, 524
right (]) operator 27, 32, 524
brightness 217
BS (backspace)
interactive executive use *(table)* 21
btoa utility
ASCII base-85 encoding compared with 132
buffer flushing
flush 594
resetfile 645
buffering
of characters, **flush** and **flushfile** operations on 74
decoding filter impact 128
encoding filter impact 128
filter use of 127
BuildChar procedure 340–341
as Type 3 font dictionary entry *(table)* 338
BuildGlyph procedure 338–340
CharStrings compared with 351–352
as Type 1 CIDFont dictionary entry *(table)* 377
as Type 3 font dictionary entry *(table)* 338
building
See creating
BuildTime system parameter 751
byte(s) 157
characters synonymous with, file operator treatment as 73
order 170
ByteOrder system parameter
machine representation of byte order 170
value *(table)* 751
bytesavailable operator 87, 539
ReusableStreamDecode filter use 154

C

C0 array 113
C1 array 113
cache
font 333–335
currentcacheparams 552
incremental definition of font impact 354
loading glyphs into (**addglyph**) 528
setting parameters (**setcacheparams**) 661
setting (**setcachedevice**) 659
setting (**setcachedevice2**) 659

cache *(continued)*
font *(continued)*
setting size (**setcachelimit**) 660
status (**cachestatus**) 540
Type 4 CIDFont advantages 380
pattern cell 250
user path 202–204
output placed in 198
retrieving status (**ucachestatus**) 707
setting (**setucacheparams**) 686
setting (**ucache**) 707
cachestatus operator 540
caching forms 207, 209
capacity
See dictionaries, capacity
carriage return (CR)
as EOL, scanner handling of 74–75
numeric values *(table)* 27
\r as ASCII encoding *(list)* 29
carriage return-line feed (CR-LF)
as EOL, scanner handling of 74–75
case sensitivity of names 40
categories, resource
See resource(s), categories
Category resource category
all resource categories contained within 99
as category implementation dictionary entry *(table)* 101
creating new resource categories *(example)* 99
resource file loading of 105
as resource for defining new resource categories *(table)* 92
CCITT facsimile standard
data compression with **CCITTFaxEncode** *(table)* 86
data decompression with **CCITTFaxDecode** *(table)* 86
defined by CCITT (Comité Consultatif International Téléphonique et Télégraphique) 143
CCITTFaxDecode filter 86, 143–146
dictionary entries *(table)* 144–145
CCITTFaxEncode filter 86, 143–146
dictionary entries *(table)* 144–145
CDevProc procedure
modifying glyph metrics with 350–351
as Type 0 CIDFont dictionary entry *(table)* 374
as Type 1 font dictionary entry *(table)* 326
as Type 2 CIDFont dictionary entry *(table)* 379
as Type 42 font dictionary entry *(table)* 347
CE encoding vector
(table) 773, 786
ceiling operator 52, 540
cell
halftone 482
cell *(continued)*
pattern 248
key 252
tiling an area with 250
CFF (Compact Font Format)
FontSet resource category 94
Type 2 font type 343–346
Chameleon (Type 14) font type 343–346
changing
See modifying
character(s) 313
See also font(s); glyph(s)
accents, recommended encoding 350
backing up, control characters for interactive executive use *(table)* 21
bitmap character, stencil masking use for painting 303
buffered, **flush** and **flushfile** operations on 74
bytes synonymous with, file operator treatment as 73
CMap code mapping resource *(table)* 91
code mapping dictionaries (**CMap**) 94
codes (**ISOLatin1Encoding**) 618
collection 365
CIDSystemInfo specification of 367
control
communications use of 26
control characters for interactive executive use *(table)* 21
echoing, enabling/disabling (**echo**) 575
encoding 328–330
erasing, control characters for interactive executive use *(table)* 21
mapping 358–362
multiple-byte encodings, CID-keyed font use 364–390
names
font dictionary access through 328
portability advantages over character codes 330
newline 26
handling 30
octal character codes 30
painting (**show**) 177, 329
reading (**read**) 73, 639
selector 335
set
encoding vectors and *(appendix)* 773–794
Expert *(table)* 787–789
standard Latin *(table)* 779–783
Symbol *(table)* 792–793
stream, as object source 24
string objects use for holding 39
undefined, handling 389–390
white-space 26
(table) 27
writing, to an output file (**write**) 73, 720

charpath operator 540–541
appending glyphs to current path with 319
reasons why it is not a user path operator 198
charstring 329
CFF based on 343
Type 0 CIDFont use 371
CharStrings dictionary
adding or replacing glyphs in 351–352
character names defined in 330
as Type 1 font dictionary entry *(table)* 326
as Type 2 CIDFont dictionary entry *(table)* 378
as Type 42 font dictionary entry *(table)* 346
chroma-key
color key masking compared with 307
CID (character identifier)
See CID-keyed fonts
CID-keyed fonts 364–390
basics of 365–367
CIDFont dictionary 368–382
CIDSystemInfo dictionary *(table)* 368
CMap resource category 94
creating 387–390
composefont 545–546
CSL use 365
dictionary, **CIDSystemInfo** dictionary entry 367–368
relationship among components *(figure)* 367
resource category 94
Type 0 371–376
bibliographic reference 371
CIDInit *(table)* 96
dictionary *(table)* 373
GlyphDirectory 375–376
Type 1 376–377
Type 2 377–379
dictionary *(table)* 378
GlyphDirectory 379
Type 4 379–382
BitmapFontInit *(table)* 96
operators for 381–382
CIDCount entry
as Type 0 CIDFont dictionary entry *(table)* 373
as Type 2 CIDFont dictionary entry *(table)* 378
CIDFont resource category 94
bibliographic reference 366
CID-keyed font use 364, 366
CIDFont dictionaries 368–382
(table) 370
cshow 551
CIDFont resource category
(table) 91
CIDFontName entry
as **CIDFont** dictionary entry *(table)* 370
CIDFontType entry
as **CIDFont** dictionary entry *(table)* 370
values *(table)* 370
CIDInit procedure set 384–385
See also CMap (character map), operators
LanguageLevel 3 operators defined in *(table)* 726
as standard procedure set in LanguageLevel 3 *(table)* 96
CIDMap entry
as Type 2 CIDFont dictionary entry 378
CIDMapOffset entry
as Type 0 CIDFont dictionary entry *(table)* 373
CIDSystemInfo dictionary 367–368
as **CIDFont** dictionary entry *(table)* 370
as CMap dictionary entry *(table)* 383
(table) 368
CIE
as Commission Internationale de l'Éclairage standard organization 214
CIE-based color spaces 220–238
color rendering dictionaries
customizing 472–473
setting (**setcolorrendering**) 663
type 1 460–473, 461–468
type 1 *(table)* 463
converting to device colors 459–473
steps involved 461
Decode array mapping 300
gamut mapping function, gamma correction contrasted with 478
operators (**currentcolorrendering**) 553
overview 214
page device color support parameter (**UseCIEColor**) 422
remapping device colors to 237–238
rendering
findcolorrendering 591
intents 469–473
overview 457
rules for shading patterns 264
CIEBasedA color space 228–231
as CIE-based color space family 214
dictionary *(table)* 229
CIEBasedABC color space 221–228
as CIE-based color space family 214
CIEBasedDEF and **CIEBasedDEFG** relationship 233
ColorSpaceFamily resource category 98
dictionary *(table)* 223–224
CIEBasedDEF color space 232–237
as CIE-based color space family 214
dictionary *(table)* 233
CIEBasedDEFG color space 232–237
as CIE-based color space family 214
dictionary *(table)* 235

circle
See arc
circular queue
treating stack portion as (**roll**) 52, 650
clear operator 52, 541
usage guidelines 804
cleardictstack operator 54, 541
dictionary stack modifiable by 46
usage guidelines 804
clearing operand stack
clear 52
elements above the highest mark (**cleartomark**) 52, 541
cleartomark operator 44, 52, 541
clip operator 178, 193, 542
clippath operator 193, 542–543
clipping
See also bounding box
even-odd (**eoclip**) 579
of function values 107
in imaging model 176
LanguageLevel 1 use for simulation of masking 302
operators
clip 542
clippath 542–543
cliprestore 543
clipsave 543–544
initclip 611
rectclip 641
path 178, 192–193
as graphics state parameter *(table)* 179
computing a new path (**clip**) 193
current, insideness testing disregard of 197
operators 193
restoring (**cliprestore**) 193
retrieving (**clippath**) 193
saving (**clipsave**) 193
setting (**initclip**) 611
stack 192
stack, as graphics state parameter *(table)* 179
stack, as one of five execution state stacks 45
rectangles (**rectclip**) 641
scan conversion 503
user paths 205
cliprestore operator 192, 193, 543
clipsave operator 192, 193, 543–544
closefile operator 80, 544
CloseTarget and **CloseSource** use by 129
with an encoding filter 128
ReusableStreamDecode filter use 154
closepath operator 177, 191, 198, 544
operation code for encoded user paths *(table)* 201
subpath closing 190, 194
CloseSource entry
ASCII85Decode use of 131
ASCIIHexDecode use of 130
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
decoding filters with 129
filter file closing 124
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
as LZW dictionary entry *(table)* 134
as **ReusableStreamDecode** dictionary entry *(table)* 155
as **SubFileDecode** dictionary entry *(table)* 152
CloseTarget entry
ASCII85Encode use of 132
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
as **DCTEncode** dictionary entry *(table)* 150
encoding filters with 129
filter file closing 124
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
as LZW dictionary entry *(table)* 134
closing
files 79–80
closefile 80, 544
restore 80
paths (**closepath**) 544
subpaths (**closepath**) 191, 194
CMap (character map)
bibliographic reference 366
dictionaries 382–387
as font mapping algorithm *(table)* 360
as Type 0 font dictionary entry *(table)* 358
CID-keyed font use 364
CIDSystemInfo dictionary entry 367–368
(example) 385–387
operators in the **CIDInit** procedure set 384–385
(table) 383
mapping algorithm 388–389
operators 384–385
beginbfchar 537
beginbfrange 537
begincidchar 537
begincidrange 537
begincmap 537
begincodespacerange 537
beginnotdefchar 538
beginnotdefrange 538
beginrearrangedfont 538
beginusematrix 538
endbfchar 577
endbfrange 577
endcidchar 577
endcidrange 578
endcmap 578

CMap (character map) *(continued)*
operators *(continued)*
endcodespacerange 578
endnotdefchar 578
endnotdefrange 579
endrearrangedfont 579
endusematrix 579
usecmap 712
specification (**usecmap**) 712
CMap dictionary
as Type 0 font dictionary entry *(table)* 358
CMap resource category 94, 388–389
(table) 91
CMapName entry
as CMap dictionary entry *(table)* 383
CMapType entry
as CMap dictionary entry *(table)* 383
CMapVersion entry
as CMap dictionary entry *(table)* 383
CMYK (cyan-magenta-yellow-black)
calibrated, **CIEBasedDEF** and **CIEBasedDEFG** use 232
color extensions, LanguageLevel 2 *(list)* 733
DeviceCMYK color space 211, 218–219
conversion between **DeviceGray** color space and 475
conversion from **DeviceRGB** to 475–477
conversion to **DeviceRGB** from 477
currentcmykcolor 215, 219
remapping to CIE-based color space 237–238
setcmykcolor 219
setcolor 219
setcolorspace use 219
operators
currentcmykcolor 553
setcmykcolor 662
setcmykcolor 194
as subtractive color model 218
CodeMap entry
as CMap dictionary entry *(table)* 383
codes
ASCII character set 26–27
character (**ISOLatin1Encoding**) 618
font encoding
for base fonts 328–330
for CID-keyed fonts 366
for composite fonts 356–362
Collate page device parameter 417
color
CIE-based, converting to device color 459–473
corrections, remapping device colors to CIE-based color space use 237
data
compressing (**DCTEncode**) *(table)* 86
color *(continued)*
data *(continued)*
decompressing (**DCTDecode**) *(table)* 86
device, converting CIE-based color to 459–473
different, tiling with, uncolored tiling pattern use 257
duotone 245
facilities, overview 210
function use 106
glyph, not retained in font cache 334
glyph painting, setting 318
as graphics state parameter *(table)* 179
gray, setting (**setgray**) 671
halftone dictionaries, with multiple color components 497–499
halftone screen definition (**setcolorscreen**) 481
high-fidelity 245
images
colorimage 544–545
compression, DCT algorithm impressive with 147
with a single source *(example)* 310
key masking, of images 307–308
lines, setting (**sethsbcolor**) 194
mapping
functions, as CIE-based color rendering dictionary component 460
sample data to 291
maps, selecting colors from, **Indexed** color space use 239–241
media (**MediaColor**) 401
models
HSB 217
process colors (**ProcessColorModel**) 420, 422–424
RGB 217
specification for rendering 419
subtractive, CMYK 218
subtractive, tints as, in **Separation** color spaces 242
multitone 245
patterns 248–287
colored tiling patterns 254–257
specifying *(table)* 251
uncolored tiling patterns 257–259
process 218, 241
rendering
as graphics state parameter *(table)* 180
CIE-based 460–461
CIE-based, customizing 472–473
CIE-based, Type 1 461–468
(figure) 213
overview 210
setting (**setcolorrendering**) 663
setting (**setcolor**) 662
spaces
See color spaces
specification
(figure) 212

color *(continued)*
specification *(continued)*
overview 210
spot 241
support for
page device parameters 419–426
page device parameters *(table)* 420–422
transfer functions
retrieving (**currenttransfer**) 563
setting (**setcolortransfer**) 479, 666
setting (**settransfer**) 479
TransferFunction in halftone dictionary 488, 490, 491, 495
transition
See gradient
values 210
operators 214–216
semantics of 211–248
vision, CIE-based color spaces modeling of 220–238, 459
wheel, HSB represented as 218
color spaces 210–248
as graphics state parameter *(table)* 179
CIE-based
See CIE-based color spaces
conversion between **DeviceRGB** and **DeviceGray** 474–475
currentcolorspace 554
device 216–220
conversion of CIE-based color to 461
conversions among 473–478
DeviceCMYK 218–219
DeviceGray 211, 219–220
DeviceGray setting (**setgray**) 671
DeviceN 245–247
DeviceRGB 217–218
overview and operators 211
Separation 241–245
DeviceCMYK 218–219
DeviceGray 211, 219–220
DeviceGray setting (**setgray**) 671
DeviceN 245–247
DeviceRGB 217–218
families of 211–214
images and 296–297
Indexed 239–241
native 473
operators 214–216
Pattern 238–239, 254
uncolored tiling pattern use 257
pattern, selecting 249
process color model contrasted with 423
relationships among *(figure)* 212
remapping **DeviceGray** to CIE-based 237–238
color spaces *(continued)*
render 461
retrieving (**currentrgbcolor**) 561–562
Separation 241–245
setcmykcolor 211
setcolorspace 211
setgray 211
sethsbcolor 211
setrgbcolor 211
setting
setcmykcolor 662
setcolorspace 665
sethsbcolor 672
setrgbcolor 681
special 238–247
special shading considerations 263–264
types of 210–216
underlying
for colored tiling pattern 254
for uncolored tiling pattern 257
ColorantDetails dictionary
as **TrappingDetails** dictionary entry *(table)* 442
ColorantDetails page device parameter
dictionary entries *(table)* 443
colorants
available, determining for a **Separation** color space 243
device, separations and 424–426
InkParams resource category *(table)* 91
managing, **Separation** color space use 241
separations compared with 242
spot 424
zone-specific colorant details 454
ColorantSetName entry
as **ColorantDetails** dictionary entry *(table)* 443
ColorantZoneDetails dictionary
entries *(table)* 454
as **Trapping** dictionary entry, entries *(table)* 449
colorimage operator 544–545
color space
conflicts with **Pattern** color space 254
use 296
images as color value source 211
sample
data sources 291
representation 290
uncolored tiling pattern prohibited from using 257
ColorRendering procedure set
customizing CRD selection use 472
LanguageLevel 3 operators defined in *(table)* 726
as standard procedure set in LanguageLevel 3 *(table)* 96
ColorRendering resource category 96
(table) 91

ColorRenderingType entry
as type 1 CIE-based CRD entry *(table)* 463
ColorRenderingType resource category 98
Colors entry
as **DCTEncode** dictionary entry *(table)* 148
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
as **LZWEncode**/**LZWDecode** dictionary entry *(table)* 134
as predictor-related entry in LZW and Flate filter dictionaries *(table)* 141
ColorSpace entry
as shading dictionary entry *(table)* 262
ColorSpace resource category 96
(table) 91
ColorSpaceFamily resource category 98
ColorTransform entry
as **DCTEncode** dictionary entry *(table)* 149
Columns entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 144
as **DCTEncode** dictionary entry *(table)* 148
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
as **LZWEncode**/**LZWDecode** dictionary entry *(table)* 134
as predictor-related entry in LZW and Flate filter dictionaries *(table)* 141
command array
as **$error** dictionary entry *(table)* 116
comment syntax 27
communication
bandwith as scarce resource, binary tokens suitable for 156
channel behavior 76
Compact Font Format (CFF)
Type 2 font type 343–346
comparing objects
for equality (**eq**) 580
for greater-than-or-equal relation (**ge**) 598
for greater-than relation (**gt**) 604
for inequality (**ne**) 628
for less-than-or-equal relation (**le**) 620
for less-than relation (**lt**) 623
compatibility
CIDFont issues, when composite font descendant 369
CIDSystemInfo requirements 368
CSL use as CID-keyed font handler 365
LanguageLevels, incompatibilities among 735
operators
currentshared 562
scheck 655
setshared 682
shareddict 689
SharedFontDirectory 689
compatibility *(continued)*
strategies for *(appendix)* 761–771
types 2 and 4 halftone directories supported for 498
upward, idiom recognition use for 119
composefont operator 545–546
CID-keyed font use 366
CIDFont resource category use by 94
CMap resource category use by 94
composite
fonts 357–364, 357
cshow 551
encoding array use 94
LanguageLevel 2 *(list)* 734
mapping algorithm, **FMapType** resource category *(table)* 92
nested 363–364
objects 34–35
access attributes 37
as graphics state parameters, handling of 178
binary object sequence structure recommendations 167
delimiters for literals 58
graphics state restrictions 215
types of *(table)* 34
VM as pool for values of 57
compression
See also decompression; filters
of data
CCITTFaxEncode *(table)* 86
FlateEncode *(overview table)* 85
image data (**DCTEncode**) *(table)* 86
LZWEncode *(overview table)* 85
RunLengthEncode *(overview table)* 86
data
LZW method 133–137
zlib/deflate compressed format, Flate encoding of 137–142
DCTEncode issues 150
filters, overview 84
of images, color, DCT algorithm impressive with 147
concat operator 547
user space modification by 185
concatmatrix operator 547
conditional execution
if 606
ifelse 607
subfile use for 151
cone depiction with radial shading 270
configurationerror error 547
connected segments 189
constants as literal objects 36
construction
See also creating; defining

construction *(continued)*
of arrays (**counttomark**) 551
of dictionaries (**<<>>**) 54, 525
of glyphs
BuildChar 340–341
BuildGlyph 338–340
of paths 189–193
operators 177, 191
operators *(list)* 518–519
operators, overview 175
of user paths 198–200
constructors
array, square brackets as 32
dictionaries, double angle brackets use for 33
control
characters
communications use of 26
control characters for interactive executive use *(table)* 21
codes, as device dependent elements independent of PostScript language 76
constructs
control operators use as 49
(example) 49
language 97
support (**ControlLanguage**) *(table)* 91
operators
(list) 512
overview 55–56
overprint 247–248
control-C
interactive executive use *(table)* 21
control-R
interactive executive use *(table)* 21
control-U
interactive executive use *(table)* 21
ControlLanguage resource category 97
description *(table)* 91
controlling devices *(chapter)* 391–456
conventions
access, for files 79
end-of-line 74–75
font naming 328
naming
devices 78
file system 78
files 77
structured programs 17–18
further information on, "PostScript Language Document Structuring Conventions" 813
conversion
CIE-based color space to device color space, steps involved 461
color, as rendering step 457
among device color spaces 473–478
from **DeviceCMYK** to **DeviceRGB** 477
between **DeviceGray** and **DeviceCMYK** 475
to **DeviceN** 478
between **DeviceRGB** color space and **DeviceGray** 474–475
from **DeviceRGB** to **DeviceCMYK** 475–477
of objects
to integers (**cvi**) 566
to literal (**cvlit**) 566
to real numbers (**cvr**) 567
to strings (**cvs**) 568
operators
(list) 512
overview 56
of radix numbers to strings (**cvrs**) 567
scan 12–13
as potential glyph rendering component 334
as rendering step 458
details 501–504
rules 502–503
of strings to a name object (**cvn**) 566
type, between integer and real objects 38
Coons
patch mesh
as special case of tensor-product patch meshes 285
as type 6 shading 261, 277–283
tensor-product patch meshes compared with 283
surface equations 278
in tensor-product patch meshes 286
coordinate(s)
space, shading 261
system 182–189
defining with respect to current page 183
forms 208
glyph 331
glyph, transforming into user coordinate system 324
operator overview 175
operators *(list)* 517, 517–518
pattern 252
source image 293–296
specifying for each graphical element 185
transformation 182–189
effects *(figure)* 188
operators 184
operators, modifying user space with 184
rotate 651
scale 654
translate 704
in user space, specifying 184
Coords array
as type 2 shading dictionary entry *(table)* 266

Coords array *(continued)*
as type 3 shading dictionary entry *(table)* 268
copy operator 51, 53, 548–549
copying 35
arrays 39
copy 548
dictionaries 42
copy 548
files 44
objects
on the operand stack (**dup**) 51, 575
simple vs. composite 35
packed arrays (**copy**) 548
portions of the operand stack (**copy**) 51, 548
strings 40
copy 548
copypage operator 549–550
PaintProc procedure prohibited from using 209
usage guidelines 804
use of 392
copyrights 9
cos operator 52, 550
count operator 52, 550
countdictstack operator 54, 551
countexecstack operator 55, 551
counting
entries on execution stack (**countexecstack**) 551
number of dictionaries on the dictionary stack (**countdictstack**) 54, 551
operand stack elements
above the highest mark (**counttomark**) 52, 551
count 52, 550
counttomark operator 44, 52, 551
Courier font family 777
CR (carriage return)
as EOL, scanner handling of 74–75
numeric values *(table)* 27
\r as ASCII encoding *(list)* 29
CR-LF (carriage return-line feed) 74–75
CRD (color rendering dictionaries) 460–473
customizing selection of 472–473
retrieving alternate (**GetSubstituteCRD**) 600
setting (**setcolorrendering**) 663
type 1 461–468
(table) 463
creating
See also construction; defining
arrays
[] 54, 524
array 53, 534
counttomark 551
creating *(continued)*
arrays *(continued)*
that share a subinterval with an array (**getinterval**) 53, 599
CID-keyed fonts 387–390
composefont 545–546
clipping path (**clip**) 193
composite
fonts 357–364
objects, impact on VM 58
dictionaries
<<>> 54, 525
dict 53, 572
files (**file**) 79–80, 587
filters (**filter**) 83–84, 589
IdiomSet instance *(example)* 120
names 40
objects, in VM *(example)* 60
packed arrays
packedarray 53, 631
that share a subinterval with a packed array (**getinterval**) 53, 599
paths (**newpath**) 191, 628
patterns 249
(**makepattern**) 625
random numbers (**rand**) 637
resource categories 99–102
resource category, **Generic** use as template for 102–103
resources (**defineresource**) 89
strings
string 53, 699
that share a subinterval with a string (**getinterval**) 53, 599
subpaths
moveto 627
rmoveto 650
tiling patterns with gradient fill 261
user paths (**upath**) 712
CreationDate entry
as type 1 CIE-based CRD entry *(table)* 468
cshow operator 321, 551
CIDFont restrictions 369
CSL (CID Support Library)
bibliographic reference 365
purposes of 365
CTM (current transformation matrix) 175
See also matrix
BuildGlyph procedure use 339
current path unaffected by changes to 191
graphical element modification *(example)* 186
as graphics state parameter *(table)* 179
operators that modify 185
setting (**setmatrix**) 675
to default for current device (**initmatrix**) 613

CTM (current transformation matrix) *(continued)*
user paths interaction with 205
user space to device space transformation specified by 184
cubic section
appending, to current path (**curveto**) 564
CurDisplayList system parameter
value *(table)* 751
CurFontCache system parameter
glyph consumption reflected in 382
value *(table)* 751
CurFormCache system parameter
value *(table)* 751
curly brackets ({})
executable arrays enclosed by 36
as procedure delimiter 32
special syntactic treatment of 27
CurMID entry
as Type 0 font dictionary entry *(table)* 358
CurOutlineCache system parameter
value *(table)* 751
CurPatternCache system parameter
value *(table)* 751
current
clipping path 178
insideness testing disregard of 197
dictionary 42
file object, retrieving (**currentfile**) 87
font 316
page 176
path 190–192, 190
point 191
currentblackgeneration operator 552
currentcacheparams operator 552
currentcmykcolor operator 215, 553
CIE-based and special color spaces not convertable by 219
DeviceCMYK use 219
currentcolor operator 215, 553
currentcolorrendering operator 553
currentcolorscreen operator 554
currentcolorspace operator 214, 554
currentcolortransfer operator 555
currentdash operator 555
currentdevparams operator 87, 555
currentdict operator 555
currentfile operator 87, 556
(example) 84
filter file use 124
obtaining current file with 74
currentflat operator 556
currentfont operator 557
rootfont compared with 318
currentglobal operator 557
currentgray operator 215, 557
DeviceGray use 220
currentgstate operator 558
global VM precautions 67
gstate object management 182
currenthalftone operator 558
currenthsbcolor operator 215, 558
DeviceRGB use 218
currentlinecap operator 559
currentlinejoin operator 559
currentlinewidth operator 559
current line width retrieving 178
currentmatrix operator 559
currentmiterlimit operator 560
currentobjectformat operator 560
currentoverprint operator 560
currentpacking operator 54, 560
currentpagedevice operator 395, 560
recovery policies and 433–434
currentpoint operator 561
currentrgbcolor operator 215, 561–562
DeviceRGB use 218
currentscreen operator 562
currentshared operator 562
currentsmoothness operator 563
currentstrokeadjust operator 563
currentsystemparams operator 563
currenttransfer operator 563
currenttrapparams operator 564
currentundercolorremoval operator 564
currentuserparams operator 564
CurScreenStorage system parameter
value *(table)* 751
CurSourceList system parameter
value *(table)* 751
CurStoredScreenCache system parameter
value *(table)* 751
CurUPathCache system parameter
value *(table)* 751
curve
Bézier, adding segment to current path (**arcto**) 191
curves
appending
to current path (**curveto**) 564
to current path (**rcurveto**) 638
curveto operator 177, 191, 198, 564
operation code for encoded user paths *(table)* 201

CutMedia page device parameter 413
cvi operator 56, 566
cvlit operator 56, 566
  user path object use 204
cvn operator 56, 566
cvr operator 56, 567
cvrs operator 56, 567
cvs operator 56, 568
  overwriting behavior of 59
cvx operator 56, 568

D

DamagedRowsBeforeError entry
  as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
dangling references
  potential for, as reason for **invalidaccess** error 60
dash
  See also line(s)
  as graphics state parameter *(table)* 180
  pattern
    currentdash 555
    setdash 194, 666
data
  compression
    CCITTFaxEncode *(table)* 86
    FlateEncode *(overview table)* 85
    image data (**DCTEncode**) *(table)* 86
    LZW method 133–137
    LZWEncode *(overview table)* 85
    RunLengthEncode *(overview table)* 86
    zlib/deflate compressed format, Flate encoding of 137–142
  decompression
    CCITTFaxDecode *(table)* 86
    FlateDecode *(overview table)* 86
    image data (**DCTDecode**) *(table)* 86
    LZWDecode *(overview table)* 85
    RunLengthDecode *(overview table)* 86
  pass-through
    NullEncode *(table)* 86
    ReusableStreamDecode 153–156
    subfile filters use for, overview 85
    SubFileDecode *(table)* 86
  sample, decoding 299–301
  sources 123–127
  targets 123–127
  transformation, filters use for 82
  types and objects 34–45
DataDict dictionary
  as type 3 image dictionary entry *(table)* 304
DataSource entry
  as image data dictionary entry *(table)* 305
  as image mask dictionary entry *(table)* 306
  as shading dictionary entry, requirements for 263
  size requirements 110
  as type 0 function dictionary entry *(table)* 109
  as type 1 image dictionary entry *(table)* 298
  as type 4 image dictionary entry *(table)* 308
  as type 4 shading dictionary entry *(table)* 270
  as type 5 shading dictionary entry *(table)* 275
  as type 6 shading dictionary entry *(table)* 279
DCT (discrete cosine transform) algorithm
  compression of image data (**DCTEncode**) *(table)* 86
  decompression of image data (**DCTDecode**) *(table)* 86
  as JPEG encoding technique, used by **DCTEncode**/**DCTDecode** filters 147
DCTDecode filter 86, 146–150
  not recommended as source of type 4 image dictionary 308
DCTEncode filter 86, 147–150
  data requirements 128
  dictionary, entries *(table)* 148
deallocation
  issues, in VM management 60
debugging
  See also errors
  == (double equals) operator use 526
Decode array
  as image data dictionary entry *(table)* 306
  as image mask dictionary entry *(table)* 307
  increasing the accuracy of encoded samples with 111
  mapping with *(figure)* 112
  as type 0 function dictionary entry *(table)* 109
  as type 1 image dictionary entry *(table)* 298
  as type 4 image dictionary entry *(table)* 308
  as type 4 shading dictionary entry *(table)* 270
  as type 5 shading dictionary entry *(table)* 275
  as type 6 shading dictionary entry *(table)* 280
DecodeA procedure
  as **CIEBasedA** dictionary entry *(table)* 229
DecodeABC array
  as **CIEBasedABC** dictionary entry *(table)* 223
DecodeDEF array
  as **CIEBasedDEF** dictionary entry *(table)* 233
  as **CIEBasedDEFG** dictionary entry *(table)* 235
DecodeLMN array
  as **CIEBasedA** dictionary entry *(table)* 230
  as **CIEBasedABC** dictionary entry *(table)* 223
DecodeParms entry
  as **ReusableStreamDecode** dictionary entry *(table)* 155
decoding
  See also encoding; filters
  ASCII, filters, overview 84

decoding *(continued)*
binary data
ASCII85Decode *(overview table)* 85, 130
ASCIIHexDecode *(overview table)* 85
filters 82, 123, 127–128
file use 124
LZW, **LZWDecode** filter 132–137
sample data, sample data 299–301
decompression
See also compression; filters
of data
CCITTFaxDecode *(table)* 86
FlateDecode *(overview table)* 86
image data (**DCTDecode**) *(table)* 86
LZWDecode *(overview table)* 85
RunLengthDecode *(overview table)* 86
filters, overview 84
def operator 54, 568
userdict dictionary entry definition with 42
default
error handling
errordict 56
procedures 115–116
matrix (**defaultmatrix**) 569
user space 183
defaultmatrix operator 569
deferred execution 25, 47–49
procedures characterized by 33
DeferredMediaSelection page device parameter 403
definefont operator 88, 93, 569
associating a resource instance with a category and name 104
composite font handling 364
font dictionary use 323
font modification restrictions 353
FontDirectory dictionary updated by *(table)* 65
FontName parameter use with 328
GlobalFontDirectory dictionary updated by *(table)* 66
defineresource operator 88, 570–571
associating a resource instance with a category and name 104
Category key use by, category implementation dictionary *(table)* 101
creating resources with 89
DefineResource key use by, category implementation dictionary *(table)* 101
font resource handling 324
implicit resource handling 98–99
InstanceType key use by, category implementation dictionary *(table)* 101
resource category implementation dictionary use by 100
types of keys 90
defineresource operator *(continued)*
VM use 89
DefineResource procedure
as category implementation dictionary entry *(table)* 101
defineresource use of 100
defineuserobject operator 571
user objects
defining 67
defining *(example)* 68
defining
See also construction; creating
fonts (**definefont**) 569
glyphs
incrementally 352–356
incrementally, general considerations 353–354
halftones 481–482
patterns 249
resources
Category *(table)* 92
defineresource 89, 570–571
user objects
defineuserobject 67, 571
defineuserobject *(example)* 68
userdict entries, **def** use for 42
definitions of terms
access
attributes 37
string 79
adjusting pages 437
arrays 38
encoding 94
homogeneous number 161
literal array object 162
packed 39
attributes
access 37
executable 36
literal 36
base fonts 321, 357
binary
object sequences 156, 163
tokens 156
binding
early 117
late 117
black generation 476–477
(table) 180
bounding box
glyph 331
(table) 208
brightness 217
byte 157
cache, user path 202

definitions of terms *(continued)*
cell
halftone 482
key pattern 252
pattern 248
characters 313
CMap (character map) 366
collection 365
newline 26
selector 335
white-space 26
charstring 329
clipping path 178, 192
stack 192
stack *(table)* 179
(table) 179
closed path 190
CMap (character map) 366
dictionaries 366
collection
character 365
garbage 63
color
space, render 461
tiling patterns 254
colors
duotone 245
high-fidelity 245
models, HSB 217
models, RGB 217
multitone 245
process 218, 242
rendering. *(table)* 180
space *(table)* 179
(table) 179
value 210
comments 27
composite
fonts 357
objects 34
connected segments 189
consume 25
control, language 97
coordinates 182
form coordinate system 208
glyph coordinate system 331
pattern coordinate system 252
copying 35
CTM (current transformation matrix) 175
(table) 179
current
clipping path 178
dictionary 42
font 316
page 176
definitions of terms *(continued)*
current *(continued)*
path 190
point 191
dash pattern *(table)* 180
decoding, filters 82, 123
default, user space 183
deferred execution 25
descendent font 357
details dictionary 397
device
file system 77
space 182
(table) 181
dictionaries, halftone 483
dictionary 41
current 42
function 107
globaldict 42
stack 42, 45, 56
systemdict 42
userdict 42
domain 107
duotone color 245
early binding 117
encapsulation 69
encoded
system names 160
user path 200
encoding
arrays 94
filters 82, 123
execution
multiple contexts 57
stack 25, 45, 57
exponential interpolation functions 107
file 43, 73
objects 43
positionable 80
standard input 44, 57
standard output 44, 57
system devices 77
fill, gradient 248
filter 82
decoding 82, 123
encoding 82, 123
fixed-pitch fonts 320
fixed-point numbers 160
flatness *(table)* 181
fonts 314
base 321, 357
composite 357
current 316
descendent 357
fixed-pitch 320

definitions of terms *(continued)*
fonts *(continued)*
modal 363
monospaced 320
parent 357
proportional 320
rearranged 364
root 357
set 343
(table) 179
variable-pitch 320
form coordinate system 208
format, structured output 172
forms 206
functions 106
dictionary 107
exponential interpolation 107
sampled 107
spot 484
stitching 107
transfer 478
gamma correction 457
gamut 245, 459–460
garbage collection 63
global virtual memory 59
globaldict dictionary 65
glyph 313
coordinate system 331
metric parameters 331
width 320
gradient fill 248, 259
graphics state 57, 178
stack 181–182
halftone 480
cell 482
dictionaries 483
screens 481
(table) 180
high-fidelity color 245
homogeneous number arrays 161
HSB color model 217
hue 217
idiom recognition 119
image 288
current 176
space 294
implicit, resources 90
incremental definition 353
integer 28
interactive executive 81
interleave blocks 304
interpolation, exponential interpolation functions 107
inverse, of a transformation 189
job 57, 68
server 68

definitions of terms *(continued)*
key, dictionary 41
key pattern cell 252
language, control 97
late binding 117
line
cap *(table)* 180
join *(table)* 180
width *(table)* 179
literal array object 162
local virtual memory 59
looping context 585
mark 44
media
destinations 416
sources 398
memory, virtual 57
misregistration 439
miter joins limit *(table)* 180
modal fonts 363
models, color 217
monospaced fonts 320
multiple execution contexts 57
multitone color 245
name 40
system name index 168
native color space 473
neutral density 444
newline characters 26
null objects 44
numbers 38
fixed-point 160
homogeneous number arrays 161
integer 28
real 28
numeric objects 37
objects 24
attributes, executable 36
attributes, literal 36
binary object sequences 156, 163
composite 34
file 43
literal array 162
null 44
save 44
simple 34
types 34
user 67
open path 190
operand stack 45, 56
operators 42
overprint *(table)* 180
packed arrays 39
page, sets 419
painting 193

definitions of terms *(continued)*
parent font 357
path 177, 189
clipping 178, 192
clipping path stack 192
clipping *(table)* 179
closed 190
current 190
encoded user path 200
open 190
subpaths 190
(table) 179
user path cache 202
patterns 248
cell 248
coordinate system 252
dash *(table)* 180
key pattern cell 252
shading 248
tiling 248
tiling, colored 254
tiling, uncolored 257
pipeline 83
point 183
current 191
polymorphism 52
position *(table)* 179
position number 398
positionable 80
positions 398
procedure 24, 38
sets 95
process colors 218, 242
proportional fonts 320
range 107
real numbers 28
rearranged font 364
recovery policy 433
regular, resources 90
render color spaces 461
request dictionary 394
resources 88
implicit 90
regular 90
reusable, streams 153
RGB color model 217
root font 357
sampled functions 107
saturation 217
save objects 44
screens, halftone 481
separation 242, 424
sequences, binary object 156, 163
server, job 68
set
font 343

definitions of terms *(continued)*
set *(continued)*
procedure 95
shading patterns 248
simple objects 34
smoothness *(table)* 181
space
color *(table)* 179
device 182
image 294
user 182
user, default 183
spot colorants 424
spot functions 484
stack 45
clipping path 192
clipping path *(table)* 179
dictionary 42, 45, 56
execution 25, 45, 57
graphics state stack 181–182
operand 45, 56
standard input file 44, 57
standard output file 44, 57
state
graphics 57, 178
graphics state stack 181–182
stencil mask 301
stitching functions 107, 113
streams, reusable 153
strings 39
access 79
stroke adjustment *(table)* 180
stroking 193
structured output format 172
subpaths 190
supercells 501
system name index 168
systemdict dictionary 65
tiling 250
patterns 248
patterns, colored 254
patterns, uncolored 257
token 26
binary 156
type 157
transfer
functions 478
(table) 180
transformation
inverse of 189
matrix form *(table)* 208
trap 439
trays 398
type
object 34
token 157

definitions of terms *(continued)*
uncolored tiling patterns 257
undercolor removal 476–477
(table) 180
unique ID 335
user
encoded user path 200
objects 67
path cache 202
space 182
space, default 183
userdict dictionary 65
value
color 210
dictionary 41
variable-pitch fonts 320
virtual memory 57
global 59
local 59
VM 57
white-space characters 26
width, glyph 320
DEL (delete)
interactive executive use *(table)* 21
deletefile operator 572
file system access by 77
operand formats 78
special file name not used by 78
deleting
See removing
deprecated practices
copypage 549–550
defining patterns as halftone screens 249
page device switching 431
deriving
fonts 348–356
descendent fonts 357
details dictionaries
for page devices 397
trapping 441–445
device(s)
See also hardware; input; output; product-dependent
color spaces 216–220
conversion between **DeviceRGB** and **DeviceGray** 474–475
conversion of CIE-based color space to, steps involved 461
conversions among 473–478
DeviceCMYK 218–219
DeviceGray 211, 219–220
DeviceGray setting (**setgray**) 671
DeviceN 245–247
DeviceRGB 217–218
device(s) *(continued)*
color spaces *(continued)*
not equivalent to CIE-based color spaces 221
overview 211
remapping **DeviceGray** to CIE-based 237–238
rendering rules for shading patterns 264
Separation 241–245
colorants, separations and 424–426
colors
converting CIE-based color to 459–473
remapping to CIE-based color spaces 237–238
dependence
color handling, **setoverprint** operator 247–248
color rendering as 210
graphics operations *(chapter)* 457–504
graphics state parameters *(table)* 180–181
Type 4 CIDFont considerations 379
file system 77
gamut mapping function, as CIE-based color rendering dictionary component 460
as graphics state parameter *(table)* 181
independence
color spaces, CIE-based 221
color spaces, overview 214
color specification as 210
coordinate system transformation enhancement of 184
graphics *(chapter)* 175–290
graphics state parameters *(table)* 179–180
image properties 289
painting operators 193–197
See also device(s), dependence
initialization
page device parameters *(table)* 426
page setup and 426–432
IODevice resource category 98
(table) 91
naming conventions 78
null
nulldevice 630
nulldevice, usage guidelines 805–806
output, dictionary 455–456
page
See page devices
page, parameters
See page device parameters
parameters
retrieving (**currentdevparams**) 555
setting (**setdevparams**) 667
serial number, retrieving (**serialnumber**) 657
setup
operators *(list)* 521
operators, overview 176
setup operators, **PaintProc** procedure prohibited from using 209

device(s) *(continued)*
space 182
halftones defined in 481
tiling of, in a type 16 halftone dictionary *(figure)* 497
transformation of user space to 184
user space and 182–184
DeviceCMYK color space
conversion
between **DeviceGray** and 475
from **DeviceRGB** to 475–477
to **DeviceRGB** from 477
as device-dependent color space 211
remapping to CIE-based color space 237–238
DeviceGray color space
conversion
between **DeviceCMYK** and 475
between **DeviceRGB** color space and 474–475
as device-dependent color space 211
remapping to CIE-based color space 237–238
setting (**setgray**) 671
DeviceN color space 245–247
conversion to 478
remapping to CIE-based color space 238
rendering rules for shading patterns 264
as special color space 214
DeviceN process color model 422–426
as **OutputDevice** dictionary entry *(table)* 456
precautions on using 424
DeviceRGB color space
conversion
between **DeviceGray** color space and 474–475
from **DeviceCMYK** to 477
to **DeviceCMYK** from 475–477
as device-dependent color space 211
remapping to CIE-based color space 237–238
dict operator 572
as composite object constructor 58
creating dictionaries with 53
dictfull error 573
LanguageLevel 1 41
dictionaries 41
accessing, methods for 41
associating keys with values in, on the dictionary stack (**def**) 54, 568
capacity
extensible in LanguageLevels 2 & 3 53
retrieving the maximum (**maxlength**) 54, 626
category implementation 100–102
(table) 101
CCITTFaxDecode, entries *(table)* 144–145
CCITTFaxEncode, entries *(table)* 144–145
dictionaries *(continued)*
CharStrings
adding or replacing glyphs in 351–352
as Type 1 font dictionary entry *(table)* 326
as Type 42 font dictionary entry *(table)* 346
character names defined in 330
CIDFont 368–382
CIDFont *(table)* 370
CIDSystemInfo 367–368
Type 0 CIDFont *(table)* 373
Type 0 font 366
Type 1 CIDFont *(table)* 377
Type 2 CIDFont *(table)* 378
CIDSystemInfo *(table)* 368
CIE-based color rendering 460, 460–473
customizing 472–473
Type 1 461–468
CIEBasedA *(table)* 229
CIEBasedABC *(table)* 223–224
CIEBasedDEF *(table)* 233, 235
CIEBasedDEFG *(table)* 235
CMap 382–387
(example) 385–387
operators, in the **CIDInit** procedure set 384–385
(table) 383
CMap, as font mapping algorithm *(table)* 360
ColorantDetails
entries *(table)* 443
TrappingDetails dictionary entry *(table)* 442
ColorantZoneDetails
as **Trapping** dictionary entry *(table)* 449
entries *(table)* 454
ColorRendering 460–468
as composite object *(table)* 34
constructors, double angle brackets use for 33
copying 42
copy 53, 548
counting number on the dictionary stack (**countdictstack**) 54, 551
creating
<<>> 54, 525
dict 53, 572
current 42
currentdict 555
DCTEncode, entries *(table)* 148
details
for page devices 397
trapping 441–445
device, output 455–456
entries, for Type 0 fonts 362
$error
default error handling use 116
entries specific to *(table)* 116
error information recorded in 115
standard local dictionary *(table)* 65

dictionaries *(continued)*
errordict 581
PostScript errors maintained in 115
redefining errors in 117
standard local dictionary *(table)* 65
(table) 65
errors
dictfull 573
dictstackoverflow 574
dictstackunderflow 574
FlateEncode/**FlateDecode** *(table)* 138
font 321–328
See also CIDFont resource category
common entries *(table)* 324
entries common to all base fonts *(table)* 325
entries in 324–328
FontInfo *(table)* 327
GlobalFontDirectory 601
graphics state restrictions on procedures in 215
Type 1 *(table)* 326
Type 3 *(table)* 338
Type 42 *(table)* 346
Font, description 93
FontDirectory
maintained by font operators 93
standard local dictionary *(table)* 65
form
describing 207
(table) 208
full, handling of 41
function 107–115
as function representation 106
type 0 107–115, 108–112
type 2 112–113
type 2, entries specific to *(table)* 113
type 3 113–114
type 3, entries specific to *(table)* 114
globaldict 42, 65
standard global dictionary *(table)* 66
GlobalFontDirectory
maintained by font operators 93
standard global dictionary *(table)* 66
halftone 483–484
installation (**sethalftone**) 482
proprietary 499–500
proprietary *(table)* 500
type 1 487–489
type 1 *(table)* 487
type 2 498
type 3 490
type 3 *(table)* 490
type 4 498
type 5 498–499
type 6 491–492
type 6 *(table)* 491
type 9 499
dictionaries *(continued)*
halftone *(continued)*
type 10 492–495
type 10 *(table)* 495
type 16 495
type 16 *(table)* 496
type 100 499
types of *(table)* 485
with multiple color components 497–499
image 297–304
data *(table)* 305
mask *(table)* 306
type 1 297–299
type 1 *(example)* 311
type 1 *(table)* 298
type 3 *(table)* 304
type 4 *(table)* 307
immediate evaluation of names use with 122
InputAttributes, managing the 408–410
installing (**begin**) 536
internal (**internaldict**) 614
job server (**serverdict**) 72, 657
key
searching for (**known**) 54, 619
searching for (**where**) 54, 718
LZW *(table)* 133
mapping a procedure over (**forall**) 53, 597
Metrics, modifying glyph metrics with 350–351
Metrics2, modifying glyph metrics with 350–351
objects 41–42
execution handling 50
operators
(list) 510
overview 52–55
operators that apply only to 54
output device 455
OutputDevice, entries *(table)* 455
page device 394–397
retrieving (**currentpagedevice**) 560
setting (**setpagedevice**) 679
pattern
components of 249
type 1 *(table)* 251
type 2 *(table)* 260
Policies, entries *(table)* 433–435
popping off the dictionary stack (**end**) 54, 577
Private
as **FDArray** array entry *(table)* 374
entries *(table)* 375
Type 0 compared with Type 1 375
procedure set 95–96
procedure substitution
idiom recognition use 119–121
IdiomSet resource category use 97
product-dependent (**statusdict**) 696
pushing onto the dictionary stack (**begin**) 54

dictionaries *(continued)*
removing entries from (**undef**) 54, 708
removing from the dictionary stack
all except for permanent entries (**cleardictstack**) 54, 541
topmost entry (end) 577
replacing values in, on the dictionary stack (**store**) 54, 698
request 394
retrieving all, from the dictionary stack (**dictstack**) 54, 573
retrieving elements of (**get**) 53, 598
retrieving the number of elements in (**length**) 53, 621
ReusableStreamDecode *(table)* 155
shading 261–263
entries common to all *(table)* 262
type 1 (function-based) *(table)* 265
type 2 (axial) *(table)* 266
type 3 (radial) *(table)* 268
type 4 (free-form Gouraud-shaded triangle mesh) *(table)* 270
type 5 (lattice-form Gouraud-shaded triangle mesh) *(table)* 275
type 6 (Coons patch mesh) *(table)* 279
type 7 (tensor-product patch mesh) 283–287
specifying (**userdict**) 713
stack 42, 45, 56
accessing 46
associating keys with values in dictionaries on the (**def**) 54, 568
contents of 42
counting the number of dictionaries on (**countdictstack**) 54, 551
executable name handling use 50–51
locating values on (**where**) 54, 718
popping dictionaries off (**end**) 54, 577
pushing dictionaries onto (**begin**) 54, 536
removing all dictionaries except for permanent entries (**cleardictstack**) 54, 541
replacing values in dictionaries on (**store**) 54, 698
retrieving all dictionaries from (**dictstack**) 54, 573
retrieving values from (**load**) 54, 622
standard 65–67
global *(table)* 66
local *(table)* 65
statusdict, standard local dictionary *(table)* 65
storing elements of (**put**) 53, 635
SubFileDecode filter *(table)* 152
syntax 33
systemdict 42, 65, 702
operator names as keys in 43
standard global dictionary *(table)* 66
user path operators 199
Trapping, entries *(table)* 447–449
dictionaries *(continued)*
trapping
retrieving (**currenttrapparams**) 564
setting (**settrapparams**) 685
TrappingDetails, entries *(table)* 442
user-defined 65–67
userdict 42, 65
standard local dictionary *(table)* 65
writeable, in global VM, guidelines for use 67
dictstack operator 54, 573
dstack array relationship to *(table)* 116
dictstackoverflow error 574
dictstackunderflow error 574
dimensionality
of functions, determination of 107
multidimensional functions, handling 110
multiple, constructing arrays with 39
of sampled functions, performance issues 108
discrete cosine transform (DCT) algorithm
compression of image data (**DCTEncode**) *(table)* 86
decompression of image data (**DCTDecode**) *(table)* 86
display
device color space specification 216
distance
vector, transforming (**dtransform**) 574–575
div operator 52, 574
immediate execution use 47
dividing numbers
div 574
idiv 605
Domain array
as function dictionary entry *(table)* 108
as type 1 shading dictionary entry *(table)* 265
as type 2 shading dictionary entry *(table)* 266
as type 3 shading dictionary entry *(table)* 268
drawing
See path(s)
rectangles (**rectstroke**) 643
dstack array
as **$error** dictionary entry *(table)* 116
dtransform operator 574–575
duotone color 245
dup operator 51, 575
in value sharing example 58
Duplex page device parameter 416
duplicating
paths 192
dynamic
formats, static vs. 14–15
resource loading, as global VM intended use 66

E

E
as real number exponent indicator 28
early binding 117
names (**bind**) 117–123, 118–121, 538–539
EarlyChange entry
as LZW dictionary entry *(table)* 133
echo operator 575
echoing
characters, enabling/disabling (**echo**) 575
edge flags
in type 4 shading dictionaries, vertex specification use 271–273
in type 6 shading dictionaries, vertex specification use 281
eexec operator 576
efficiency
LZW vs. Flate filters 138–139
user paths, lack of side effects contribution to 198
Effort entry
as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
elements
of array
loading onto the operand stack (**aload**) 54, 528
storing objects into (**astore**) 54, 535
of composite objects
retrieving (**get**) 53, 598
retrieving the number of (**length**) 53, 621
storing (**put**) 53, 635
of packed array, loading onto the operand stack (**aload**) 54, 528
embedded
programs, encapsulation of, **save** and **restore** functions 62
Emulator resource category 98
superseded by **PDL** resource category 97
(table) 91
Enabled entry
Trapping dictionary, entries *(table)* 447
encapsulation 69
of **BuildGlyph** procedure 339
of embedded programs, **save** and **restore** functions 62
forms 206
job
overriding 70–72
overriding, LanguageLevel 1 (**exitserver**) 70, 72–73
overriding, LanguageLevel 2 (**startjob**) 70
of **PaintProc** tiling pattern procedure 253
of paths 192
of programs, subfile use for error recovery in 151
of **stop** effects (**stopped**) 697
encapsulation *(continued)*
of user path
gsave and **grestore** use for 205
predictability advantage 197
of VM 68–72
Encode array
as type 0 function dictionary entry *(table)* 109
as type 3 function dictionary entry *(table)* 114
EncodeABC array
as type 1 CIE-based CRD entry *(table)* 464
encoded
number strings
as binary tokens 171–172
homogeneous number array use 162, 171
operators that use 172
system names 160, 168–169
as binary tokens 160
user path 200–202
(example) 202
EncodedByteAlign entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 144
EncodeLMN array
as type 1 CIE-based CRD entry *(table)* 463
encoding
See also filters
arrays 94
ASCII
arrays 32
comments 27
dictionaries 33
filters, overview 84
names 31
numbers 28
procedures 32–33
standard character set use 26–34
strings, ASCII base-85 31
strings, hexadecimal 30–31
strings, hexadecimal, reading from an input file (**readhexstring**) 73
strings, hexadecimal, writing to an output file (**writehexstring**) 73
strings, literal text 29–30
binary data
ASCII85Encode *(overview table)* 85, 131
ASCIIHexEncode *(overview table)* 85
details 156–174
of font sets 343
bit-oriented
CCITT fax standard as (**LowBitFirst**) 146
CCITTFaxDecode *(table)* 86
CCITTFaxEncode *(table)* 86
LZW as (**LowBitFirst**) 136

encoding *(continued)*
of characters 328–330
multiple-byte, CID-keyed font use 364–390
data transmission requirements, ASCII compared with binary 157
filters 82, 123, 128–129
Flate, LZW encoding compared with 138
font, modifying 330
LZW
Flate encoding compared with 138
LZWEncode filter 133–137
PostScript language
ASCII 25
binary 156–174
standard, **StandardEncoding** resource 94
system name *(appendix)* 795–799
Unicode, CMap use for 366
vector (**StandardEncoding**) 693
vectors
changing the 349–350
character code and glyph mapping 328–330
retrieving (**findencoding**) 591
(table) 773–774, 784–794
Encoding array
as base font dictionary entry *(table)* 325
incremental definition rules 354
PrefEnc array relationship with 362
as Type 0 font dictionary entry *(table)* 357
as Type 2 CIDFont dictionary entry *(table)* 378
Encoding resource category
(table) 91
end operator 54, 577
defineresource use of 100
dictionary stack modifiable by 46
endbfchar operator 384, 577
endbfrange operator 384, 577
endcidchar operator 385, 577
endcidrange operator 385, 578
endcmap operator 384, 578
endcodespacerange operator 384, 578
endnotdefchar operator 385, 578
endnotdefrange operator 385, 579
EndOfBlock entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
EndOfLine entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 144
EndPage page device parameter
device initialization *(table)* 427
operations *(table)* 428–430
endrearrangedfont operator 384, 579
endusematrix operator 385, 579
entries
See dictionaries
enumeration
array, dictionary, or string elements (**forall**) 55
font resources, preferred method 93
instances of resource category (**resourceforall**) 89
operators
filenameforall 588
forall 597
graphics (**pathforall**) 632
resourceforall 646
environment
execution
components, graphics state 178–182
components of 57
components, stacks and memory 56–72
components, standard input and output files 73–87
job execution 68–72
eoclip operator 579
even-odd rule use 195
EOD (end-of-data)
> (angle bracket) as indication of **ASCIIHexDecode** 130
CloseTarget and **CloseSource** use of 129
encoding filter transmission of 128
filter termination 126–129
required for filter data 127
~> (tilde angle bracket) as indication of
ASCII85Decode 131
ASCII85Encode 131
EODCount entry
as **SubFileDecode** dictionary entry *(table)* 152
EODString entry
as **SubFileDecode** dictionary entry *(table)* 152
EOF (end-of-file) 80
file termination 126–129
filter handling contrasted to other files 127
eofill operator 580
even-odd rule use 195
EOL (end-of-line)
communication channel level translation 76
conventions 74–75
handling
readline 75
transparent (**read**) 75
transparent (**readstring**) 75
transparent (**write**) 75
transparent (**writestring**) 75
newline characters 26
handling 30
EPS (encapsulated PostScript)
bibliographic reference 9
files, remapping device colors to CIE-based color space 237

EPS (encapsulated PostScript) *(continued)*
PaintProc reference of, **ReusableStreamDecode** filter use with 153
eq operator 55, 580
equality
comparing objects for (**eq**) 580
testing
See also comparing
names 40
user path values 203
user paths, caching basis 203
equals, double (==) operator 87
= operator compared with 87
printing operand stack contents using (**pstack**) 87
equals (=) operator 87
== operator compared with 87
printing operand stack contents using (**stack**) 87
erasepage operator 581
usage guidelines 805
erasing
characters, control characters for interactive executive use *(table)* 21
lines, control characters for interactive executive use *(table)* 21
page (**erasepage**) 581
$error dictionary 527
default error handling use 116
entries specific to *(table)* 116
error information recorded in 115
as standard local dictionary *(table)* 65
(table) 65
errordict dictionary 581
default error handlers, **stop** use 56
as errors dictionary *(overview)* 56
PostScript errors maintained in 115
redefining errors in, as user error handling modification mechanism 117
standard handlers maintained in 115
as standard local dictionary *(table)* 65
(table) 65
errorinfo array
as **$error** dictionary entry *(table)* 116
errorname entry
as **$error** dictionary entry *(table)* 116
errors 114–117
array out-of-bounds (**rangecheck**) 38
category implementation procedures, interpreter assumptions 101
configurationerror 547
dictfull 573
LanguageLevel 1 41
dictstackoverflow 574
dictstackunderflow 574
errors *(continued)*
in encapsulated programs, subfile use for recovery from 151
execstackoverflow 584
file access 80
handleerror 605
structured output error handling 173–174
handling 115–117
default procedures 115–116
encapsulating the effects of **stop** (**stopped**) 697
$error 527
terminating execution (**stop**) 697
unsatisfied parameter requests 432–439
user modification mechanisms 117
initiation 115–116
interrupt 614
invalidaccess 616
file access cause 80
local VM reference by global VM object cause 60
invalidexit 617
invalidfileaccess 617
file access cause 80
invalidfont 617
invalidrestore 617
ioerror 618
file access cause 80
LanguageLevel 1 (**dictfull**) 41
limitcheck 621
implementation limits 737–743
radix number use 28
real number use 28
(list) 523
malformed hexadecimal strings (**syntaxerror**) 30
mathematical operation use (**undefinedresult**) 52
memory exhaustion cause (**VMerror**) 62, 716
name search failure (**undefined**) 51
nocurrentpoint 629
number tolerated, by a **CCITTFaxEncode**/**CCITTFaxDecode** filter *(table)* 145
radix number use (**limitcheck**) 28
rangecheck 638
array out-of-bounds 38
real number use (**limitcheck**) 28
resource file loading 105
stackoverflow 693
stackunderflow 693
standard error file, **file** handling 80
syntaxerror 701
(malformed hexadecimal strings) 30
timeout 702
translating resource names into file names 104
typecheck 706
undefined 709
name search failure 51

errors *(continued)*
undefinedfilename 709
invalid EOL cause 82
undefinedresource 709
undefinedresult 709
mathematical operation use 52
unmatchedmark 711
unregistered 712
VMerror, memory exhaustion cause 62, 716
escape character (\)
special characters that use *(list)* 29
EscChar entry
as Type 0 font dictionary entry *(table)* 358
estack array
as **$error** dictionary entry *(table)* 116
evaluation
of names
immediate evaluation 121–123
immediate, syntax of 31
even-odd rule 196
See also insideness testing
clipping (**eoclip**) 579
ineofill 197
insideness testing
(**ineofill**) 610
(**inueofill**) 614
inueofill 197
nonzero winding number rule compared with 194
painting
areas (**eofill**) 580
user paths (**ueofill**) 708
exch operator 51, 581
exchanging
objects, on the operand stack (**exch**) 51, 581
exclusive or operation (**xor**) 722
exec operator 55, 56, 74, 582
implicit, binary object sequence use 163
no equivalent operator for functions 106
run as combination of **file** and 87
scanner invoked by 157
execform operator 582
form
dictionary use 208–209
invocation by 207
Form resource category use 95
execstack operator 55, 583
estack array relationship to *(table)* 116
execstackoverflow error 584
execuserobject operator 67, 68, 584
executable
arrays
building user paths as 204
execution handling 50
interpreter handling of *(example)* 48
semantics of 36
with immediate execution, binary object sequence as 163
attribute
setting (**cvx**) 568
testing for (**xcheck**) 721
encoded system name specification of 168
files, execution handling 50
names
ASCII encoding 31
handling *(example)* 49
semantics of 36
null, execution handling 51
object attribute 36–37, 36
operator
execution handling 51
semantics of 36
strings, execution handling 50
execute access attribute
setting (**executeonly**) 584
execute-only
access, as object attribute 37
executeonly operator 56, 584
execution
conditional
if 606
ifelse 607
deferred 47–49
procedures characterized by 33
of encrypted file (**eexec**) 576
environment
components, graphics state 178–182
components, stacks and memory 56–72
components, standard input and output files 73–87
encapsulating the effects of **stop** (**stopped**) 697
for a print job 68–72
of files (**run**) 87
immediate 46–47
multiple contexts 57
of objects
effects of 24
exec 582
with the executable attribute 50–51
of programs
exec 74
run 653
requirement, binary object sequences suitable for 156, 163

execution *(continued)*
semantics 46–51
stack 25, 45, 57
accessing 46
counting entries on (**countexecstack**) 551
executable file handling use 50
executable string handling use 50
executing top element of (**exec**) 582
reading (**execstack**) 583
starting (**start**) 693
state, stacks used to manage 45
terminating
quit 637
stop 697
of user objects
execuserobject 67, 584
execuserobject *(example)* 68
executive
interactive
control characters *(table)* 21
invoking (**executive**) 81, 585
prompting user (**prompt**) 635
executive operator 585
interactive executive invoked by 81
usage guidelines 805
exit operator 55, 585
invalid execution error (**invalidexit**) 617
exitserver operator 72–73, 586
guidelines for use 70
as LanguageLevel 1 equivalent to **startjob** 70
multiple execution impact 72
as **serverdict** element 72
usage guidelines 805
exp operator 52, 586
Expert character set
(table) 787–789
Expert encoding vector
(table) 774, 790
ExpertSubset encoding vector
(table) 774, 791
explicit
masking, of images 303–307
exponential interpolation functions 107
type 2 function dictionary 112–113
entries specific to *(table)* 113
exponentiation
exp 586
exponents
in real numbers 28
type 2 function dictionary specification (**N**) *(table)* 113
Extend array
as type 2 shading dictionary entry *(table)* 267
as type 3 shading dictionary entry *(table)* 268

F

facsimile-encoding
CCITTFaxDecode filter 143–146
CCITTFaxEncode filter 143–146
data compression with (**CCITTFaxEncode**) *(table)* 86
data decompression from (**CCITTFaxDecode**) *(table)* 86
FactoryDefaults system parameter
value *(table)* 751
false 55, 587
bind not usable with 119
as boolean value 38
FamilyName entry
as entry in **FontInfo** dictionary *(table)* 327
FDArray array
entries in dictionary *(table)* 374
as Type 0 CIDFont dictionary entry *(table)* 373
FDBytes entry
as Type 0 CIDFont dictionary entry *(table)* 373
FDepVector array
as Type 0 font dictionary entry *(table)* 357
FF (form feed)
\f as ASCII encoding *(list)* 29
numeric values *(table)* 27
FID entry
as **CIDFont** dictionary entry *(table)* 371
as font dictionary entry *(table)* 325
file operator 587
file
object creation by 79
system access by 77
%lineedit file handling 81
obtaining file objects for standard input and standard output files 74
operand formats 78
run as combination of **exec** and 87
special file handling 80–82
%statementedit file handling 81
file(s)
access conventions *(table)* 79
closing 79–80
closefile 80, 544
restore 80
as composite object *(table)* 34
copying 44
creating (**file**) 79–80, 587
as data sources and targets 124
deleting (**deletefile**) 572
errors (**undefinedfilename**) 709
executing 50
run 87
filtered
semantics of 123–156

file(s) *(continued)*
filtered *(continued)*
valid as operands to file operators 83
flushing (**flushfile**) 594
input 73
arbitrary data source as (**SubFileDecode**) *(table)* 86
executing programs from (**exec**) 74
reading characters from (**read**) 73
reading lines from (**readline**) 73
reading strings from (**readstring**) 73
scanning for PostScript tokens (**token**) 73
names 77–80
retrieving (**filenameforall**) 588
syntax 77
objects 43–44
files represented by 73
retrieving the current (**currentfile**) 87
operators
basic, overview 73–76
(list) 513–514
miscellaneous, overview 87
output 73
arbitrary data target as (**NullEncode**) *(table)* 86
writing characters to (**write**) 73
writing strings to (**writestring**) 73, 720
position
retrieving (**fileposition**) 589
setting (**setfileposition**) 80, 668
random-access, creating (**ReusableStreamDecode**) *(table)* 86
reading, in arbitrary order (**setfileposition**) 80
renaming (**renamefile**) 644
resetting (**resetfile**) 645
resources as 103–105
retrieving (**currentfile**) 556
as sample data source 291
special 80–82
standard
handling 74
input 44, 57
input, as execution environment component 73–87
output 44, 57
output, as execution environment component 73–87
status, retrieving (**status**) 696
status information
retrieving (**bytesavailable**) 87
retrieving (**status**) 87
subfile filters, overview 85
system
access permitted (**deletefile**) 77
access permitted (**file**) 77
access permitted (**filenameforall**) 77
access permitted (**fileposition**) 77
access permitted (**renamefile**) 77
access permitted (**setfileposition**) 77
file(s) *(continued)*
system *(continued)*
access permitted (**status**) 77
naming conventions 78
operators, LanguageLevel 2 *(list)* 734
terminating, with EOF 126–129
writing, in arbitrary order (**setfileposition**) 80
filenameforall operator 588
file system access by 77
operand formats 78
special file name not used by 78
wildcard use with 78
fileposition operator 589
file system access by 77
ReusableStreamDecode filter use 154
fill gradient 259
shading dictionary specification 261
shfill 260, 689
fill operator 177, 589
current color use 178
path-enclosed region painting 194
filling 194–196
See also painting
insideness testing 196–197
rectangles (**rectfill**) 642
rules
even-odd 196
nonzero winding number 195
user paths
ufill 197
ueofill 708
Filter entry
as **ReusableStreamDecode** dictionary entry *(table)* 155
filter operator 589
creating filters with 83–84
procedure as data target use 125
ReusableStreamDecode filter use with 153
Filter resource category
implicit resources as instances of 90
filters
See also decoding; encoding
ASCII
decoding, overview 84
encoding, overview 84
ASCII85Decode 130
ASCII85Encode 131–132
ASCIIHexDecode 130
ASCIIHexEncode 130
as binary data handling mechanism 76
CCITTFaxDecode 143–146
dictionary entries *(table)* 144–145
CCITTFaxEncode 143–146
dictionary entries *(table)* 144–145
compression, overview 84

filters *(continued)*
creating (**filter**) 83–84, 589
DCTDecode 146–150
DCTEncode 147–150
decoding 82, 123, 127–128
decompression, overview 84
details of individual 129–156
encoding 82, 123, 128–129
EOF handling contrasted to other files 127
file objects created by, valid as operands to file operators 83
as files in a pipeline 83
filter operator, **Filter** resource category use by 98
Filter resource category 98
(table) 91
FlateDecode 137–142
FlateEncode 137–142
as image data source *(example)* 311
image data sources compared with 291
line editor, **file** handling 80
LZWDecode 132–137
LZWEncode 133–137
NullEncode 156
overview 82–87
ReusableStreamDecode 153–156
RunLengthDecode 142
RunLengthEncode 142–143
running backwards, procedures as data source and target use for 126
semantics of 123–156
standard 84–86
overview *(table)* 85
standard names *(list)* 590
statement editor, **file** handling 80
SubFileDecode 151–152
supported, determining (**resourceforall**) 85
terminating, with EOD 126–129
findcolorrendering operator 591
rendering intent specification with 469
findencoding operator 591
findfont operator 88, 93, 592
accessing fonts with 315
bind not usable with 119
font execution initiated by 323
FontDirectory dictionary accessed by *(table)* 65
GlobalFontDirectory dictionary accessed by *(table)* 66
findresource operator 88, 592
FindResource key use by, category implementation dictionary *(table)* 101
IdiomSet use 121
implementation dictionary 100
implicit resource handling 98–99
as page description resource accessor 103
as resource facility foundation 89
resourceforall and **resourcestatus** compared with 90
FindResource procedure
as category implementation dictionary entry *(table)* 101
fixed-pitch fonts 320
fixed-point numbers 160
See also mathematics; numbers
as binary tokens 160
Flate encoding
See also filters
dictionary *(table)* 138
as IETF standard 137
LZW encoding compared with 138
predictor functions use with 139–141
FlateDecode filter 86, 137–142
FlateEncode filter 85, 137–142
flatness parameter
graphics state *(table)* 181
retrieving (**currentflat**) 556
setting (**setflat**) 669
flattening
paths (**flattenpath**) 593
flattenpath operator 593
floor operator 52, 594
flush operator 594
buffered character handling 74
flushfile operator 594
buffered character handling 74
with a decoding filter 127
with an encoding filter 128
ReusableStreamDecode filter use 154
flushing
buffers
flush 594
resetfile 645
files (**flushfile**) 594
FMapType entry
mapping algorithms 358–360
(table) 360
as Type 0 font dictionary entry *(table)* 357
FMapType resource category 98
(table) 92
Font resource category 93
(table) 91
VM allocation mode policies 105
font(s)
accessing
CFF and Chameleon fonts in a PostScript program 345–346
findfont 315
base 357
dictionary entries common to all *(table)* 325
binary data section (**StartData**) 694
BuildChar procedure 340–341

font(s) *(continued)*
BuildGlyph procedure 338–340
cache 333–335
currentcacheparams 552
incremental definition of font impact 354
loading glyphs into (**addglyph**) 528
setting parameters (**setcacheparams**) 661
setting (**setcachedevice**) 659
setting (**setcachedevice2**) 659
setting size (**setcachelimit**) 660
status (**cachestatus**) 540
Type 4 CIDFont advantages 380
user path cache analogous to 202
CFF 343–346
FontSet resource category 94
(chapter) 313–390
CID-keyed 364–390
CIDFont resource category 94
CMap 94
Type 0 371–376, 375–376
Type 0 (**CIDInit**) *(table)* 96
Type 1 376–377
Type 2 377–379
Type 4 379–382
Type 4 (**BitmapFontInit**) *(table)* 96
composite 357–364
CID-keyed 387–390
cshow 551
encoding array use 94
nested 363–364
current 316
defining (**definefont**) 569
definitions
FontDirectory dictionary *(table)* 65
in global VM, **GlobalFontDirectory** *(table)* 66
derivation 348–356
descendant 357
dictionaries 321–328
CIDFont *(table)* 370
CIDSystemInfo 367–368
common entries *(table)* 324
entries common to all base fonts *(table)* 325
entries in 324–328
FontInfo *(table)* 327
GlobalFontDirectory 601
graphics state restrictions on procedures in 215
Type 0 CIDFont *(table)* 373
Type 1 CIDFont *(table)* 377
Type 1 *(table)* 326
Type 2 CIDFont *(table)* 378
Type 3 *(table)* 338
Type 42 *(table)* 346
used to manage 41
encoding
modifying the encoding scheme 330
font(s) *(continued)*
encoding *(continued)*
modifying the encoding vector 349
execution of 323
families
Courier 777
Helvetica 776
Times 775
fixed-pitch 320
FontDirectory 595
formats 8
garbage collection benefits for management of 64
as graphics state parameter *(table)* 179
incremental definition of 352–356
introduction into VM 88
invalid specification error (**invalidfont**) 617
Latin-text
ISOLatin1Encoding 618
ISOLatin1Encoding resource 95
mapping 358–362
modification 348–356
modifying, unique ID precautions 336
monospaced 320
naming conventions 328
operators
iteration (**cshow**) 551
iteration (**kshow**) 619
(list) 521–522
overview 175
organization and use 313–321
parent 357
proportional 320
rearranged 364
removing (**undefinefont**) 323
resource category 94
retrieving
currentfont 557
findfont 592
rootfont 651
root 357
scaling
makefont 315, 624
scalefont 654
selectfont 656
selecting 316–318
selectfont 656
sets 343
FontSet 344–345
setting (**setfont**) 670
special graphics effects 318–320
specifying (**usefont**) 713
subsetting 352–356
TrueType 346–348
Type 0
CID-keyed 371–376

font(s) *(continued)*
Type 0 *(continued)*
dictionary entries for 362
Type 1, incremental definition of 355
Type 2 343–346
Type 3 337–342
(example) 341–342
incremental definition of 355
Type 14 343–346
Type 42 346–348
incremental definition of 355–356
types *(table)* 322
undefining (**undefinefont**) 710
unique ID 335–337
variable-pitch 320
FontBBox array
See also bounding box
as base font dictionary entry *(table)* 325
as **CIDFont** dictionary entry *(table)* 370
fonts 333
(table) 370
FontDirectory dictionary 595
maintained by font operators 93
as standard local dictionary *(table)* 65
fontID objects
execution handling 50
as an object type 45
as simple object *(table)* 34
FontInfo dictionary
as font dictionary entry *(table)* 324
(table) 327
FontMatrix array
as **CIDFont** dictionary entry *(table)* 370
as **FDArray** array dictionary entry *(table)* 374
as font dictionary entry *(table)* 324
nested composite font treatment 364
as Type 0 font dictionary entry 362
Type 4 CIDFont use 381
FontName entry
as **FDArray** array dictionary entry *(table)* 374
as font dictionary entry *(table)* 324
FontResourceDir system parameter
value *(table)* 752
FontSet resource category 344–345
(table) 91
Type 2 and Type 14 fonts use of 343
FontSetInit procedure set
LanguageLevel 3 operators defined in *(table)* 726
as standard procedure set in LanguageLevel 3 *(table)* 96
FontType entry 98
as **CIDFont** dictionary entry *(table)* 370
as font dictionary entry *(table)* 324
(table) 92
values *(table)* 370
for operator 55, 596
forall operator 55, 597
mapping a procedure over composite objects 53
packed array access permitted by 54
form feed (FF) character
\f as ASCII encoding *(list)* 29
numeric values *(table)* 27
Form resource category 95
(table) 91
formats
dynamic, static vs. 14–15
object
retrieving (**currentobjectformat**) 560
setting (**setobjectformat**) 677
print, translating from other 19–20
structured output 172–174, 172
forms 206–209
appearance, describing 207
caching 207
coordinate system 208
dictionary, describing 207
operators *(list)* 520
painting
execform 582
procedure for *(table)* 208
transformation matrix *(table)* 208
using 207–209
FormType entry 98
as form dictionary entry *(table)* 208
purpose of 207
(table) 92
FORTH
as PostScript influence 23
framedevice operator
usage guidelines 805
free-form
Gouraud-shaded triangle meshes, as type 4 shading 261, 270–274
Frequency entry
as type 1 halftone dictionary entry *(table)* 487
FullName entry
as entry in **FontInfo** dictionary *(table)* 327
Function entry
as type 1 shading dictionary entry *(table)* 265
as type 2 shading dictionary entry *(table)* 266
as type 3 shading dictionary entry *(table)* 268
as type 4 shading dictionary entry *(table)* 271
as type 5 shading dictionary entry *(table)* 275
as type 6 shading dictionary entry *(table)* 280
function(s) 106–114
based shading patterns, as type 1 shading 261, 265–266

function(s) *(continued)*
black-generation
retrieving (**currentblackgeneration**) 552
setting (**setblackgeneration**) 476, 658
setting usage guidelines (**setblackgeneration**) 806
CIE-based color mapping, CIE-based color conversion to device color use 460
CIE-based gamut mapping
as CIE-based color rendering dictionary component 460
CIE-based color conversion to device color use 459
gamma correction contrasted with 478
color conversion, use, as rendering step 457
color mapping, as CIE-based color rendering dictionary component 460
dictionaries 107–115
entries common to *(table)* 108
type 0 108–112
type 2 112–113
type 2, entries specific to *(table)* 113
type 3 113–114
type 3, entries specific to *(table)* 114
dictionary, color transition definition by, shading dictionary relationship to 263
exponential interpolation 107
type 2 function dictionary 112–113
type 2 function dictionary, entries specific to *(table)* 113
FunctionType resource category *(table)* 92
halftone, use, as rendering step 458
predictor
LZW or Flate dictionary entries related to *(table)* 141
PNG-based 139
TIFF-based 139
TIFF-based compared with PNG-based 140
with LZW and Flate filters 139–141
sampled
ReusableStreamDecode filter use 153
(type 0 function dictionary) 108–112
spot 484–489
stitching 107
type 3 function dictionary 113–114
type 3 function dictionary, entries specific to *(table)* 114
transfer 478–480, 478
retrieving (**currenttransfer**) 563
setting (**setcolortransfer**) 479, 666
setting (**settransfer**) 479, 685
use, as rendering step 457
trigonometric
atan 535–536
cos 550
sin 692
function(s) *(continued)*
type 0 107
type 2 107
type 3 107
undercolor removal
retrieving (**currentundercolorremoval**) 564
setting (**setundercolorremoval**) 476, 687
Functions array
as type 3 function dictionary entry *(table)* 114
FunctionType entry
as function dictionary entry *(table)* 108
function types specified in 107
FunctionType resource category 98

G

gamma correction 457
CIE-based gamut mapping function contrasted with 478
gamut 245, 459–460
See also CIE-based color spaces
mapping functions
color rendering dictionary component 460
gamma correction contrasted with 478
WhitePoint and **BlackPoint** use 224
garbage collection
See also memory; VM (virtual memory)
CloseTarget and **CloseSource** use by 129
closing files with 80
filter pipeline impact 129
managing (**vmreclaim**) 716
save and **restore** compared with 63–64
setting VM allocation threshold for (**setvmthreshold**) 688
gcheck operator 598
See also VM (virtual memory)
VM storage management role 61
GDBytes entry
as Type 0 CIDFont dictionary entry *(table)* 373
as Type 2 CIDFont dictionary entry *(table)* 378
ge operator 55, 598
generating
random numbers
rand 637
retrieving current state (**rrand**) 652
srand 692
generation
black
retrieving function (**currentblackgeneration**) 552
setting function (**setblackgeneration**) 658
setting function usage guidelines (**setblackgeneration**) 806

generation *(continued)*
machine, binary encodings used for 25
of unique identifier 335–337
Generic resource category
category 102–103
(table) 92
GenericResourceDir system parameter
value *(table)* 752
GenericResourcePathSep system parameter
value *(table)* 752
get operator 53, 598
GetHalftoneName operator 599
customizing CRD selection use 472
getinterval operator 599
array creation with 39
creating objects that share a subinterval with a composite object 53
GetPageDeviceName operator 600
customizing CRD selection use 472
GetSubstituteCRD operator 600
customizing CRD selection use 472
global
standard dictionaries *(table)* 66
VM 59–61
category implementation dictionary located in 101
guidelines for use 66
setting mode for (**setglobal**) 670
testing objects for eligibility (**gcheck**) 598
globaldict dictionary 42, 65, 601
as dictionary stack component 46
as standard global dictionary *(table)* 66
GlobalFontDirectory dictionary 601
maintained by font operators 93
as standard global dictionary *(table)* 66
glyph(s) 313
See also character(s); font(s)
adding 351–352
appending, to current path (**charpath**) 319
bitmaps
stencil masking use for painting 303
Type 4 CIDFont use 379
Type 4 CIDFont use (**BitmapFontInit**) *(table)* 96
bounding box 331
setting (**setcachedevice**) 659
setting (**setcachedevice2**) 659
color, setting 318
coordinate system 331
transforming into user coordinate system 324
creating
BuildChar 340–341
BuildGlyph 338–340
descriptions 314
incremental definition of 352–356
glyph(s) *(continued)*
incremental definition of *(continued)*
general considerations 353–354
left sidebearing 332
loading into font cache (**addglyph**) 528
metrics 331–333
changing the 350–351
overriding 321
operators
(list) 521–522
overview 175
origin 331
outlines, treating as a path 319
painting
glyphshow 602
show 177, 316, 690
stencil masking use for bitmapped characters 303
widthshow 718
xshow 722
xyshow 722
yshow 723
positioning 320–321
ashow 320, 534–535
awidthshow 320, 536
cshow 321, 551
kshow 321, 619
show 320, 690
widthshow 320, 718
xshow 320, 722
xyshow 321, 722
yshow 320, 723
reference point 331
removing
removeall 644
removeglyphs 644
replacing 351–352
scaling (**scalefont**) 315
spacing, modification difficulties 350
subsetting 352–356
general considerations 353–354
width 320, 331
adjusting (**widthshow**) 718
retrieving (**stringwidth**) 699
setting (**setcharwidth**) 661
GlyphData entry
as Type 0 CIDFont dictionary entry *(table)* 374
GlyphDirectory entry
as Type 0 CIDFont dictionary entry 375–376
(table) 374
as Type 2 CIDFont dictionary entry *(table)* 379
as Type 42 font dictionary entry 355–356
(table) 347
glyphshow operator 602
character name use with 330

GlyphDirectory entry *(continued)*
CharStrings interaction 352
CIDFont restrictions 369
Gouraud shading
See also patterns
free-form Gouraud-shaded triangle meshes 261, 270–274
lattice-form Gouraud-shaded triangle meshes 261, 274–276
gradient(s)
See also shading
fill 248, 259
shading dictionary specification 261
in imaging model 176
graphical elements
defining independently of other elements 185
graphics
operators
facilities *(chapter)* 175–290
overview of main groups 175–176
special effects for fonts 318–320
state
See graphics state
graphics state 57, 178–182
accessing (**gstate**) 58
current halftone dictionary 483
font parameter, accessing 318
initializing (**initgraphics**) 612
managing
currentgstate 182
gstate 182
setgstate 182
objects, global VM precautions 67
operators
device-dependent *(list)* 516–517
device-independent *(list)* 515–516
overview 175
restrictions 215
parameters
device-dependent *(table)* 180–181
device-independent *(table)* 179–180
used by **findcolorrendering** 471
restoring
grestore 603
grestoreall 61
retrieving
currentgstate 558
gstate 604
saving (**gsave**) 603
setting (**setgstate**) 671
stack 181–182
as one of five execution state stacks 45
user path no effect on 198
gray
See also color
graphics state *(continued)*
color space
conversion between **DeviceGray** and **DeviceCMYK** 475
conversion between **DeviceRGB** and **DeviceGray** 474–475
DeviceGray 211, 219–220
remapping **DeviceGray** to CIE-based 237–238
setting (**setgray**) 671
component of calibrated gray space, **CIEBasedA** representation 228
levels, halftones, supercell enhancement of 500–501
mapping sample data to color component values for (**setgray**) 291
operators
currentgray 557
setgray 671
grayscale data
compressing (**DCTEncode**) *(table)* 86
decompressing (**DCTDecode**) *(table)* 86
greater-than-or-equal relation
comparing objects for (**ge**) 598
greater-than relation
comparing objects for (**gt**) 604
grestore operator 603
clipping path stack management 192
current path restored by 190
encapsulating paths with 192
execform invocation of 209
graphics state stack management 181
user path encapsulation by 205
grestoreall operator 603
graphics state reset by 61
usage guidelines 805
gsave operator 603
current path saved by 190
encapsulating paths with 192
execform invocation of 209
graphics state stack management 181
user path encapsulation by 205
gstate objects
execution handling 50
interactive application management with 181
as an object type 45
(table) 34
gstate operator 604
as composite object constructor 58
gstate object management 182
gt operator 49, 55, 604
guidelines
See also algorithms; rules
exitserver operator 70
operator usage *(appendix)* 801–810

guidelines *(continued)*
PaintProc procedure use
form 209
tiling pattern 253
startjob operator 70
VM, global 66

H

Halftone resource category 96
(table) 91
halftone(s) 480–501
See also color; gray; monochrome
defining 481–482
screens with spot functions 484–489
dictionaries 483–484
installation (**sethalftone**) 482
proprietary 499–500
proprietary *(table)* 500
type 1 487–489
type 1 *(table)* 487
type 2 498
type 3 490
type 3 *(table)* 490
type 4 498
type 5 498–499
type 6 491–492
type 6 *(table)* 491
type 9 499
type 10 492–495
type 10 *(table)* 495
type 16 495
type 16 *(table)* 496
type 100 499
types of *(table)* 485
with multiple color components 497–499
functions, use, as rendering step 458
as graphics state parameter *(table)* 180
name, retrieving (**GetHalftoneName**) 599
resource categories
Halftone 96
Halftone *(table)* 91
HalftoneType 98
HalftoneType *(table)* 92
screens 481, 482–483
defining patterns as, deprecated practice 249
definition (**setcolorscreen**) 481
definition (**setscreen**) 481
retrieving (**currenthalftone**) 558
retrieving (**currentscreen**) 562
setting (**setcolorscreen**) 664
setting (**sethalftone**) 671
setting (**setscreen**) 681
threshold array definition of 489–497
halftone(s) *(continued)*
spot functions 484–489
supercells 500–501
transfer functions (**sethalftone**) 479
user parameters
AccurateScreens 482, 487, 757
HalftoneMode 482
MaxSuperScreen 482
HalftoneMode user parameter
halftone-setting operator impact 482
value *(table)* 749
HalftoneName entry
as proprietary halftone dictionary entry *(table)* 500
as type 1 halftone dictionary entry *(table)* 487
as type 3 halftone dictionary entry *(table)* 490
as type 6 halftone dictionary entry *(table)* 491
as type 10 halftone dictionary entry *(table)* 495
as type 16 halftone dictionary entry *(table)* 496
HalftoneType entry
as proprietary halftone dictionary entry *(table)* 500
as type 1 halftone dictionary entry *(table)* 487
as type 3 halftone dictionary entry *(table)* 490
as type 6 halftone dictionary entry *(table)* 491
as type 10 halftone dictionary entry *(table)* 495
as type 16 halftone dictionary entry *(table)* 496
HalftoneType resource category 98
(table) 92
handleerror error 605
handleerror operator
binary key use *(table)* 117
default error handling use 116
errorinfo array use *(table)* 116
newerror entry use *(table)* 116
redefining, as user error handling modification mechanism 117
structured output error handling 173–174
handling errors 115–117
default procedures 115–116
$error 527
handleerror 605
unsatisfied parameter requests 432–439
user modification mechanisms 117
hardware
See also devices; input; output; product-dependent
HWOptions resource category 98
options (**HWOptions**) *(table)* 91
output, raster 11–12
headers
binary object sequences 163
resource file 105
Height entry
as image data dictionary entry *(table)* 305
as image mask dictionary entry *(table)* 306

Height entry *(continued)*
as type 1 image dictionary entry *(table)* 298
as type 3 halftone dictionary entry *(table)* 490
as type 4 image dictionary entry *(table)* 307
as type 6 halftone dictionary entry *(table)* 491
as type 16 halftone dictionary entry *(table)* 496
Height2 entry
as type 16 halftone dictionary entry *(table)* 496
Helvetica font family 776
hexadecimal
ASCII
decoding binary data from (**ASCIIHexDecode**) *(overview table)* 85
encoding binary data as (**ASCIIHexEncode**) *(overview table)* 85
radix number representation of 28
strings
ASCII encoding 30–31
reading (**readhexstring**) 639
writing (**writehexstring**) 720
high-fidelity color 245
render color space use 461
highlights
diffuse achromatic, **WhitePoint** use 224
specular, compared with diffuse achromatic highlights 224
hit detection
insideness testing use 196
homogeneous
number arrays
as binary tokens 161–162
encoded number string use 171
horizontal
sampling, **DCTEncode** dictionary entry *(table)* 148
writing, font metrics for 332
HSamples entry
as **DCTEncode** dictionary entry *(table)* 148
HSB (hue-saturation-brightness)
See also color, models
as additive color model 217
as alternative RGB convention 211
retrieving (**currentsbcolor**) 558
setting (**sethsbcolor**) 672
hue 217
Huffman coding
Flate encoding use 137
FlateDecode filter use, overview *(overview table)* 86
FlateEncode filter use, overview *(overview table)* 85
HuffTables array
as **DCTEncode** dictionary entry *(table)* 149
HWOptions resource category 98
(table) 91
HWResolution page device parameter
OutputDevice dictionary entry *(table)* 456
page image placement parameter *(table)* 414

I

ID (identifier)
extended unique ID *(table)* 251
unique
extended
*See **XUID** array*
generation of 335–337
numbers 336
numbers, extended 337
identifiers
name objects use for 40
identifying
glyphs in font cache 335
identity
matrix (**identmatrix**) 605
transform
matrix 626
Type 2 CIDFont defined in terms of 379
Type 42 fonts defined in terms of 347
identmatrix operator 605
idiom recognition 119–121
IdiomRecognition user parameter
bind operator use 119–121
value *(table)* 749
IdiomSet resource category 97
creating an instance of *(example)* 120
idiom recognition use of 119–121
matching procedures different from other resource categories 121
IdiomSet resource category description *(table)* 91
idiv operator 52, 605
idtransform operator 606
IEEE 754 standard
real number representation 170
if operator 55, 606
boolean object use with 38
ifelse operator 55, 607
boolean object use with 38
(example) 49
image operator 177, 607
bilevel picture encoding convention, **CCITTFaxEncode**/**CCITTFaxDecode** filter use 145
CIE-based color space use 221
color space conflicts with **Pattern** color space 254
color space use 296
DCTDecode filter output 146

image operator *(continued)*
(example) 84
image dictionary use 297–304, 299–301
images as color value source 211
Indexed color space use with 241
prohibited from font cache use 334
sample
data sources 291
representation 290
Separation color space use 243, 244
uncolored tiling pattern prohibited from using 257
image(s)
color
colorimage 544–545
compression, DCT algorithm impressive with 147
maps, **Indexed** color space use 239–241
spaces and 296–297
values 211
with a single source *(example)* 310
compressing (**DCTEncode**) *(table)* 86
current 176
data
compressing (**DCTEncode**) *(table)* 86
decompressing (**DCTDecode**) *(table)* 86
ReusableStreamDecode filter use 153
DCTDecode filter output as stream of 146
decompressing (**DCTDecode**) *(table)* 86
dependence, **WhitePoint** and **BlackPoint** values 225
dictionaries 297–304
data *(table)* 305
mask *(table)* 306
type 1 297–299
type 1 *(example)* 311
type 1 *(table)* 298
type 3 *(table)* 304
type 4 *(table)* 307
height, in scan lines, specifying in a **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
interpolation 301
stencil masking effect 303
masking 301–308
color key masking 307–308
explicit masking 303–307
LanguageLevel 1 and 2 mechanisms for 302
stencil masking 302–303
monochrome *(example)* 309–310
page
placement 413–416
placement, parameters *(table)* 414–416
painting 288–311
image 177, 607
requirements for 290
parameters 289–290
image(s) *(continued)*
PostScript support for CCITT fax standard designed for 143
sample representation 290–293
sampling
DCTDecode filter output 146
LZW support of predictor functions for 133
source, coordinate system 293–296
space, user space relationship to 294
trapping 453–454
using (examples) 308–314
ImageInternalTrapping entry
Trapping dictionary, entries *(table)* 448
imagemask operator 608
color space use 297
image dictionary use 297–304, 299–301
painting bitmapped character glyphs with 303
sample data sources 291
sample representation 290
stencil mask use 301
stencil masking 302–303
tiling pattern use *(table)* 252
uncolored tiling pattern use 257
ImageMatrix array
as image data dictionary entry *(table)* 305
as image mask dictionary entry *(table)* 306
as type 1 image dictionary entry *(table)* 298
as type 4 image dictionary entry *(table)* 308
ImageResolution entry
Trapping dictionary, entries *(table)* 449
images
threshold arrays compared with 489
ImageShift page device parameter 414
ImageToObjectTrapping entry
Trapping dictionary, entries *(table)* 448
ImageTrapPlacement entry
Trapping dictionary, entries *(table)* 449
ImageType entry
as image data dictionary entry *(table)* 305
as image mask dictionary entry *(table)* 306
as type 1 image dictionary entry *(table)* 298
as type 3 image dictionary entry *(table)* 304
as type 4 image dictionary entry *(table)* 307
ImageType resource category 98
(table) 92
imaging
model 14, 176–178
multiple contexts, gstate objects use with 181
ImagingBBox page device parameter 414
immediate execution 46–47

immediately evaluated names 121–123
ASCII encoding compared with binary object sequence encoding 167
binary object sequence encoding 166
specifying in a binary object sequence 169
syntax of 31
implementation
limits
(appendix) 737–743
exceeded error (**limitcheck**) 621
of resource categories, implementation dictionary use 100–102
Implementation entry
as form dictionary entry *(table)* 208
as type 1 pattern dictionary entry *(table)* 252
as type 2 pattern dictionary entry *(table)* 260
implicit
key lookup, dictionary stack search by 42
parameters, painting operators 177
resources 90, 98–99
overview *(table)* 91–92
inclusive or operation (**or**) 631
incremental definition 353
of fonts
CIDFont glyphs, **GlyphDirectory** use 375–376
CIDFont glyphs, Type 1 377
rules for 354
independence
device
color spaces, CIE-based 221
color spaces, overview 214
color specification as 210
coordinate system transformation enhancement of 184
graphics state parameters *(table)* 179–180
See also portability
user space vs device space 183
index operator 52, 610
Indexed color space 239–241
axial shading pattern (type 2) prohibited with 266
Decode array mapping 300
DeviceN use, to produce multitone 246–247
Function parameter of type 4 shading dictionary prohibited with 271
Function parameter of type 5 shading dictionary prohibited with 275
Function parameter of type 6 shading dictionary prohibited with 280
function-based shading (type 1) pattern prohibited with 265
radial shading pattern (type 3) prohibited with 268
remapping to CIE-based color space 238
rendering rules for shading patterns 264
Indexed color space *(continued)*
as special color space 214
ineofill operator 610
insideness testing 197
infill operator 610
insideness testing 197
initclip operator 611
usage guidelines 805
initgraphics operator 612
Install procedure use 426
usage guidelines 805
initialization
clipping path (**initclip**) 611
CTM, to default for current device (**initmatrix**) 613
device
page device parameters *(table)* 426–427
page setup and 426–432
graphics state (**initgraphics**) 612
initiation
of error handling 115–116
initmatrix operator 613
usage guidelines 805
ink
device color space specification 216
InkParams resource category 97
InkParams resource category description *(table)* 91
input
See also output
errors (**ioerror**) 618
files 73
arbitrary data source as (**SubFileDecode**) *(table)* 86
executing programs from (**exec**) 74
reading characters from (**read**) 73
reading lines from (**readline**) 73
reading strings from (**readstring**) 73
scanning for PostScript tokens (**token**) 73
standard 57
as execution environment component 73–87
file handling 80
handling 74
stream, creating positionable (**ReusableStreamDecode**) *(table)* 86
InputAttributes dictionary
managing 408–410
InputAttributes page device parameter 400
InsertSheet page device parameter 402
insideness testing
See also clipping; even-odd rule
ineofill 610
infill 610
instroke 613

insideness testing *(continued)*
inueofill 614
inufill 615
inustroke 616
operators *(list)* 520
painting use of 196–197
Install page device parameter 426
installing
dictionaries (**begin**) 536
InstanceType entry
as category implementation dictionary entry *(table)* 101
instroke operator 613
insideness testing 197
integer(s) 28
See also mathematics; numbers
binary object sequence encoding 166
converting
objects to (**cvi**) 566
to real numbers (**cvr**) 567
dividing (**idiv**) 605
numeric objects 37–38, 37
objects, execution handling 50
rounding (**round**) 652
shifting bits in binary representation of (**bitshift**) 539
as simple object *(table)* 34
strings use of 39
Intent entry
as **ReusableStreamDecode** dictionary entry *(table)* 155
intents
rendering
remapping device colors to CIE-based color space use 237
specifying for CIE-based color spaces 469–473
interaction
models, PostScript interpreter 15–17
interactive
applications
ByteOrder and **RealFormat** use 170
gstate objects use with 181–182
insideness testing 196
executive
control characters for *(table)* 21
invoking (**executive**) 81, 585
prompting user (**prompt**) 635
interpreter, use 20–21
InterleaveType entry
as type 3 image dictionary entry *(table)* 304
interleaving
blocks 304
of image sample data stream, as **DCTDecode** filter output 146
interleaving *(continued)*
JPEG standard alternative to, not useful for **image** operator 147
by row *(table)* 304
by sample *(table)* 304
in sample data 292
internaldict operator 614
International Commission on Illumination
See CIE (Commission Internationale de l'Éclairage)
Interpolate entry
as image data dictionary entry *(table)* 306
as image mask dictionary entry *(table)* 307
as type 1 image dictionary entry *(table)* 298
as type 4 image dictionary entry *(table)* 308
interpolation
algorithms, for gradient fills, color space sensitivity 264
bilinear *(example)* 111
exponent, type 2 function dictionary specification (**N**) *(table)* 113
exponential interpolation functions 107
type 2 function dictionary 112–113
type 2 function dictionary, entries specific to *(table)* 113
image, stencil masking effect 303
of images 301
as parametric value, compared with color space interpolation in gradient fills 263
interpreter
errors (**unregistered**) 712
interpreters
Emulator resource category *(table)* 91
interactive, control characters for *(table)* 21
interactive use 20–21
page description, **PDL** resource category 97
parameters
(appendix) 745–760
operators *(list)* 522
PostScript 24–25
application interactions with 15–22
interaction models 15–17
interrupt error 614
special handling 115
inueofill operator 614
encoded number string use 172
insideness testing 197
inufill operator 615
encoded number string use 172
insideness testing 197
inustroke operator 616
encoded number string use 172
insideness testing 197
invalidaccess error 616

invalidaccess error *(continued)*
file
access 80
as data sources and targets 124
object access 79
local VM reference by global VM object 60
invalidexit error 617
invalidfileaccess error 617
file access 80
invalidfont error 617
definefont failure 323
invalidrestore error 617
stack modification 61
inversion
inverse delta transform (**idtransform**) 606
matrix (**invertmatrix**) 618
of sample color intensities 300
stitching function use 114
of a transformation 189
transformation (**itransform**) 619
invertmatrix operator 618
invoking
interactive executive (**executive**) 81, 585
procedures, rule for 49
IODevice resource category 98
resourceforall use with 87
ioerror error 618
DCTEncode filter 128
DCTEncode issues 150
file access 80
invalid EOD indicator
ASCII85Decode 131
ASCIIHexDecode 130
procedure as data target and source precautions 125
string as data target exhaustion 126
write to closed data target 128
isFixedPitch entry
as entry in **FontInfo** dictionary *(table)* 327
ISO
JPEG (Joint Photographic Experts Group) standard
See JPEG (Joint Photographic Experts Group) standard
ISO 10646 Unicode standard
CMap use for encoding 366
ISOLatin1Encoding 618
ISOLatin1Encoding encoding array 95, 618
as Latin-text encoding scheme 330
(table) 773, 785
ItalicAngle entry
as entry in **FontInfo** dictionary *(table)* 327
iteration operators
cshow 551
iteration operators *(continued)*
exit 585
fonts (**kshow**) 619
for 596
loop 623
repeat 645
itransform operator 619

J

JFIF (JPEG File Interchange Format)
See JPEG (Joint Photographic Experts Group) standard
JIS Japanese encoding
CMap use for encoding 366
job 57
components of 69
encapsulation 70–72
LanguageLevel 1 (**exitserver**) 72–73
LanguageLevel 2 (**startjob**) 695
execution environment 68–72
initiating (**exitserver**) 586
server 68
dictionary (**serverdict**) 72, 657
operations (step sequence) 69
starting (**startjob**) 695
JobName user parameter
value *(table)* 750
Jog page device parameter 418
join between line segments (**setlinejoin**) 194
JPEG (Joint Photographic Experts Group) standard
compression of image data (**DCTEncode**) *(table)* 86
DCTDecode/**DCTEncode** filter use 146
decompression of image data (**DCTDecode**) *(table)* 86
ISO/IEC 10918-1, bibliographic reference 814
PostScript relationship to 150

K

K entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 144
kerning
See also font(s)
information available in AFM and ACRM files 333
keys
associating with values, in dictionaries on the dictionary stack (**def**) 54, 568
dictionary 41
searching for (**known**) 54, 619
pattern cell 252
as resource identifiers (**defineresource**) 90

known operator 54, 619
deprecated for locating operators in **systemdict** 43
kshow operator 55, 321, 619

L

language
control 97
support (**ControlLanguage**) *(table)* 91
Emulator resource category 98
(table) 91
natural
Localization resource category 97
LZW method suited to 133
Asian character sets, CID-keyed font use 364–390
support (**Localization**) *(table)* 91
page description 13–15
PDL resource category 97
PostScript
introduction *(chapter)* 1–10
programming language structure and components *(chapter)* 23–174
LanguageLevel 1
accessing external font resources 346
clipping as simulation of masking 302
color space restrictions 211
definefont 88
deletefile 77
dictfull error 41
dictionary capacity, fixed 53
encapsulation overriding (**exitserver**) 72–73
exitserver 70, 72–73
file 77
file system access (**file**) 77
filenameforall 77
fileposition 77
findfont 88
image facilities 288–289
path storage limits 190
renamefile 77
rendering support 458
save and **restore** as VM reclamation mechanisms 62
setfileposition 77
status 77
virtual memory, local 59
LanguageLevel 2
feature summary 731–735
LanguageLevel 3
feature summary 725–731
overview 5–7
LanguageLevel entry
as font dictionary entry *(table)* 325
languagelevel operator 620
LanguageLevels
compatibility strategies *(appendix)* 761–771
feature summary *(appendix)* 725–736
incompatibilities among 735
late binding 117
Latin-text
See also font(s)
Chamelion font format as implementation of 343
encoding schemes 330
standard Latin character set *(table)* 779–783
Latin-text fonts
ISOLatin1Encoding 618
ISOLatin1Encoding resource 95
lattice-form
Gouraud-shaded triangle meshes 261, 274–276
le operator 55, 620
LeadingEdge page device parameter 402
left parenthesis (()
special syntactic treatment of 27
left sidebearing
glyph 332
length
See also size; width
of composite object, retrieving (**length**) 53, 621
dictionary 41
of names 40
length operator 621
retrieving the number of elements in a composite object 53
less-than-or-equal relation
comparing objects for (**le**) 620
less-than relation
comparing objects for (**lt**) 623
LF (line feed)
as EOL, scanner handling of 74–75
\n as ASCII encoding *(list)* 29
numeric values *(table)* 27
libraries, procedure
See procedure(s), sets
LicenseID system parameter
value *(table)* 752
lifetime
of global virtual memory 60
of local virtual memory 59
limitcheck error 621
available storage exhausted by a path 190
clipping simulation of masking 302
font cache 382
halftone cell memory size 488
radix number cause 28
real number cause 28
limits
architectural *(table)* 739

limits *(continued)*
implementation
(appendix) 737–743
exceeded error (**limitcheck**) 621
memory *(table)* 741–742
miter joins, setting (**setmiterlimit**) 194
to path size 190
line feed (LF)
as EOL, scanner handling of 74–75
\n as ASCII encoding *(list)* 29
numeric values *(table)* 27
line(s)
See also path(s)
appending
to current path (**lineto**) 622
to current path (**rlineto**) 650
cap
as graphics state parameter *(table)* 180
retrieving (**currentlinecap**) 559
setting (**setlinecap**) 194, 673
color, setting (**sethsbcolor**) 194
dashed
as graphics state parameter *(table)* 180
currentdash 555
setdash 194, 666
end-of-line, conventions 74–75
erasing, control characters for interactive executive use *(table)* 21
join
as graphics state parameter *(table)* 180
retrieving (**currentlinejoin**) 559
setting (**setlinejoin**) 673
painting (**stroke**) 177, 700
reading (**readline**) 73, 639
redisplaying, control characters for interactive executive use *(table)* 21
straight, adding to current path (**lineto**) 191
stroking 193–194
width
retrieving (**currentlinewidth**) 559
setting (**setlinewidth**) 193, 674
(table) 179
%lineedit file
file handling 80
lineto operator 177, 191, 198, 622
operation code for encoded user paths *(table)* 201
LISP
atoms, PostScript names similar to 40
as PostScript influence 23
literals
array object 162
attribute
handling objects with 50
setting (**cvlit**) 566
literals *(continued)*
attribute *(continued)*
testing for (**xcheck**) 721
composite, delimiters for 58
object attribute 36–37
syntax of 31
text strings, ASCII encoding 29–30
ln operator 52, 622
load operator 54, 622
loading
elements of an array or packed array onto the operand stack (**aload**) 54, 528
resources into VM 105
values from dictionary stack onto the operand stack (**load**) 54, 622
local
standard dictionaries *(table)* 65
VM 59–61
reference by global VM object prohibited 60
reference by global VM object prohibited, **systemdict** exceptions 66
save and **restore** operations 61–62
localization
support (**Localization**) *(table)* 91
Localization resource category 97
(table) 91
locating
values on the dictionary stack (**where**) 54, 718
location
page, decoupling from graphical element description 185
log operator 52, 623
logarithms
common (**log**) 623
natural (**ln**) 622
logical operations
See boolean
loop operator 55, 623
looping
See iteration
loops
context 585
exiting (**exit**) 585
lossy
compression of image data (**DCTEncode**) *(table)* 86
DCTEncode/**DCTDecode** filters characterized as 147
decompression of image data (**DCTDecode**) *(table)* 86
filters, disadvantages 308
LowBitFirst entry
as LZW dictionary entry *(table)* 134
LZWEncode use 136–137
lt operator 55, 623

luminance
representation, **CIEBasedA** use 228
LZW (Lempel-Ziv-Welch) adaptive compression method
compressing data with (**LZWEncode**) *(overview table)* 85
decompressing data encoded with (**LZWDecode**) *(overview table)* 85
dictionary *(table)* 133
encoding sequence *(example)* 135
Flate encoding compared with 138
Flate encoding use 137
FlateDecode filter use, overview *(overview table)* 86
FlateEncode filter use, overview *(overview table)* 85
predictor functions use with 139–141
URL for information on 133
LZWDecode filter 85, 132–137
LZWEncode filter 85, 133–137

M

machine
generation, binary encodings used for 25
representation
ByteOrder and **RealFormat** use 170
of numeric objects, accessibility of 38
makefont operator 624
composite font handling 364
Font resource category instances as operands to 93
scaling fonts with 315
makepattern operator 625
creating patterns with 249
Pattern resource category use 95
management
of memory 56–72
manipulation
of matrices 187–189
ManualFeed page device parameter 402
ManualSize array
as **OutputDevice** dictionary entry *(table)* 455
mapping
algorithm, CMap 388–389
algorithms
FMapType selection of 358–360
FMapType *(table)* 360
characters 358–362
color, functions, as CIE-based color rendering dictionary component 460
colors, to device color, as rendering step 457
control for Type 0 fonts with 360–362
with the **Decode** array *(figure)* 112
fonts, **FMapType** resource category *(table)* 92
image space, into user space 294–296
a procedure over a composite object (**forall**) 53, 597
mapping *(continued)*
color, selecting colors from, **Indexed** color space use 239–241
Margins page device parameter 415
mark 44
binary object sequence encoding 166
errors (**unmatchedmark**) 711
objects 44
execution handling 50
removing, from operand stack (**cleartomark**) 52
as simple object *(table)* 34
mark operator 52, 625
See also square bracket, left ([) operator
mark objects created by 44
marking
an operand stack position (**mark**) 52, 625
MaskColor array
as type 4 image dictionary entry *(table)* 307
MaskDict dictionary
as type 3 image dictionary entry *(table)* 304
masking
image, LanguageLevel 1 and 2 mechanisms for 302
image dictionary *(table)* 306
of images 301–308
color key masking 307–308
explicit masking 303–307
stencil 301
imagemask 608
of stencil 302–303
matching
requests with attributes, in media selection 403–407
mathematics
arithmetic
add 527
idiv 605
mul 627
neg 628
sqrt 692
sub 701
ceiling 540
common logarithms (**log**) 623
errors (**undefinedresult**) 709
exponentiation (**exp**) 586
floor 594
natural logarithms (**ln**) 622
operators
(list) 508
overview 52
random number generation
rand 637
retrieving current state (**rrand**) 652
srand 692
remainder (**mod**) 627
rounding integers (**round**) 652

mathematics *(continued)*
trigonometric functions
atan 535–536
cos 550
sin 692
truncating numbers (**truncate**) 705
matrix
See also CTM (current transformation matrix)
concatenation of operations on 188–189
image space transformation 294–296
inversion (**invertmatrix**) 618
operators
concat 547
concatmatrix 547
currentmatrix 559
defaultmatrix 569
identmatrix 605
(list) 517–518
matrix 626
overview 175
representation and manipulation 187–189
rotation (**rotate**) 651
transformation 184–186
applying (**transform**) 703
inverting (**idtransform**) 606
(table) 208
user path operator use 205
Matrix array
as form dictionary entry *(table)* 208
as type 1 shading dictionary entry *(table)* 265
matrix operator 626
MatrixA array
as **CIEBasedA** dictionary entry *(table)* 229
MatrixABC array
as **CIEBasedABC** dictionary entry *(table)* 223
as type 1 CIE-based CRD entry *(table)* 463
MatrixLMN array
calculation of 226
as **CIEBasedA** dictionary entry *(table)* 230
as **CIEBasedABC** dictionary entry *(table)* 224
as type 1 CIE-based CRD entry *(table)* 463
MatrixPQR array
as type 1 CIE-based CRD entry *(table)* 465
MaxDictStack user parameter
value *(table)* 750
MaxDisplayAndSourceList system parameter
value *(table)* 752
MaxDisplayList system parameter
value *(table)* 752
MaxExecStack user parameter
value *(table)* 750
MaxFontCache system parameter
value *(table)* 752
MaxFontItem user parameter
value *(table)* 750
MaxFormCache system parameter
value *(table)* 752
MaxFormItem user parameter
value *(table)* 750
MaxImageBuffer system parameter
value *(table)* 752
maximum
dictionary capacity, retrieving (**maxlength**) 54, 626
maxlength operator 54, 626
MaxLocalVM user parameter
value *(table)* 750
MaxOpStack user parameter
value *(table)* 750
MaxOutlineCache system parameter
value *(table)* 752
MaxPatternCache system parameter
value *(table)* 752
MaxPatternItem user parameter
value *(table)* 750
MaxScreenItem user parameter
value *(table)* 750
MaxScreenStorage system parameter
value *(table)* 752
MaxSeparations page device parameter 421
MaxSourceList system parameter
value *(table)* 752
MaxStoredScreenCache system parameter
value *(table)* 752
MaxSuperScreen user parameter
halftone-setting operator impact 482
value *(table)* 750
MaxUPathCache system parameter
value *(table)* 752
MaxUPathItem user parameter
value *(table)* 750
media
deferred, selection of 411
destinations 416
roll-fed, parameters *(table)* 412–413
selection 398–412
parameters *(table)* 400–403
recovery policies and 436–438
sources 398
special, handling of 407–408
MediaClass page device parameter
media selection parameter *(table)* 402
OutputDevice dictionary entry *(table)* 455
MediaColor page device parameter 401
MediaPosition page device parameter 403
MediaWeight page device parameter 401

memory
See also garbage collection
as execution environment component 56–72
limits *(table)* 741–742
management 56–72
composite object creation issues 53
reclamation of, **save** and **restore** functions 62
virtual
See VM (virtual memory)
VM (virtual memory), operators *(list)* 514
meshes
Coons patch, as type 6 shading 261, 277–283
data, for shading dictionaries, **ReusableStreamDecode** filter use with 153
free-form Gouraud-shaded triangle, as type 4 shading 261, 270–274
lattice-form Gouraud-shaded, as type 5 shading 261, 274–276
tensor-product patch, as type 7 shading 261, 283–287
metrics
glyphs
changing the 350–351
information 331–333
modifying 321
Metrics dictionary
modifying glyph metrics with 350–351
as Type 0 CIDFont dictionary entry *(table)* 373
as Type 1 font dictionary entry *(table)* 326
as Type 2 CIDFont dictionary entry *(table)* 379
as Type 42 font dictionary entry *(table)* 347
Metrics2 dictionary
modifying glyph metrics with 350–351
as Type 0 CIDFont dictionary entry *(table)* 374
as Type 1 font dictionary entry *(table)* 326
as Type 2 CIDFont dictionary entry *(table)* 379
as Type 42 font dictionary entry *(table)* 347
MIDVector entry
as Type 0 font dictionary entry *(table)* 358
MinFontCompress user parameter
value *(table)* 750
MirrorPrint page device parameter 415
misregistration
trapping as solution to 439
miter joins limit
See also line(s)
as graphics state parameter *(table)* 180
retrieving (**currentmiterlimit**) 560
setting (**setmiterlimit**) 194, 675
mod operator 52, 627
modal fonts 363
model(s)
color
additive (RGB and HSB) 217
contrasted with color space 423
model(s) *(continued)*
color *(continued)*
HSB 217
process color model (**ProcessColorModel**) 420, 422–424
RGB 217
subtractive, CMYK 218
subtractive, tints as, in **Separation** color spaces 242
of color, vision, CIE-based color spaces use for 220–238
data, elements and characteristics 23
DeviceN, precautions on using 424
imaging 14, 176–178
interaction, PostScript interpreter 15–17
painting 176
of PostScript interpreter 68
modifying
encoding vectors 349–350
fonts 348–356
glyph metrics 350–351
initial virtual memory 70–72
rendering priorities, rendering intent use for 469–473
unit lengths independently (**translate**) 185
user space 185–187
monochrome
CCITT fax encoding used for compressing 143
images *(example)* 309–310
stencil masking (**imagemask**) 608
monospaced fonts 320
moveto operator 177, 191, 627
adding segments to a path 191
operation code for encoded user paths *(table)* 201
setting font position with 315
user path 198
structuring 199
moving
user space origin (**translate**) 185, 704
mul operator 52, 627
multiple
byte character encodings, CID-keyed font use 364–390
execution contexts
concurrent access management 57
Display PostScript System manuals 811
standard error file shared among 81
standard I/O files private to 81
imaging contexts, gstate objects use with 181
MultipleDataSources entry
as image data dictionary entry *(table)* 305
as image mask dictionary entry *(table)* 306
as type 1 image dictionary entry *(table)* 298
as type 4 image dictionary entry *(table)* 308
multiplication
matrix, concatenation performed by 189
numbers (**mul**) 627
multitone color 245

N

N entry
as type 2 function dictionary entry *(table)* 113
name(s)
arrays, binary object sequence encoding 166
binary object sequence encoding 166
binding, early binding of (**bind**) 117–123, 118–121, 538–539
booleans, binary object sequence encoding 166
converting strings to (**cvn**) 566
creating 40
encoded system 168–169
as binary tokens 160
equality, testing for 40
errors (**undefined**) 709
evaluation, immediate, syntax of 31
executable
effects of executing 24
execution handling 50–51
handling *(example)* 49
semantics of 36
as first-class data objects 40
halftone, retrieving (**GetHalftoneName**) 599
immediately evaluated 121–123
ASCII encoding compared with binary object sequence encoding 167
binary object sequence encoding 166
specifying in a binary object sequence 169
integers, binary object sequence encoding 166
length of 40
literal vs. executable 36
marks, binary object sequence encoding 166
names, binary object sequence encoding 166
nulls, binary object sequence encoding 166
numbers, binary object sequence encoding 166
objects 40–41
operator
as **systemdict** dictionary keys 43
not reserved in PostScript 23
PostScript product, retrieving (**product**) 634
real numbers, binary object sequence encoding 166
searches, environment for 45
as simple object *(table)* 34
strings, binary object sequence encoding 166
strings and, interchangeable in an dictionary 41
syntax 31
system name 168
encodings *(appendix)* 795–799
translation of, by **ResourceFileName** key 103–104
named
files 77–80
resources 87–105
naming conventions
devices 78
file system 78
font 328
native color space 473
process color model, specifying 422–424
natural language
Asian character sets, CID-keyed font use 364–390
Localization resource category 97
LZW method suited to 133
support (**Localization**) *(table)* 91
natural logarithms
ln 622
ne operator 55, 628
neg operator 52, 628
negation
of numbers (**neg**) 628
NegativePrint page device parameter 415
nesting
of composite fonts 363–364
of **save** and **restore** 62
neutral density 444
newerror entry
as **$error** dictionary entry *(table)* 116
newline characters 26
handling 30
newpath operator 177, 628
beginning paths with 191
execform invocation of 209
user path invocation of 205
no-op
executable null object use for 51
noaccess operator 56, 628
nocurrentpoint error 629
undefined current point 191
None colorant name
Separation color space use 243
nonzero winding number rule 195
See also clipping; even-odd rule; insideness testing
even-odd rule compared with 194
insideness testing
infill 610
inufill 615
painting, user paths (**ufill**) 708
not equals relation
comparing objects for (**ne**) 628
not operator 55, 629
.notdef character name
beginnotdefrange operator 385
CIDFont equivalent for 369
encoding vector use 330

.notdef character name *(continued)*
endnotdefrange operator 385
replacing 351
undefined character handling, in CMap mapping operation 390
Notice entry
as entry in **FontInfo** dictionary *(table)* 327
nul (null) character
numeric values *(table)* 27
null
binary object sequence encoding 166
device
nulldevice 630
nulldevice, usage guidelines 805–806
executable, execution handling 51
objects 44
null 630
(table) 34
null operator 630
nulldevice operator 630
usage guidelines 805–806
NullEncode filter 86, 156
null
bind not usable with 119
dictionary keys not permitted to be 41
name, **systemdict** null object use 44
number(s) 37–38
See also mathematics
absolute value (**abs**) 527
adding (**add**) 527
binary object sequence encoding 166
dividing
div 574
idiv 605
encoded number strings
as binary tokens 171–172
homogeneous number array use 162, 171
operators that use 172
fixed-point 160
as binary tokens 160
graphics state parameters, handling of 178
homogeneous number arrays 161
as binary tokens 161–162
encoded number string use 171
multiplying (**mul**) 627
negation of (**neg**) 628
nonzero winding number rule 195
radix 28
random
generating (**rand**) 637
generating (**srand**) 692
retrieving current state (**rrand**) 652
real 28
representation
in homogeneous number array *(table)* 162
of binary tokens 169–170
rounding (**round**) 652
subtracting (**sub**) 701
syntax 28
truncating (**truncate**) 705
unique ID 336
extended 337
NumCopies page device parameter 417

O

object(s) 24
array 38
attributes of 35–37
binary object sequences 156, 163–168
binary tokens compared with 158, 169
encoded system names in 168–169
setting format (**setobjectformat**) 677
writing to standard output (**printobject**) 634
writing (**writeobject**) 721
boolean 38
comparing
for equality (**eq**) 580
for greater-than-or-equal relation (**ge**) 598
for greater-than relation (**gt**) 604
for inequality (**ne**) 628
for less-than-or-equal relation (**le**) 620
for less-than relation (**lt**) 623
composite 34–35, 34
access attributes 37
as graphics state parameters, handling of 178
binary object sequence structure recommendations 167
copying (**copy**) 53
graphics state restrictions 215
pushing/popping effects 46
retrieving elements of (**get**) 53, 598
retrieving the number of elements in (**get**) 53
storing elements of (**put**) 53, 635
VM as pool for values of 57
converting to strings (**cvs**) 568
copying
on the operand stack (**dup**) 51, 575
simple vs. composite 35
creating, in VM *(example)* 60
data 34–45
dictionary 41–42
exchanging, on the operand stack (**exch**) 51, 581
executable
effects of executing 24
exec 582
files, execution handling 50
names, execution handling 50–51

object(s) *(continued)*
executable *(continued)*
null, execution handling 51
sources of 24
strings, execution handling 50
external, introduction into VM 88
file 43–44, 43
files represented by 73
retrieving the current (**currentfile**) 87
fontID
as a simple object *(table)* 34
as an object type 45
execution handling 50
format, retrieving (**currentobjectformat**) 560
gstate, as an object type 45
literal array 162
mark 44
name 40–41
double angle brackets as 33
null 44
numeric 37
integer 37–38
real 37–38
operator 42–43
execution handling 51
packed array 39
removing, from the top of the operand stack (**pop**) 51, 633
save 44
simple 34–35, 34
names as 40
source of, for immediate execution 47
storing into an array (**astore**) 54, 535
string 39–40
tokens as components of 26
types 34–45, 34
retrieving as a name object (**type**) 705
(table) 34
user 67–68, 67
defining (**defineuserobject**) 67, 571
defining (**defineuserobject**) *(example)* 68
executing (**execuserobject**) 67, 584
executing (**execuserobject**) *(example)* 68
manipulating *(example)* 68
undefining (**undefineuserobject**) 67, 711
UserObjects 713
writing
text representation, to standard output (**=**) 87
text representation, to standard output (**==**) 87
octal character codes
ASCII encoding *(list)* 29
radix number representation of 28
usage 30
opacity
overprint control of 247–248
opaque
painting model use of 176
operand stack
accessing 46
as an array (**index**) 52, 610
clearing
clear 52
elements above the highest mark (**cleartomark**) 52, 541
copying
objects on (**dup**) 51, 575
portions of (**copy**) 51, 548
counting elements
above the highest mark (**counttomark**) 52, 551
count 52, 550
exchanging objects on (**exch**) 51, 581
as execution environment component 56
loading
elements of an array or packed array onto (**aload**) 54, 528
values from dictionary stack onto (**load**) 54, 622
marking position on 44
mark 52, 625
as one of five execution state stacks 45
operators
(list) 508
overview 51–52
printing contents, using **=** (**stack**) 87
procedure handling use 50
removing
mark from (**cleartomark**) 52
objects from the top of (**pop**) 51, 633
treating as circular queue (**roll**) 52, 650
writing, to standard output (**stack**) 692
operands
order of, execution handling 47
procedures as, for control constructs 49
operating system
named file handling integration with 77
operators
arithmetic
(list) 508
overview 52
array
(list) 509
overview 52–55
attribute
(list) 512
overview 56
basic, overview of 51–56
bitwise
(list) 511
overview 55
boolean
(list) 511

operators *(continued)*
boolean *(continued)*
(overview) 55
(chapter) 505–723
clipping path 193
CMap, in the **CIDInit** procedure set 384–385
control
(list) 512
overview 55–56
conversion
(list) 512
overview 56
coordinate system *(list)* 517–518
detailed reference descriptions, in alphabetical order 524–723
device setup *(list)* 521
dictionaries *(list)* 510
dictionary, overview 52–55
executable, semantics of 36
executing, effects of 24
execution, handling 51
file
basic, overview 73–76
(list) 513–514
miscellaneous, overview 87
font *(list)* 521–522
form *(list)* 520
glyph *(list)* 521–522
graphics
display and printer facilities *(chapter)* 175–290
overview of main groups 175–176
graphics state
device-dependent *(list)* 516–517
device-independent *(list)* 515
insideness-testing 197
(list) 520
mathematical, overview 52
mathematics *(list)* 508
matrix *(list)* 517–518
names
as **systemdict** dictionary keys 43
not reserved in PostScript 23
objects 42–43
operand stack
(list) 508
overview 51–52
output *(list)* 521
packed array
(list) 509
overview 52–55
painting 177
(list) 519–520
path
clipping 193
construction 177, 191
operators *(continued)*
path construction *(list)* 518–519
pattern *(list)* 520
polymorphic, overview 52–55
rectangle, user path operators relationship to 205–206
relational
(list) 511
overview 55
resource 88–90
(list) 514
as simple object *(table)* 34
string
(list) 511
overview 52–55
substitution, **bind** use of 118–119
systemdict dictionary use for *(table)* 66
type
(list) 512
overview 56
for Type 4 CIDFonts 381–382
usage guidelines *(appendix)* 801–810
user path 204–204
(list) 198
virtual memory 61–62
VM (virtual memory) *(list)* 514
or (inclusive or) operator 55, 631
order
byte, token type specification 170
matrix multiplication significance 189
of operands, execution handling 47
path, reversing (**reversepath**) 649
Order entry
as type 0 function dictionary entry *(table)* 109
Ordering entry
as **CIDSystemInfo** dictionary entry *(table)* 368
Orientation page device parameter 412
origin
device space, location of 182
glyph 331
user space, physical page relationship 183
ostack array
as **$error** dictionary entry *(table)* 116
outlines
glyph, treating as a path 319
output
See also input
delivery specification
for pages 416–419
for pages, parameters *(table)* 417–419
device dictionary 455–456
echoing characters on, enabling/disabling (**echo**) 575
errors (**ioerror**) 618

output *(continued)*
files 73
arbitrary data target as (**NullEncode**) *(table)* 86
writing characters to (**write**) 73, 720
writing strings to (**writestring**) 73
operators
(list) 521
overview 176
standard 57
as execution environment component 73–87
file handling 80
handling 74
printing stack objects to (**pstack**) 635
writing binary object sequences (**printobject**) 634
writing characters to (**print**) 633
writing operand stack contents (**stack**) 692
writing text representation of an object to (=) 87
writing text representation of an object to (==) 87
structured 172–174
format 172
OutputAttributes page device parameter 419
OutputDevice page device parameter
(table) 417
OutputDevice resource category 97
dictionary entries *(table)* 455
(table) 91
OutputFaceUp page device parameter 418
OutputType page device parameter 417
overflow
dictionary, handling of 41
overprinting
control 247–248
DeviceN color space use 245
as graphics state parameter *(table)* 180
parameter
retrieving (**currentoverprint**) 560
setting (**setoverprint**) 247, 677
overriding
glyph metrics 350–351
resource categories 99
overview
of operators, general-purpose 51–56
overwriting
subintervals of
arrays (**putinterval**) 53, 636
strings (**putinterval**) 53, 636

P

PackBits (Apple Macintosh)
RunLengthEncode filter similar to 142
packed arrays 39
See also array(s)
as composite object *(table)* 34
copying (**copy**) 53, 548
creating (**packedarray**) 53, 631
elements
loading onto the operand stack (**aload**) 54, 528
retrieving (**get**) 53, 598
retrieving the number of (**length**) 53, 621
storing (**put**) 53, 635
mapping a procedure over (**forall**) 53, 597
mode
retrieving the setting (**currentpacking**) 54, 560
setting (**setpacking**) 54, 678
operators
(list) 509
overview 52–55
procedure objects as 33
subintervals of, creating objects that share (**getinterval**) 53
packedarray operator 631
as composite object constructor 58
creating packed arrays with 53
page
*See also **PDL** resource category*
adjusting 437
delivery specification 416–419
parameters *(table)* 417–419
descriptions
decoding filters use in 82
graphics state parameters appropriate for *(table)* 179
graphics state stack management of 181
language interpreters, **PDL** resource category 97
languages 13–15
PostScript program structure 17–18
resource access restrictions 103
devices
See page devices
device parameters
See page device parameters
erasing (**erasepage**) 581
image placement 413–416
parameters *(table)* 414–416
outputting (**showpage**) 549–550
physical, default user space origin relationship to 183
rendering (**showpage**) 177, 690
sets 419
setup, device initialization and 426–432
page devices
capabilities (**OutputDevice**) *(table)* 91
controlling *(chapter)* 391–456
details dictionaries 397
dictionary 394–397
retrieving (**currentpagedevice**) 560

page devices *(continued)*
initialization *(table)* 426
OutputDevice resource category 97
parameters
See page device parameters
retrieving name of (**GetPageDeviceName**) 600
setting (**setpagedevice**) 679
using 393–398
page device parameters 398–439
AdvanceDistance 413
AdvanceMedia 412
BeginPage 427, 427–428
categories *(table)* 399
Collate 417
color support 419–426
(table) 420–422
CutMedia 413
DeferredMediaSelection 403
Duplex 416
EndPage 427, 428–430
handling of unsatisfied requests 432–439
HWResolution 414
ImageShift 414
ImagingBBox 414
initialization *(table)* 426
InputAttributes 400
InsertSheet 402
Install 426
Jog 418
LeadingEdge 402
ManualFeed 402
Margins 415
MaxSeparations 421
media selection *(table)* 400–403
MediaClass 402
MediaColor 401
MediaPosition 403
MediaWeight 401
MirrorPrint 415
NegativePrint 415
NumCopies 417
Orientation 412
OutputAttributes 419
OutputDevice 417
OutputFaceUp 418
OutputType 417
PageDeviceName 420
PageOffset 415
PageSize 401
Policies 433
ProcessColorModel 420, 422–424
recovery policies *(table)* 433
roll-fed media *(table)* 412–413
RollFedMedia 402, 412
page device parameters *(continued)*
SeparationColorNames 421
SeparationOrder 421
Separations 420
Trapping 422, 447
TrappingDetails 422, 442
TraySwitch 403
Tumble 416
UseCIEColor 422
PageCount system parameter
value *(table)* 752
PageDeviceName page device parameter 420
PageOffset page device parameter 415
PageSize page device parameter
media selection parameter *(table)* 401
OutputDevice dictionary entry *(table)* 455
Policies dictionary *(table)* 434
painting 193–197
See also drawing; filling; rendering
areas, with patterns, **Pattern** color space use 238–239
characters (**show**) 329
clipping path impact on 192
even-odd rule use in
eofill 580
ueofill 708
forms
execform 582
procedure for *(table)* 208
glyphs
glyphshow 602
show 316, 690
widthshow 718
xshow 722
xyshow 722
yshow 723
images 288–311
image 177, 607
requirements for 290
in imaging model 176
lines (**stroke**) 700
model 176
nonzero winding number rule use in (**ufill**) 708
operators
compared with path construction operators 191
halftone dictionary use 486
(list) 519–520
overview 177
overview 175
patterns 248–287, 250
procedures for 249
shading (**shfill**) 689
regions (**fill**) 194, 589
user path (**ustroke**) 714

painting *(continued)*
user paths
operators for 204
ufill 197
ustroke 197
PaintProc procedure
creating a pattern cell with 250
EPS reference, **ReusableStreamDecode** filter use with 153
as form dictionary entry *(table)* 208
tiling pattern use 253–254
as type 1 pattern dictionary entry *(table)* 251
PaintType entry
as Type 0 CIDFont dictionary entry *(table)* 373
as Type 1 font dictionary entry *(table)* 326
as type 1 pattern dictionary entry *(table)* 251
as Type 2 CIDFont dictionary entry *(table)* 378
as Type 42 font dictionary entry *(table)* 347
parameters
device
IODevice resource category *(table)* 91
retrieving (**currentdevparams**) 87, 555
setting (**setdevparams**) 87, 667
graphics state
device-dependent *(table)* 180–181
device-independent *(table)* 179–180, 179
implicit, painting operators 177
interpreter
(appendix) 745–760
LanguageLevel 3 *(table)* 728
operators *(list)* 522
page device
See page device parameters
system
See system parameters
user
See user parameters
parent font 357
parentheses (())
ASCII encoding *(list)* 29
as literal text string delimiters 29
passwords
startjob use 70
patch meshes
as control surfaces 278
Coons, as type 6 shading 261, 277–283
tensor-product, as type 7 shading 261, 283–287
path(s)
See also insideness testing
appending
curves to (**curveto**) 564
curves to (**rcurveto**) 638
glyphs to (**charpath**) 319
path(s) *(continued)*
appending *(continued)*
lines to (**lineto**) 622
lines to (**rlineto**) 650
bounding box
retrieving (**pathbox**) 631
setting (**setbox**) 657
clipping 178, 192–193, 192
computing intersection path (**clip**) 193
current, insideness testing disregard of 197
operators 193
path stack 192
restoring (**cliprestore**) 193
retrieving (**clippath**) 193
saving (**clipsave**) 193
setting (**initclip**) 611
stack, as one of five execution state stacks 45
closing (**closepath**) 544
construction 189–193
operators 177, 191
operators compared with painting operators 191
operators *(list)* 518–519
operators, overview 175
creating (**newpath**) 628
current 190–192
encapsulating 192
enclosed regions
even-odd rule 196
insideness testing 196–197
nonzero winding number rule 195
enumerating elements (**pathforall**) 632
flattening (**flattenpath**) 593
as graphics state parameter *(table)* 179
insideness testing 196–197
new, beginning (**newpath**) 191
operators, operands as coordinates in user space 183
replacing (**strokepath**) 700
reversing (**reversepath**) 649
subpaths 190
closing (**closepath**) 191, 194
creating (**moveto**) 627
creating (**rmoveto**) 650
treating glyph outlines as 319
user
See user paths
using repeatedly 192
pathbox operator 631
pathforall operator 55, 632
Pattern color space 238–239, 254
remapping to CIE-based color space 238
shading dictionary ColorSpace parameter value prohibited from being 262
as special color space 214
uncolored tiling pattern use 257

Pattern resource category 95
(table) 91
pattern(s) 248–287
cell 248
tiling an area with 250
color, specifying *(table)* 251
color space, selecting 249
coordinate system 252
creating 249
dash
as graphics state parameter *(table)* 180
currentdash 555
setdash 194, 666
defining 249
dictionaries
components of 249
type 1 *(table)* 251
type 2 *(table)* 260
function use 106
in imaging model 176
key pattern cell 252
LanguageLevel 2 use for simulation of masking 302
operators *(list)* 520
painting 250
painting areas with, **Pattern** color space use 238–239
painting with, procedures for 249
shading 248, 259–290
painting (**shfill**) 689
rendering rules 264
stitching function use 114
type 1 (function-based) 261, 265–266
type 2 (axial) 261, 266–268
type 3 (radial) 261, 267–270
type 4 (free-form Gouraud-shaded triangle mesh) 261, 270–274
type 5 (lattice-form Gouraud-shaded triangle mesh) 261, 274–276
type 6 (Coons patch mesh) 261, 277–283
type 7 (tensor-product patch mesh) 261, 283–287
types of 265–290
stroked line
operators that determine 194
setdash 194
tiling 248, 250–259
colored 254–257
uncolored 257–259
uncolored, restrictions on **PaintProc** 215
using 249–250
patterns
creating (**makepattern**) 625
setting (**setpattern**) 680
PatternType entry
as type 1 pattern dictionary entry *(table)* 251
as type 2 pattern dictionary entry *(table)* 260
PatternType resource category 98
(table) 92
PDF (Portable Document Format)
bibliographic reference 812
function use in 106
PostScript relationship 9
PDL resource category 97
Emulator resource category superseded by 97, 98
(table) 91
percent sign (%)
comments initiated by 27
special syntactic treatment of 27
perception
human color, CIE-based color spaces modeling of 459
Perceptual rendering intent
characteristics *(table)* 470
performance
See also storage
as **bind** benefit 119
compression/decompression filters use to improve 84
DCTEncode issues 150
Flate encoding compared with LZW 138
glyph rendering issues 334
requirement, binary object sequences suitable for 156, 163
user paths, lack of side effects contribution to 198
persistence
of global virtual memory 60
issues, in global VM use 66
of local virtual memory 59
physical
page, default user space origin relationship to 183
pipeline 83
closing, for encoding filters 128
establishing 124
garbage collection and **restore** impact on 129
ReusableStreamDecode filter use 154
PNG (Portable Network Graphics) standard
predictor functions based on 139
as W3C standard 139
points 183
Bézier control, type 6 shading pattern data source use 280
current
error (**nocurrentpoint**) 629
establishing (**moveto**) 191
retrieving (**currentpoint**) 561
insideness testing 196–197
reference, glyph 331
policies
recovery
media selection and 436–438
page device parameter *(table)* 433

Policies page device parameter 433–435
PolicyNotFound entry
  Policies dictionary *(table)* 433
PolicyReport procedure
  Policies dictionary *(table)* 435
  unsatisfied parameter request handling with 438–439
polymorphism operators 52–55
pool
  global VM storage 59
  local VM storage 59
pop operator 51, 633
popping
  dictionaries, off the dictionary stack (**end**) 54, 577
portability
  See also device(s), independence
  barrier to, filter use 86
  character names advantages over character codes 330
  device color space limitation on 216
  string object values 40
position
  as graphics state parameter *(table)* 179
positioning
  files (**setfileposition**) 80, 668
  glyph (**cshow**) 321
  glyphs 320–321
    ashow 320, 534–535
    awidthshow 320, 536
    cshow 551
    kshow 321, 619
    show 320, 690
    widthshow 320, 718
    xshow 320, 722
    xyshow 321, 722
    yshow 320, 723
PostScript
  interpreter 24–25
    application interactions with 15–22
    interaction models 15–17
  language
    binary encoding, details 156–174
    introduction *(chapter)* 1–10
    programming language structure and components *(chapter)* 23–174
    syntax 25–34
  product
    retrieving name (**product**) 634
    retrieving revision level (**revision**) 649
  scanner, operations 25–26
  version, retrieving (**version**) 716
precautions
  executable attribute 36
  graphics state objects, global VM issues 67
  immediate evaluation of names 122–123
precision
  numeric objects 38
  Predictor entry
  as **FlateEncode**/**FlateDecode** dictionary entry *(table)* 138
  as LZW dictionary entry *(table)* 134
  as predictor-related entry in LZW and Flate filter dictionaries *(table)* 141
predictor functions
  LZW or Flate dictionary entries related to *(table)* 141
  LZW support of 133
  PNG-based 139
  TIFF-based 139
    compared with PNG-based 140
  with LZW and Flate filters 139–141
PrefEnc array
  as Type 0 font dictionary entry 362
    (table) 358
print
  formats, translating from other 19–20
  job
    See job
  server, as PostScript interpreter model 68
print operator 74, 633
  output compared with structured output 172
PrinterName system parameter
  value *(table)* 753
printing
  operand stack contents
    using == (**pstack**) 87
    using = (**stack**) 87
  stack objects (**pstack**) 635
printobject operator 634
  binary object sequence writing 173
Private dictionary
  as **FDArray** array entry *(table)* 374
  of **FDArray** dictionary, entries *(table)* 375
  Type 0 CIDFont compared with Type 1 375
  as Type 1 font dictionary entry *(table)* 326
procedure(s) 24, 38
  array semantics compared with 32
  bound, benefits of using 118
  as data sources and targets 124–126
  dictionaries used to manage 41
  execution handling 50
  forms compared with 206
  functions compared with 106
  IdiomSet resource category *(table)* 91
  interpreter handling of *(example)* 48
  invoking, rule for 49
  libraries
    See procedure(s), sets
  object, self-contained, user path as 197

procedure(s) *(continued)*
as operands for control constructs 49
PostScript, Type 3 fonts handled through 337
procedure substitution dictionary, idiom recognition use 119–121
ProcSet resource category 95
(table) 91
as sample data source 291
semantics of 36
sets 95
BitmapFontInit *(table)* 96
CIDInit *(table)* 96
ColorRendering *(table)* 96
FontSetInit *(table)* 96
LanguageLevel 3 operators defined in *(table)* 726
standard, LanguageLevel 3 *(table)* 96
Trapping *(table)* 96
substitution, **IdiomSet** resource category use 97
syntax 32–33
process colors 218, 241
DeviceN, precautions on using 424
model
color space contrasted with 423
specifying (**ProcessColorModel**) 420, 422–424
ProcessColorModel *(table)* 473
ProcessColorModel page device parameter
color support 422–424
(table) 420
native color space determined by 473
OutputDevice dictionary entry *(table)* 456
page device dictionary, **Separation** color space use 243
ProcSet resource category 95
ColorRendering procedure set, customizing CRD selection use 472
(table) 91
product
dictionaries (**statusdict**) 696
PostScript
retrieving name (**product**) 634
retrieving revision level (**revision**) 649
product operator 634
product-dependent
dictionary, **statusdict** dictionary *(table)* 65
resources
ControlLanguage resource category 97
HWOptions resource category 97
Localization resource category 97
PDL resource category 97
programming style
garbage collection impact on 63
programs
control constructs
See control, operators
encapsulated, subfile use for error recovery in 151
programs *(continued)*
execution of
exec 74
run 653
structure of, PostScript page description 17–18
structured
conventions, further information on, "PostScript Language Document Structuring Conventions" 813
save and **restore** intended for 62
prolog
modifications for correct global VM use 67
persistence issues, in global VM use 66
save and **restore** functions 62
in structured document 17–18
prompt operator 635
proportional fonts 320
proprietary
CRDs 460
font formats 322
halftone dictionaries 499–500
protocols
bit-stream, LZW as, implications for **LowBitFirst** key 136
communication 76
pstack operator 87, 635
pushing
dictionaries, onto the dictionary stack (**begin**) 54
put operator 53, 635
putinterval operator 636
overwriting subintervals of a composite object 53

Q

QFactor entry
as **DCTEncode** dictionary entry *(table)* 149
QuantTables entry
as **DCTEncode** dictionary entry *(table)* 148
queue
circular, treating as stack portion as (**roll**) 52, 650
quit operator 637
usage guidelines 806

R

radial shading pattern 261, 267–270
radix numbers 28
See also mathematics; numbers
converting to strings (**cvrs**) 567
rand operator 637
as pseudo-random number generator 52

random numbers
generating
rand 637
retrieving current state (**rrand**) 652
srand 692
random-access file
creating a creating positionable stream for (**ReusableStreamDecode**) *(table)* 86
range
errors (**rangecheck**) 638
of function 107
numeric objects 38
Range array
as function dictionary entry *(table)* 108
RangeA array
as **CIEBasedA** dictionary entry *(table)* 229
RangeABC array
as **CIEBasedABC** dictionary entry *(table)* 223
as type 1 CIE-based CRD entry *(table)* 464
rangecheck error 638
array out-of-bounds 38
DataSource size 110
edge flags in type 4 shading 273
function dimensionality inconsistency 107
type 6 halftone dictionary input file termination 492
user path structuring 199
RangeDEF array
as **CIEBasedDEF** dictionary entry *(table)* 233
as **CIEBasedDEFG** dictionary entry *(table)* 235
RangeHIJ array
as **CIEBasedDEF** dictionary entry *(table)* 233
RangeHIJK array
as **CIEBasedDEFG** dictionary entry *(table)* 235
RangeLMN array
as **CIEBasedA** dictionary entry *(table)* 230
as **CIEBasedABC** dictionary entry *(table)* 223
as type 1 CIE-based CRD entry *(table)* 463
RangePQR array
as type 1 CIE-based CRD entry *(table)* 465
raster
image processor
See RIP (Raster Image Processor)
output devices 11–12
rcheck operator 56, 638
rcurveto operator 191, 198, 638
operation code for encoded user paths *(table)* 201
read access attribute
setting (**readonly**) 640
testing (**rcheck**) 638
read operator 639
characters treated as data not programs 157
EOL transparent to 75
read operator *(continued)*
reading, characters 73
read-only access
as object attribute 37
readhexstring operator 639
reading, hexadecimal strings 73
reading
asynchronous, precautions 154
characters
hexadecimal (**readhexstring**) 639
read 73, 639
execution stack (**execstack**) 583
files
access convention 79
in arbitrary order (**setfileposition**) 80
hexadecimal characters (**readhexstring**) 639
hexadecimal strings (**readhexstring**) 73
lines (**readline**) 73, 639
realtime clock (**realtime**) 641
strings
hexadecimal (**readhexstring**) 73
readstring 73, 641
readline operator 639
EOL handling 75
reading, lines 73
readonly operator 56, 640
readstring operator 641
characters treated as data not programs 157
EOL transparent to 75
reading, strings 73
real numbers 28
See also mathematics; numbers
binary object sequence encoding 166
converting, objects to (**cvr**) 567
converting to integers (**cvi**) 566
execution handling 50
numeric objects 37–38, 37
representation of, in binary tokens and binary object sequences 170
as simple object *(table)* 34
RealFormat system parameter
machine representation of real numbers 170
value *(table)* 753
realtime clock
reading (**realtime**) 641
realtime operator 641
rearranged
fonts 364
reclamation
of memory, **save** and **restore** functions 62
recognition
of idioms 119–121

record
as irrelevant in PostScript environment 142
specifying as operand of **RunLengthEncode** filter 143
recordstacks entry
as **$error** dictionary entry *(table)* 117
recovery
policies
media selection and 436–438
page device parameter *(table)* 433
rectangles
clipping (**rectclip**) 641
drawing (**rectstroke**) 643
encoded number string use 171
filling (**rectfill**) 642
operators, user path operators relationship to 205–206
rectclip operator 641
encoded number string use 172
rectangle operand formats 206
rectfill operator 642
encoded number string use 172
(example) 171
rectangle operand formats 206
rectstroke operator 643
encoded number string use 172
rectangle operand formats 206
recursion
tail, interpreter facilitation of 50
redefining
resource categories 99
redisplaying
lines, control characters for interactive executive use *(table)* 21
reference point
glyph 331
regions
painting (**fill**) 194, 589
path-enclosed
even-odd rule 196
insideness testing 196–197
nonzero winding number rule 195
Registry entry
as **CIDSystemInfo** dictionary entry *(table)* 368
regular resources 90
overview *(table)* 91
relational
operators
(list) 511
overview 55
RelativeColorimetric rendering intent
characteristics *(table)* 470
remainder
discarding (**idiv**) 605
remainder *(continued)*
retrieving (**mod**) 627
remapping
device colors to CIE-based color spaces 237–238
removeall operator 644
Type 4 CIDFont use 381
removeglyphs operator 644
Type 4 CIDFont use 381
removing
dictionaries from the dictionary stack
all except for permanent entries (**cleardictstack**) 54, 541
end 54, 577
entries from a dictionary (**undef**) 54, 708
files (**deletefile**) 572
fonts (**undefinefont**) 323
glyphs
removeall 644
removeglyphs 644
mark, from operand stack (**cleartomark**) 52
objects, from the top of the operand stack (**pop**) 51, 633
operand stack elements
above the highest mark (**cleartomark**) 52, 541
clear 52
resources (**undefineresource**) 89
undercolor 476–477
renamefile operator 644
file system access by 77
operand formats 78
special file name not used by 78
renaming
files (**renamefile**) 644
renderbands operator
usage guidelines 806
rendering
See also painting
(chapter) 457–504
color
(figure) 213
models specification 419
overview 210
setting (**setcolorrendering**) 663
ColorRendering resource category 96
ColorRenderingType resource category *(table)* 92
dictionary, CRD (**findcolorrendering**) 591
intents
customizing with CRD selection 472–473
specifying for CIE-based color spaces 469–473
(table) 470
pages (**showpage**) 177, 690
process color model applicable only to 423
render color spaces 461

rendering *(continued)*
scan conversion 12–13
as potential glyph rendering component 334
details 501–504
overview 458
shading pattern rules 264
steps involved in 457
RenderingIntent entry
as type 1 CIE-based CRD entry *(table)* 468
RenderTable array
as type 1 CIE-based CRD entry *(table)* 467–468, 467
repeat operator 55, 645
replacing
glyphs 351–352
paths (**strokepath**) 700
user path (**ustrokepath**) 715
values in a dictionary on the dictionary stack (**store**) 54, 698
representation
binary, of integers, shifting bits in (**bitshift**) 539
binary data, ASCII-encoded hexadecimal strings 30–31
FontSet resource 344–345
functions, dictionary use for 106
GlyphDirectory, recommendations for 356
of images 290–293
machine
ByteOrder and **RealFormat** use 170
of numeric objects, accessibility of 38
of matrices 187–189
non-decimal base numbers, radix number use for 28
of number, by binary tokens and binary object sequences 169–170
number, in homogeneous number array *(table)* 162
of objects
behavior 34–45
internal 34–45
syntactic 25
of sampling, type 0 function dictionary use 108
user path operators 199
request dictionary 394
reserved words
non-existent in PostScript 43
resetfile operator 645
ReusableStreamDecode filter use 154
resetting
See also restoring
files (**resetfile**) 645
resolution
page device (**HWResolution**) 414
resource(s)
categories 90–99
creating 99–102
Generic 102–103
resource(s) *(continued)*
categories *(continued)*
implementation dictionary 100–102
implementation dictionary *(table)* 101
LanguageLevel 3 *(table)* 727
Category *(table)* 92
CIDFont 94
(table) 91
CMap 94
(table) 91
ColorRendering 96
(table) 91
ColorRenderingType 98
(table) 92
ColorSpace 96
(table) 91
ColorSpaceFamily 98
ControlLanguage 97
(table) 91
defining
Category *(table)* 92
defineresource 89, 570–571
deleting (**undefineresource**) 89
dynamic loading, as global VM intended use 66
Emulator 98
(table) 91
Encoding 94
(table) 91
enumerating (**resourceforall**) 646
errors (**undefinedresource**) 709
as files 103–105
Filter 98
(table) 91
FMapType 98
(table) 92
Font
description 93
(table) 91
fonts as, in LanguageLevels 2 & 3 323
FontSet 94, 344–345
(table) 91
FontType 98
(table) 92
Form 95
(table) 91
FormType 98
(table) 92
FunctionType 98
(table) 92
Generic *(table)* 92
Halftone 96
(table) 91
HalftoneType 98
(table) 92

resource(s) *(continued)*
HWOptions 98
(table) 91
IdiomSet 97
(table) 91
ImageType 98
(table) 92
implicit 90, 98–99
overview *(table)* 91–92
InkParams 97
(table) 91
inquiring about, **resourceforall** and **resourcestatus** use for 90
IODevice 98
(table) 91
keys as identifiers for (**defineresource**) 90
Localization 97
(table) 91
named 87–105
operators 88–90
(list) 514
OutputDevice 97
(table) 91
Pattern 95
(table) 91
PatternType 98
(table) 92
PDL 97
(table) 91
ProcSet 95
(table) 91
regular 90
overview *(table)* 91
retrieving
findresource 89, 592
resourceforall 85, 87, 89
ShadingType 98
(table) 92
status, retrieving (**resourcestatus**) 89, 647
TrapParams 97
(table) 91
TrappingType 98
(table) 92
undefining (**undefineresource**) 710
VM use 89
ResourceFileName procedure
as category implementation dictionary entry *(table)* 101
translating resource names into file names 103–104
r**resourceforall** operator 88, 646
findresource compared with 90
implementation dictionary 100
implicit resource handling 98–99
r**resourceforall** operator *(continued)*
ResourceForAll entry, category implementation dictionary, use by *(table)* 101
resourcestatus compared with 93
retrieving
all available device parameter sets 87
all available resource instances in a particular category 89
the complete set of supported filters 85
ResourceForAll procedure
as category implementation dictionary entry *(table)* 101
resourcestatus operator 88, 647
findresource compared with 90
implementation dictionary 100
implicit resource handling 98–99
as preferred method of determining resource availability 93
resource file header use 105
ResourceStatus key use by, category implementation dictionary *(table)* 101
retrieving status information about a resource 89
ResourceStatus procedure
as category implementation dictionary entry *(table)* 101
restore operator 648
CloseTarget and **CloseSource** use by 129
closing files with 80
deallocation issues 60
encapsulation by 69
errordict changes subject to 115
garbage collection compared with 63–64
global virtual memory unaffected by 59
incremental definition of font impact 354
invalid execution error (**invalidrestore**) 617
local VM operations 61–62
pipeline impact 129
procedure as data target and source precautions 125
ReusableStreamDecode filter use 154
save objects manipulated by 44
VM impact 72
restoring
See also resetting
clipping path (**cliprestore**) 193
graphics state
grestore 603
grestoreall 603
VM state (**restore**) 648
retrieving
See also accessing
alternate CRD (**GetSubstituteCRD**) 600
black-generation function (**currentblackgeneration**) 552
bounding box, for current path (**pathbox**) 631

retrieving *(continued)*
clipping path (**clippath**) 193
color spaces (**currentrgbcolor**) 561–562
current file object (**currentfile**) 87
device
dependent parameters (**currentdevparams**) 87
retrieving name of (**GetPageDeviceName**) 600
serial number (**serialnumber**) 657
all dictionaries, from the dictionary stack (**dictstack**) 54, 573
elements of composite objects (**get**) 53, 598
encoding vector (**findencoding**) 591
files
currentfile 556
filenameforall 588
position (**fileposition**) 589
status (**status**) 696
flatness parameter (**currentflat**) 556
fonts
cache parameters (**currentcacheparams**) 552
currentfont 557
findfont 592
rootfont 651
selectfont 656
glyph, width (**stringwidth**) 699
graphics state
currentgstate 558
gstate 604
halftone
name (**GetHalftoneName**) 599
screen (**currenthalftone**) 558
screen (**currentscreen**) 562
line
cap parameter (**currentlinecap**) 559
join parameter (**currentlinejoin**) 559
width parameter (**currentlinewidth**) 559
maximum dictionary capacity (**maxlength**) 54, 626
miter joins limit (**currentmiterlimit**) 560
the number of elements in composite objects (**length**) 53
object
format (**currentobjectformat**) 560
type as a name object (**type**) 705
overprinting parameter (**currentoverprint**) 560
packed array mode (**currentpacking**) 560
page device dictionary (**currentpagedevice**) 560
PostScript product
name (**product**) 634
revision level (**revision**) 649
resources (**findresource**) 89, 592
smoothness parameter (**currentsmoothness**) 563
status
file (**status**) 696
information for a file (**bytesavailable**) 87
information for a file (**status**) 87
retrieving *(continued)*
status *(continued)*
of a resource (**resourcestatus**) 89, 647
VM (**vmstatus**) 717
stroke adjustment parameter (**currentstrokeadjust**) 563
system parameters (**currentsystemparams**) 563
time (**usertime**) 714
transfer functions
(**currentcolortransfer**) 555
(**currenttransfer**) 563
trapping dictionary (**currenttrapparams**) 564
undercolor removal function (**currentundercolorremoval**) 564
user parameters (**currentuserparams**) 564
user path, cache status (**ucachestatus**) 707
values, from the dictionary stack (**load**) 54, 622
VM allocation mode (**currentglobal**) 557
ReusableStreamDecode filter 86, 153–156
as **DataSource** entry in type 0 function dictionary *(table)* 109
dictionary *(table)* 155
EOD not applicable to 129
filter file use precautions 124
SubFileDecode filter use with 151
reversepath operator 649
reversing
paths (**reversepath**) 649
revision level
PostScript product, retrieving (**revision**) 649
revision operator 649
Revision system parameter
value *(table)* 753
RGB (red-green-blue) color space
as additive color model 217
calibrated from scanners, **CIEBasedDEF** and **CIEBasedDEFG** use 232
conversion
between **DeviceRGB** and **DeviceGray** 474–475
to **DeviceCMYK** from **DeviceRGB** 475–477
to **DeviceRGB** from **DeviceCMYK** 477
device (**DeviceRGB**) 98, 211, 217–218
DeviceRGB operator use
currentrgbcolor 218
setrgbcolor 218
remapping **DeviceRGB** to CIE-based color space 237–238
retrieving (**currentrgbcolor**) 561–562
setting
sethsbcolor 672
setrgbcolor 681
stroked line color (**setrgbcolor**) 194

right parenthesis ( ))
\) as ASCII encoding *(list)* 29
special syntactic treatment of 27
RIP (Raster Image Processor)
in-RIP trapping 439–454
rlineto operator 191, 198, 650
operation code for encoded user paths *(table)* 201
rmoveto operator 191, 198, 650
operation code for encoded user paths *(table)* 201
roll operator 52, 650
roll-fed media
parameters *(table)* 412–413
RollFedMedia page device parameter 402, 412
root font 357
rootfont operator 651
accessing graphics state font parameter 318
currentfont compared with 318
rotate operator 184, 651
CTM manipulation by 189
user space modification by 185
rotation
matrix
notation 188
rotate 651
user path caching not possible for (**rotate**) 203
round operator 52, 652
rounding
integers (**round**) 652
Rows entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 145
as **DCTEncode** dictionary entry *(table)* 148
rrand operator 52, 652
rules
See also algorithms; guidelines
CIDFont use, as current font 369
even-odd 196
clipping (**eoclip**) 579
ineofill 197
insideness testing (**ineofill**) 610
inueofill 197
nonzero winding number rule compared with 194
painting areas (**eofill**) 580
painting user paths (**ueofill**) 708
filling, nonzero winding number 195
GlyphDirectory use with TrueType font data 356
incremental definition of
Encoding array 354
fonts 354
interpreter execution, executable array handling 48–49
nesting of font mapping algorithms 363–364
rules *(continued)*
nonzero winding number
even-odd rule compared with 194
insideness testing (**infill**) 610
insideness testing (**inufill**) 615
painting user paths (**ufill**) 708
rendering, of shading patterns 264
scan conversion 502–503
trapping
black 451–452
normal 449–451
run operator 87, 653
RunLengthDecode filter 84, 86, 142
RunLengthEncode filter 86, 142–143

S

sampled image
See image(s)
sampling
data
decoding of 299–301
functions, **ReusableStreamDecode** filter 153
functions, **ReusableStreamDecode** filter use 153
functions
(example) 111
type 0 function dictionary 108–112
horizontal, **DCTEncode** dictionary entry *(table)* 148
images, LZW support of predictor functions for 133
representation 290–293
type 0 function dictionary use 108
size of sample, type 0 function dictionary specification *(table)* 109
vertical, **DCTEncode** dictionary entry *(table)* 148
saturation 217
Saturation rendering intent
characteristics *(table)* 470
save objects 44
as composite object *(table)* 34
execution handling 50
save operator 653
encapsulation by 69
errordict changes subject to 115
garbage collection compared with 63–64
global virtual memory unaffected by 59
incremental definition of font impact 354
local VM operations 61–62
nesting impact on **startjob** operations 71
outermost execution by job server 69
save objects created by 44
VM impact 72

saving
clipping path (**clipsave**) 193
graphics state (**gsave**) 603
state, of VM (**save**) 653
scale
in fixed-point numbers 160
scale operator 184, 654
CTM manipulation by 189
user space modification by 185
scalefont operator 654
composite font handling 364
Font resource category instances as operands to 93
scaling glyphs with 315
scaling
fonts
makefont 315, 624
scalefont 315, 654
selectfont 656
matrix notation 188
user coordinate space (**scale**) 654
user path caching not possible for (**scale**) 203
scan
conversion 12–13
as potential glyph rendering component 334
as rendering step 458
details 501–504
rules 502–503
of input files, for PostScript tokens (**token**) 73
order, source image 294
of strings, for PostScript tokens (**token**) 702
scanner
calibrated RGB from, **CIEBasedDEF** and **CIEBasedDEFG** use 232
comments treatment 27
PostScript, operations 25–26
scheck operator 655
screens
See halftone(s), screens
script
save and **restore** functions 62
in structured document 17–18
SDBytes entry
as **Private** dictionary entry *(table)* 375
search operator 55, 655
searching
for dictionary key (**known**) 54, 619
for name, environment for 45
strings
for initial substring (**anchorsearch**) 529
for substrings in (**search**) 655
segments
connected 189
selectfont operator 656
composite font handling 364
font selection efficiency 317–318
selecting
fonts 316–318
selectfont 656
media
deferred 411
for page devices 398–412
parameters *(table)* 400–403
recovery policies and 436–438
semantics
array vs. procedures 32
of binary tokens 158
BuildChar compared with **BuildGlyph** 340
CIE-based color spaces 220–221
of color values 211–248
data sources, filters and images compared 291
of executable objects 36
execution 46–51
of filters 123–156
form dictionary 207
Separation color space 241–245
DeviceN color space compared with 246
remapping to CIE-based color space 238
rendering rules for shading patterns 264
as special color space 214
SeparationColorNames page device parameter
color support *(table)* 421
page device dictionary, **Separation** color space use 243
SeparationOrder page device parameter 421
separations
colorants compared with 242
device colorants and 424–426
Separations page device parameter 420
sequence(s)
binary object 156, 163–168
binary tokens compared with 158, 169
encoded system names in 168–169
setting format (**setobjectformat**) 677
writing to standard output (**printobject**) 634
writing (**writeobject**) 721
files as 73
serial
communications protocol, independent of PostScript language 76
serial number
device, retrieving (**serialnumber**) 657
serialnumber operator 657
server
job 68
dictionary (**serverdict**) 72, 657
operations (step sequence) 69

server *(continued)*
print, as PostScript interpreter model 68
serverdict dictionary 657
contents of 72
setbbox operator 198, 657
operation code for encoded user paths *(table)* 201
ucache required to precede *(example)* 202–203
user path structuring 199
setblackgeneration operator 658
black generation with 476
usage guidelines 806
setcachedevice operator 659
BuildGlyph procedure use 340
setcachedevice2 operator 659
BuildGlyph procedure use 340
setcachelimit operator 660
setcacheparams operator 661
setcharwidth operator 661
BuildGlyph procedure use 340
setcmykcolor operator 194, 211, 215, 662
DeviceCMYK use 219
setcolor operator 211, 215, 662
CIE-based color space use 221
DeviceCMYK use 219
DeviceGray use 220
DeviceN color space use 246
DeviceRGB use 218
mapping sample data to color component values for 291
pattern use 250
Separation color space use 244
setcolorrendering operator 663
ColorRendering resource category use 96
usage guidelines 806
setcolorscreen operator 664
halftone screen definition 481
usage guidelines 806
setcolorspace operator 211, 214, 665
CIEBasedA use 228
CIEBasedABC use 222
CIEBasedDEF use 233
CIEBasedDEFG use 233
DeviceCMYK use 219
DeviceGray use 220
DeviceN use 246
DeviceRGB use 218
Pattern use 249
Separation use 242, 244
setcolortransfer operator 666
specifying transfer functions 479
type 1 halftone dictionary overriding of 488
usage guidelines 806
setdash operator 666
stroked line pattern 194
setdevparams operator 87, 667
errorinfo array use *(table)* 116
setfileposition operator 668
file system access by 77
positioning a file object with 80
ReusableStreamDecode filter use 154
setflat operator 669
usage guidelines 806
setfont operator 670
setglobal operator 670
usage guidelines 806
VM allocation controlled by 60
setgray operator 211, 215
DeviceGray use 220
mapping sample data to color component values for 291
stroked line color 194
setgray operators 671
setgstate operator 671
clipping path stack management 192
gstate object management 182
usage guidelines 806–807
sethalftone operator 671
halftone dictionary installation by 482
Halftone resource category use
specifying transfer functions 479
type 1 halftone dictionary selected by 487
usage guidelines 807
sethsbcolor operator 194, 211, 215, 672
DeviceRGB use 218
setlinecap operator 194, 673
setlinejoin operator 194, 673
setlinewidth operator 178, 193, 674
setmatrix operator 675
usage guidelines 807–808
setmiterlimit operator 194, 675
setobjectformat operator 677
disabling binary encodings with 157
number representation of binary object sequence controlled by 173
setoverprint operator 677
overprint control by 247
usage guidelines 808
setpacking operator 678
setting array packing mode 54
setpagedevice operator 679
See also page devices
errorinfo array use *(table)* 116
media property handling with 184
usage guidelines 808

setpattern operator 194, 215, 680
Pattern resource category use 95
selecting patterns with 250
setrgbcolor operator 211, 215, 681
DeviceRGB use 218
stroked line color 194
sets
character
Expert *(table)* 787–789
standard Latin *(table)* 779–783
Symbol *(table)* 792–793
font (**FontSet**) 344–345
procedure
See procedure(s), sets
setscreen operator 681
halftone screen definition 481
as type 1 halftone dictionary alternative 487
usage guidelines 808
setshared operator 682
usage guidelines 808
setsmoothness operator 682
interpolation use, in shading color spaces 263
usage guidelines 808
setstrokeadjust operator 683
automatic stroke adjustment 503–504
stroked line thickness consistency 194
setsystemparams operator 684
setting
access attributes
executeonly 584
readonly 640
to noaccess (**noaccess**) 628
binary object sequence format (**setobjectformat**) 677
black-generation function (**setblackgeneration**) 658
bounding box (**setbox**) 657
clipping path, to default setting (**initclip**) 611
color
rendering (**setcolorrendering**) 663
setcolor 662
spaces (**setcmykcolor**) 662
spaces (**setcolorspace**) 665
spaces (**setgray**) 671
spaces (**sethsbcolor**) 672
spaces (**setrgbcolor**) 681
transfer function (**setcolortransfer**) 479, 666
transfer function (**settransfer**) 479
CTM
setmatrix 675
to default for current device (**initmatrix**) 613
dash pattern (**setdash**) 666
device, parameters (**setdevparams**) 667
device-dependent parameters (**setdevparams**) 87
executable attribute (**cvx**) 568
setting *(continued)*
file position (**setfileposition**) 668
flatness parameter (**setflat**) 669
font cache
parameters (**setcacheparams**) 661
setcachedevice 659
setcachedevice2 659
size (**setcachelimit**) 660
fonts (**setfont**) 670
global VM mode (**setglobal**) 670
glyph width (**setcharwidth**) 661
graphics state
setgstate 671
to default values (**initgraphics**) 612
halftone screen
setcolorscreen 664
sethalftone 671
halftone screens (**setscreen**) 681
line, width (**setlinewidth**) 674
lines
cap (**setlinecap**) 673
join (**setlinecap**) 673
literal attribute (**cvlit**) 566
miter joins limit (**setmiterlimit**) 194, 675
overprinting parameter (**setoverprint**) 247, 677
packed array mode (**setpacking**) 54, 678
page device parameters (**setpagedevice**) 679
patterns (**setpattern**) 680
rendering priorities, rendering intent use for 469–473
smoothness parameter (**setsmoothness**) 682
stroke adjustment parameter (**setstrokeadjust**) 683
system parameters (**setsystemparams**) 684
transfer functions (**settransfer**) 685
trapping, zone (**settrapzone**) 686
trapping parameters, dictionary (**settrapparams**) 685
undercolor removal function (**setundercolorremoval**) 476, 687
user parameters (**setuserparams**) 687
user path
cache (**setucacheparams**) 686
cache (**ucache**) 707
VM allocation threshold (**setvmthreshold**) 688
settransfer operator 685
halftone dictionary overriding of 488
specifying transfer functions 479
usage guidelines 808–809
settrapparams operator 685
TrapParams resource category use 97
settrapzone operator 686
setucacheparams operator 686
setundercolorremoval operator 687
undercolor removal 476
usage guidelines 809

setup
of pages, device initialization and 426–432
setuserparams operator 687
setvmthreshold operator 688
sfnts array
as Type 2 CIDFont dictionary entry *(table)* 378
as Type 42 font dictionary entry *(table)* 347
shading
axial, as type 2 shading 261, 266–268
color space considerations 263–264
Coons patch meshes, as type 6 shading 261, 277–283
dictionaries 261–263
entries common to all *(table)* 262
type 1 (function-based) *(table)* 265
type 2 (axial) *(table)* 266
type 3 (radial) *(table)* 268
type 4 (free-form Gouraud-shaded triangle mesh) *(table)* 270
type 5 (lattice-form Gouraud-shaded triangle mesh) *(table)* 275
type 6 (Coons patch mesh) *(table)* 279
type 7 (tensor-product patch mesh) 283–287
function use 106
function-based, as type 1 shading 261
Gouraud
free-form Gouraud-shaded triangle meshes, as type 4 shading 261, 270–274
lattice-form Gouraud-shaded triangle meshes, as type 5 shading 261, 274–276
pattern types 265–290
type 1 (function-based) 261, 265–266
type 2 (axial) 261, 266–268
type 3 (radial) 261, 267–270
type 4 (free-form Gouraud-shaded triangle mesh) 261, 270–274
type 5 (lattice-form Gouraud-shaded triangle mesh) 261, 274–276
type 6 (Coons patch mesh) 261, 277–283
type 7 (tensor-product patch mesh) 261, 283–287
patterns 248, 259–290
painting (**shfill**) 689
ReusableStreamDecode filter use with 153
radial, as type 3 shading 261, 267–270
ShadingType resource category *(table)* 92
stitching function use 114
tensor-product patch meshes, as type 7 shading 261, 283–287
Shading dictionary
as type 2 pattern dictionary entry *(table)* 260
ShadingType entry
as shading dictionary entry *(table)* 262
ShadingType resource category 98
shadowing
of operator definitions, in **systemdict** dictionary 43
shadows
diffuse achromatic, **BlackPoint** use 224
shape library
Chamelion font format as implementation of 343
shapes
complex, creating with type 4 shading patterns 273
shareddict operator 689
SharedFontDirectory dictionary 689
sharing
access management provision 57
composite object values, between composite object copies 35
values, among composite objects 58
shfill operator 689
shading pattern use 260
tiling patterns with gradient fill use 261
ShiftIn entry
as Type 0 font dictionary entry *(table)* 358
shifting
bits (**bitshift**) 539
ShiftOut entry
as Type 0 font dictionary entry *(table)* 358
show operator 177, 690
character painting steps 329
CIDFont restrictions 369
composite font use 357–359
FMapType 9 composite font use 387–390
in garbage collection example 63
glyph positioning 320
painting glyphs with 316
showpage operator 690
copypage compared with 549–550
PaintProc procedure prohibited from using 209
rendering by 177
use of 392
side effects
defineresource 324
of file operations, impact on file objects 44
forms free of 206
operator usage, guidelines for avoiding unintended 801–810
prohibited to
BuildGlyph font procedure 339
DecodeABC procedures 223
DecodeLMN procedures 223
functions 106
halftone dictionary procedures 484
Indexed color space lookup procedures 240
PaintProc form procedure 209
PaintProc pattern procedures 253
Separation color space tintTransform operator 244
user paths free of 198
simple objects 34–35
names as 40

simple objects *(continued)*
types of *(table)* 34
sin operator 52, 692
size
See also length; width
font cache, setting (**setcachelimit**) 660
page (**PageSize**) 401
reduction, compression/decompression filters use to improve 84
Size array
as type 0 function dictionary entry *(table)* 109
slash (/)
as literal name delimiter 31
special syntactic treatment of 27
slash, double (//)
as immediately evaluated name delimiter 31
sliding
traps 452–453
SlidingTrapLimit entry
Trapping dictionary, entries *(table)* 448
smoothness
as graphics state parameter *(table)* 181
parameter
retrieving (**currentsmoothness**) 563
setting (**setsmoothness**) 682
snapshots
state, save objects use for 44
of VM state, as **save** operations 61
source(s)
color images (examples) 310
data 123–127
image, coordinate system 293–296
of objects, for immediate execution 47
sample data
multiple, handling 292
object types 291
organization of 292
premature termination handling 292
SP (space) character
numeric values *(table)* 27
space(s)
character (SP), numeric values *(table)* 27
color
See color spaces
device 182
halftones defined in 481
tiling of, in a type 16 halftone dictionary *(figure)* 497
transformation of user space to 184
unaffected by changes to CTM 191
user space and 182–184
image, user space relationship to 294
user 182
coordinates in 184
space(s) *(continued)*
user *(continued)*
default 183
device space and 182–184
image space relationship to 294
modifying 185–187
origin, physical page relationship 183
transformation to device space 184
spacing
glyph, modification difficulties 350
text
ashow 320, 534–535
awidthshow 320, 536
cshow 321, 551
kshow 321, 619
show 320
widthshow 320, 718
xshow 320, 722
xyshow 321, 722
yshow 320, 723
of tiling patterns, specifying *(table)* 252
special
color spaces 238–247
files 80–82
specification
color *(figure)* 212
of color, overview 210
specular highlights
compared with diffuse achromatic highlight 224
sphere
depiction of, radial shading use 269
spot
colorants 424
colors 241
functions 484–489
SpotFunction procedure
as type 1 halftone dictionary entry *(table)* 487
sqrt operator 52, 692
square bracket. left ([) operator
array construction role 32
mark objects created by 44, 524
special syntactic treatment of 27
square bracket, right (]) operator
See also array(s)
as array constructor 32, 524
special syntactic treatment of 27
square bracket ([]) operators
creating arrays with 54
literal array object created by 36
square root (**sqrt**) 692
srand operator 52, 692
stability
of user path, due to encapsulation 197

stack operator 87, 692
stack(s) 45–46
  call, execution stack use as 45
  clipping path 45, 192
  dictionary 42, 45, 56
    accessing 46
    associating keys with values in dictionaries on the (**def**) 54, 568
    contents of 42
    counting the number of dictionaries on (**countdictstack**) 54, 551
    executable name handling use 50–51
    locating values from (**where**) 54, 718
    popping dictionaries off (**end**) 54, 577
    pushing dictionaries onto (**begin**) 54, 536
    removing all dictionaries except for permanent entries (**cleardictstack**) 54, 541
    replacing values in dictionaries on (**store**) 54, 698
    retrieving all dictionaries from (**dictstack**) 54, 573
    retrieving values from (**load**) 54, 622
  errors
    stackoverflow 693
    stackunderflow 693
  execution 25, 45, 57
    accessing 46
    counting entries on (**countexecstack**) 551
    executable file handling use 50
    executable string handling use 50
    executing top element of (**exec**) 582
    reading (**execstack**) 583
  as execution environment component 56–72
  graphics state 181–182
    as one of five execution state stacks 45
  local VM management like 59
  not restored by **resourceforall** 100
  operand
    See operand stack
  PaintProc procedure prohibited from modifying 209
  printing, objects to standard output (**pstack**) 635
  removing, objects from the top of (**pop**) 633
  unmodified by **restore** 61
stackoverflow error 693
stackunderflow error 693
standard(s)
  CCITT 86, 143
  CIE, color spaces based on 220–238
  CIE 1976 L*u*v space, **CIEBasedDEF** and **CIEBasedDEFG** use 232
  CIEBasedA support of (examples) 230–231
  CIEBasedABC support of 222
    (examples) 225–228
  dictionaries 65–67
    global *(table)* 66
    local *(table)* 65
standard(s) *(continued)*
  encodings, **StandardEncoding** resource 94
  error file, **file** handling 80
  fax
    CCITTFaxDecode 143–146
    CCITTFaxEncode 143–146
  filters 84–86
    overview *(table)* 85
  input file 44, 57
    as execution environment component 73–87
    file handling 80
    handling 74
  ISO 10646 (Unicode), CMap use for encoding 366
  JPEG
    *See **DCTEncode** filter*
  Latin character set *(table)* 779–783
  output
    printing stack objects to (**pstack**) 635
    writing binary object sequences (**printobject**) 634
  output file 44, 57
    as execution environment component 73–87
    file handling 80
    handling 74
    writing characters to (**print**) 633
    writing operand stack contents (**stack**) 692
    writing text representation of an object to (=) 87
    writing text representation of an object to (==) 87
  PNG, predictor functions based on 139
  procedure sets, LanguageLevel 3 *(table)* 96
  TIFF
    LZW method use 133
    predictor functions based on 139
StandardEncoding array 94, 693
  (table) 773, 784
start operator 693
StartData operator 694
starting
  execution (**start**) 693
  job (**startjob**) 695
startjob operator 695
  encapsulation overriding by 70
  guidelines for use 70
  operations 70–72
  usage guidelines 809
StartJobPassword system parameter
  value *(table)* 753
StartupMode system parameter
  value *(table)* 753
state
  execution, stacks used to manage 45
  graphics
    See graphics state
  of local VM, snapshot of, as **save** operation 61

state *(continued)*
VM
restoring (**restore**) 648
saving (**save**) 653
%statementedit file
file handling 80
interactive executive use 81
statements
aborting, control characters for interactive executive use *(table)* 21
syntax of 81
static
formats, dynamic vs. 14–15
status
files, retrieving (**status**) 696
resource, retrieving (**resourcestatus**) 89, 647
retrieving
for a file (**bytesavailable**) 87
for a file (**status**) 87
user path cache, retrieving (**ucachestatus**) 707
VM, retrieving (**vmstatus**) 717
status operator 87, 696
file system access by 77
operand formats 78
statusdict dictionary 696
as standard local dictionary *(table)* 65
usage guidelines 809
%stderr file
file handling 80
%stdin file
file handling 80
%stdout file
file handling 80
stencil
mask 301
images 302–303
monochrome (**imagemask**) 608
treating a tiling pattern as *(table)* 252
StepLimit entry
ColorantZoneDetails dictionary *(table)* 454
Trapping dictionary, entries *(table)* 447
stitching functions 107, 113
type 3 function dictionary 113–114
entries specific to *(table)* 114
stop operator 56, 697
default error handling use 116
encapsulating the effects of (**stopped**) 697
stopped operator 56, 697
default error handling use 116
storage
See also performance; VM (virtual memory)
allocation in VM, by composite object constructors 58
storage *(continued)*
as scarce resource
binary tokens suitable for 156
compression/decompression filters value in managing 84
graphics state handling issues 182
Indexed color space advantages for 239–241
path size limits based on 190
Type 2 and Type 14 fonts handling of 343
store operator 54, 698
storing
elements, of composite objects (**put**) 53, 635
objects into an array (**astore**) 54, 535
stream(s)
byte, samples treated as 290
character
as executable object source 24
file as 43
of image samples, **DCTDecode** filter output 146
input, creating positionable (**ReusableStreamDecode**) *(table)* 86
reusable, **ReusableStreamDecode** filter 153
sample data represented as 110
string operator 699
as composite object constructor 58
creating strings with 53
string(s)
access, for files *(table)* 79
arrays compared with 39
ASCII base-85, ASCII encoding 31
binary object sequence encoding 166
as binary tokens 160
as composite object *(table)* 34
converting
objects to (**cvs**) 568
radix numbers to (**cvrs**) 567
to a name object (**cvn**) 566
to integers (**cvi**) 566
to real numbers (**cvr**) 567
copying 40
copy 53, 548
creating (**string**) 53, 699
as data sources and targets 126
encoded number
as binary tokens 171–172
homogeneous number array use 162, 171
operators that use 172
executable, execution handling 50
hexadecimal
ASCII encoding 30–31
reading (**readhexstring**) 73, 639
writing (**writehexstring**) 73, 720
literal text, ASCII encoding 29–30
mapping a procedure over (**forall**) 53, 597

string(s) *(continued)*
names and, interchangeable in an dictionary 41
objects 39–40
operators
(list) 511
overview 52–55
portability issues 40
reading (**readstring**) 73, 641
retrieving
elements of (**get**) 53, 598
the number of elements in (**length**) 53, 621
reusable streams compared with 153
as sample data source 291
scanning, for PostScript tokens (**token**) 702
searching
for initial substring (**anchorsearch**) 529
for substrings in (**search**) 655
storing elements of (**put**) 53, 635
subintervals of
creating objects that share (**getinterval**) 53
overwriting (**putinterval**) 53, 636
syntax 29–31
writing (**writestring**) 73, 721
stringwidth operator 699
accessing glyph width with 320
stroke adjustment
automatic 503–504
as graphics state parameter *(table)* 180
parameter
retrieving (**currentstrokeadjust**) 563
setting (**setstrokeadjust**) 683
stroke operator 177, 700
current line width use 178
operations 193
strokepath operator 700
strokes
See also drawing; line(s); painting
insideness testing (**instroke**) 613
painting lines with 193–194
StrokeWidth entry
as Type 0 CIDFont dictionary entry *(table)* 373
as Type 1 font dictionary entry *(table)* 326
as Type 2 CIDFont dictionary entry *(table)* 378
as Type 42 font dictionary entry *(table)* 347
structure
PostScript language *(chapter)* 23–174
program, PostScript page description 17–18
structured
output 172–174
programs
conventions, further information on, "PostScript Language Document Structuring Conventions" 813
structured *(continued)*
programs *(continued)*
save and **restore** intended for 62
sub operator 52, 701
subfile
applications for 151
filters, overview 85
SubFileDecode filter 86, 151–152
dictionary *(table)* 152
subintervals
of arrays
creating objects that share (**getinterval**) 53, 599
overwriting (**putinterval**) 53, 636
of packed arrays, creating objects that share (**getinterval**) 53, 599
of strings
creating objects that share (**getinterval**) 53, 599
overwriting (**putinterval**) 53, 636
subpaths 190
See also path(s)
closing (**closepath**) 191, 194
creating
moveto 627
rmoveto 650
SubrCount entry
as **Private** dictionary entry *(table)* 375
SubrMapOffset entry
as **Private** dictionary entry *(table)* 375
subsetting
fonts 352–356
glyphs, incrementally, general considerations 353–354
substitution
names, immediate evaluation 121–123
operator, **bind** use of 118–119
SubsVector entry
as font mapping algorithm 360–362
mapping control for Type 0 fonts with 360–362
as Type 0 font dictionary entry *(table)* 358
subtracting
numbers (**sub**) 701
subtractive
color models
CMYK 218
tints as, in **Separation** color spaces 242
supercells 500–501
Supplement entry
as **CIDSystemInfo** dictionary entry *(table)* 368
Symbol character set 778
(table) 792–793
Symbol encoding vector
(table) 774, 794

syntax
arrays 32
dictionaries 33
errors (**syntaxerror**) 701
file names 77
immediately evaluated name syntax 121–123
literals 29–30, 31
names 31
numbers 28
PostScript 25–34
procedures 32–33
of statements 81
strings 29–31
ASCII base-85 31
hexadecimal 30–31
hexadecimal, reading from an input file (**readhexstring**) 73
hexadecimal, writing to an output file (**writehexstring**) 73
syntaxerror error 701
binary object sequence 167–168
binary token types 159
hexadecimal strings 30
system
coordinate
operator overview 175
source image 293–296
transformations and 182–189
encoded names 168–169
as binary tokens 160
name
index 168
table 168
parameters
See system parameters
system parameters
BuildTime 751
ByteOrder 170, 751
CurDisplayList 751
CurFontCache 382, 751
CurFormCache 751
CurOutlineCache 751
CurPatternCache 751
CurScreenStorage 751
CurSourceList 751
CurStoredScreenCache 751
CurUPathCache 751
FactoryDefaults 751
FontResourceDir 752
GenericResourceDir 752
GenericResourcePathSep 752
LicenseID 752
MaxDisplayAndSourceList 752
MaxDisplayList 752
MaxFontCache 752
MaxFormCache 752
system parameters *(continued)*
MaxImageBuffer 752
MaxOutlineCache 752
MaxPatternCache 752
MaxScreenStorage 752
MaxSourceList 752
MaxStoredScreenCache 752
MaxUPathCache 752
PageCount 752
PrinterName 753
properties of 746–749
RealFormat 170, 753
retrieving (**currentsystemparams**) 563
Revision 753
setting (**setsystemparams**) 684
StartJobPassword 753
StartupMode 753
SystemParamsPassword 753
systemdict dictionary 42, 65, 702
as dictionary stack component 46
LanguageLevel 3 operators defined in *(list)* 726
null objects as components of 44
operator names as keys in 43
as standard global dictionary *(table)* 66
user path operators 199
SystemParamsPassword system parameter
value *(table)* 753

T

tab (tab) character
numeric values *(table)* 27
\t as ASCII encoding *(list)* 29
Table array
as **CIEBasedDEF** dictionary entry *(table)* 234
as **CIEBasedDEFG** dictionary entry *(table)* 235
tail recursion
interpreter facilitation of 50
targets
arbitrary, treating as output file, **NullEncode** filter 156
data 123–127
templates
forms use as 207
resource category, **Generic** use as 102–103
tensor-product patch meshes
as type 7 shading 261, 283–287
terminating
execution
quit 637
stop 697
files, with EOF 126–129
filters, with EOD 126–129
iteration execution (**exit**) 585
VM alterations 71

testing
See also insideness testing
access attributes
rcheck 638
wcheck 718
equality
See also comparing
names 40
user path values 203
executable attribute (**xcheck**) 721
insideness
ineofill 610
infill 610
instroke 613
inueofill 614
inufill 615
inustroke 616
of a point 196–197
operators 197
operators *(list)* 520
objects for global VM eligibility (**gcheck**) 598
text
explicitly positioned, encoded number string use 171
Latin
ISOLatin1Encoding 618
ISOLatin1Encoding resource 95
literal text strings, ASCII encoding 29–30
representation
of an object, writing to standard output (**=**) 87
of an object, writing to standard output (**==**) 87
showing, basics of 314–316
spacing
ashow 320, 534–535
awidthshow 320, 536
cshow 321, 551
kshow 321, 619
show 320
widthshow 320, 718
xshow 320, 722
xyshow 722
yshow 320, 723
string objects use for holding 39
textures
See patterns
thickness
stroked line, consistency (**setstrokeadjust**) 194
three-dimensional objects
radial shading patterns used for 267
threshold arrays 489–497
type 3 halftone dictionaries 490
type 6 halftone dictionaries 491–492
type 10 halftone dictionaries 492–495
type 16 halftone dictionaries 495–497
Thresholds entry
as type 3 halftone dictionary entry *(table)* 490
Thresholds file
as type 6 halftone dictionary entry *(table)* 491
as type 10 halftone dictionary entry *(table)* 495
as type 16 halftone dictionary entry *(table)* 496
TIFF (tag image file format) standard
Data Compression scheme #32773, **RunLengthEncode** filter similar to 142
LZW method use 133
predictor functions based on 139
tilde angle bracket (~>)
as EOD indication
ASCII85Decode 131
ASCII85Encode 131
tilde angle brackets (<~~>)
as ASCII base-85 strings delimiter 31
tiling
of device space, in a type 16 halftone dictionary *(figure)* 497
with different colors, uncolored tiling pattern use 257
patterns 248, 250–259
colored 254–257
uncolored 257–259
uncolored, restrictions on **PaintProc** 215
with gradient fill, creating 261
TilingType entry
as type 1 pattern dictionary entry *(table)* 252
time
errors (**timeout**) 115, 702
retrieving (**usertime**) 714
transmission, compression/decompression filters use to improve 84
timeout error 702
special handling 115
Times font family 775
tints
as subtractive colors, in **Separation** color spaces 242
token operator 55, 702
binary object sequence impact by 163
scanner invoked by 157
scanning, input files for PostScript tokens 73
tokens 26
ASCII, binary tokens compared with 158
binary 156, 158–162
ASCII tokens compared with 158
encoded number strings as 171–172
encoded system names in 168–169
interpretation *(table)* 158–159
object sequence 163
object sequence, binary tokens compared with 158, 169
double angle brackets as 33

tokens *(continued)*
scanning
input files for (**token**) 73
strings for (**token**) 702
type 157
binary object sequences 163
number representation specified in 169
topology
path 190
trademarks 9
transfer functions 478–480
as graphics state parameter *(table)* 180
retrieving
currentcolortransfer 555
currenttransfer 563
setting
setcolortransfer 479, 666
settransfer 479, 685
use, as rendering step 457
TransferFunction procedure
as type 1 halftone dictionary entry *(table)* 488
as type 3 halftone dictionary entry *(table)* 490
as type 6 halftone dictionary entry *(table)* 491
as type 10 halftone dictionary entry *(table)* 495
as type 16 halftone dictionary entry *(table)* 496
transform operator 703
transformation
component, in **CIEBasedA** color space *(figure)* 229
coordinate
effects *(figure)* 188
rotate 651
scale 654
systems and 182–189
transformation operators, modifying user space with 184
translate 704
of data, filters use for 82
of distance vector (**dtransform**) 574–575
identity (**matrix**) 626
image space, matrix source and use 294–296
inverse of 189
itransform 619
matrix 184–186
applying (**transform**) 703
CTM
See CTM (current transformation matrix)
inverting (**idtransform**) 606
(table) 208
of sample color values, **DCTEncode** dictionary entry *(table)* 149
two-dimensional, mathematical description 187
TransformPQR array
as type 1 CIE-based CRD entry *(table)* 465, 465–467
translate operator 184, 704
CTM manipulation by 189
user space modification by 185
translation
matrix notation 187
object (**translate**) 704
from other print formats 19–20
of resource names into file names, by **ResourceFileName** key 103–104
translate 704
user path caching value (**translate**) 203
translucency
overprint control of 247–248
transparency
of data transmission
binary encoding requirement 157
NullEncode *(table)* 86
subfile filters use for, overview 85
SubFileDecode *(table)* 86
TrapColorScaling entry
ColorantZoneDetails dictionary *(table)* 454
Trapping dictionary, entries *(table)* 448
TrapParams resource category 97
(table) 91
trapping
application-level compared with in-RIP 440
detail dictionary 441–445
dictionary
retrieving (**currenttrapparams**) 564
setting (**settrapparams**) 685
image 453–454
in-RIP 439–454
InkParams resource category 97
parameters 446–454
rules
black 451–452
normal 449–451
sliding traps 452–453
zones 445–446
setting (**settrapzone**) 686
Trapping page device parameter
color support *(table)* 422
dictionary *(table)* 447–449
Trapping procedure set
LanguageLevel 3 operators defined in *(table)* 726
as standard procedure set in LanguageLevel 3 *(table)* 96
TrappingDetails page device parameter
color support *(table)* 422
dictionary 441–445
entries *(table)* 442
TrappingDetailsType array
as **OutputDevice** dictionary entry *(table)* 456

TrappingOrder array
as **TrappingDetails** dictionary entry *(table)* 442
TrappingType resource category 98
(table) 92
TrapSetName entry
Trapping dictionary, entries *(table)* 447
TrapWidth entry
Trapping dictionary, entries *(table)* 448
trays 398
TraySwitch page device parameter 403
triangle meshes
free-form Gouraud-shaded triangle meshes, as type 4 shading 261, 270–274
lattice-form Gouraud-shaded triangle meshes, as type 5 shading 261, 274–276
trigonometric
functions
atan 535–536
cos 550
sin 692
true 704
bind not usable with 119
as boolean value 38
(overview) 55
TrueType
See Type 42 (TrueType) fonts
truncate operator 52, 705
truncating
numbers (**truncate**) 705
Tumble page device parameter 416
turning
user space axes around current origin (**translate**) 185, 704
Type entry
TrappingDetails dictionary *(table)* 442
type operator 56, 705
InstanceType key use by, category implementation dictionary *(table)* 101
type(s)
conversion, between integer and real objects 38
errors (**typecheck**) 706
font *(table)* 322
object 34–45, 34
retrieving as a name object (**type**) 705
(table) 34
operators
individual
*See **type***
(list) 512
overview 56
procedure object, as either array or packed array 33
of shading patterns 265–290
type(s) *(continued)*
token
binary object sequences 163
number representation specified in 169
typecheck error 706
image
dictionary 298
masking interleaving 303
user path
malformation 204
rule violation 201
structuring 199

U

uappend operator 204, 706
clipping user paths with 205
encoded number string use 172
ucache operator 198, 707
operation code for encoded user paths *(table)* 201
user path
caching with 202–203
structuring 199
ucachestatus operator 707
ueofill operator 204, 708
ufill operator 177, 204, 708
encoded user path handling 200
garbage collection benefits for management of 63
treating a user object as a user path *(example)* 68
user path construction and painting 197
UIDBase entry
as **CIDFont** dictionary entry *(table)* 371
UIDOffset entry
as CMap dictionary entry *(table)* 384
Uncompressed entry
as **CCITTFaxEncode**/**CCITTFaxDecode** dictionary entry *(table)* 144
undef operator 54, 708
undefined
characters, handling
undefined error 709
image dictionary 298
immediate evaluation of names 122
immediately evaluated names 167
LanguageLevel 2, **Separation** or **DeviceN** use with **Indexed** color space 239
name search failure cause 51
operators that cause, due to graphics state restrictions 215
resource file loading 105
system name index 168
tiling pattern colors *(table)* 252

undefinedfilename error 709
invalid EOL cause 82
undefinedresource error 709
undefinedresult error 709
function-based shading pattern 266
mathematical operation cause 52
noninvertable CTM 189
undefinefont operator 93, 710
usage guidelines 809
undefineresource operator 88, 710
deleting resources with 89
IdiomSet use 121
implementation dictionary 100
implicit resource handling 98–99
UndefineResource key use by, category implementation dictionary *(table)* 101
usage guidelines 809
UndefineResource procedure
as category implementation dictionary entry *(table)* 101
undefineuserobject operator 67, 711
undefining
fonts (**undefinefont**) 710
resources (**undefineresource**) 710
user objects (**undefineuserobject**) 67, 711
undercolor removal 476–477
See also black generation; color; overprinting
function
retrieving (**currentundercolorremoval**) 564
setting (**setundercolorremoval**) 476, 687
as graphics state parameter *(table)* 180
UnderlinePosition entry
as entry in **FontInfo** dictionary *(table)* 327
UnderlineThickness entry
as entry in **FontInfo** dictionary *(table)* 327
Unicode
CMap use for encoding 366
unique ID 335
extended, as LanguageLevel 2 feature (**XUID**) 336
font modification implications 348
generation of 335–337
numbers 336
extended 337
UniqueID entry
as base font dictionary entry *(table)* 326
supported by all language levels 336
UnitLength entry
as LZW dictionary entry *(table)* 134
LZWEncode use 136–137
unlimited access
as object attribute 37
unmatchedmark error 711
unregistered error 712
upath operator 204, 712
UseCIEColor page device parameter 422
remapping device colors to CIE-based color space specification 237
usecmap operator 384, 712
usefont operator 384, 713
user
defined, dictionaries 65–67
interactive executive prompting of (**prompt**) 635
objects 67–68
defining (**defineuserobject**) 67, 571
defining (**defineuserobject**) *(example)* 68
executing (**execuserobject**) 67, 584
executing (**execuserobject**) *(example)* 68
manipulating *(example)* 68
undefining (**undefineuserobject**) 67, 711
UserObjects 713
parameters
See user parameters
paths
See user paths
space 182
coordinates in 184
default 183
device space and 182–184
image space relationship to 294
modifying 185–187
origin, physical page relationship 183
transformation to device space 184
user parameters
AccurateScreens 482, 487, 749, 757
HalftoneMode 482, 749
IdiomRecognition 119–121, 749
JobName 750
MaxDictStack 750
MaxExecStack 750
MaxFontItem 750
MaxFormItem 750
MaxLocalVM 750
MaxOpStack 750
MaxPatternItem 750
MaxScreenItem 750
MaxSuperScreen 482, 750
MaxUPathItem 750
MinFontCompress 750
properties of 746–749
retrieving (**currentuserparams**) 564
setting (**setuserparams**) 687
VMReclaim 750
VMThreshold 750
user paths 197–206
appending (**uappend**) 706

user paths *(continued)*
cache 202–204
output placed in 198
setting (**setucacheparams**) 686
setting (**ucache**) 707
status retrieving (**ucachestatus**) 707
constructing 198–200
ustroke 197
creating
(example) 68
upath 712
encapsulation
gsave and **grestore** use for 205
predictability advantage 197
encoded 200–202
(example) 202
encoded number string use 171
filling
ufill 197
ufill *(example)* 68
operators 204–204
(list) 198
painting
ueofill 708
ufill 708
ustroke 714
ustrokepath 715
structuring of 199
userdict dictionary 42, 65, 713
as dictionary stack component 46
as standard local dictionary *(table)* 65
usage guidelines 810
UserObjects array 713
defineuserobject management of *(example)* 68
user objects held in 67
usertime operator 714
ustroke operator 177, 197, 204, 714
ustrokepath operator 204, 715

V

values
associating keys with, in dictionaries on the dictionary stack (**def**) 54, 568
associating with names 41
color 210
in **CIEBasedABC** color spaces 222
operators 214–216
semantics of 211–248
of composite objects, not copied by stack operators 51
composite objects
sharing between copies 35
VM as pool for 57
values *(continued)*
dictionary 41
dictionary object, access attributes apply to 37
locating, on the dictionary stack (**where**) 54, 718
of objects, pushing/popping effects 46
replacing, in a dictionary on the dictionary stack (**store**) 54, 698
retrieving, from the dictionary stack (**load**) 54, 622
sharing, among composite objects 58
simple objects 35
user paths, as equality testing basis 203
variable-pitch fonts 320
variables
dictionaries used to manage 41
vector(s)
distance, transforming (**dtransform**) 574–575
encoding
changing the 349–350
character code and glyph mapping 328–330
retrieving (**findencoding**) 591
StandardEncoding 693
(table) 773–774, 784–794
version
PostScript, retrieving (**version**) 716
version entry
as entry in **FontInfo** dictionary *(table)* 327
version operator 716
vertex
specification, in type 4 shading 271–274
vertical
sampling, **DCTEncode** dictionary entry *(table)* 148
vertical writing
font metrics for 332
VerticesPerRow entry
as type 5 shading dictionary entry *(table)* 275
virtual memory
See VM (virtual memory)
VM (virtual memory) 57–59
See also garbage collection
allocation
policies, resource loading issues 105
retrieving (**currentglobal**) 557
setglobal control of 60
setting (**setvmthreshold**) 688
alterations, termination of 71
composite object
creation impact on 58
values contained in 35
consumption of, by composite object creation operators 53
creating objects in *(example)* 60
encapsulation of 68–72
encoded number string use in 171

VM (virtual memory) *(continued)*
errordict dictionary located in 115
errors (**VMerror**) 62, 716
font introduction into 88
font resource use of 93
forms 206
global 59–61
category implementation dictionary located in 101
guidelines for use 66
setting mode for (**setglobal**) 670
testing objects for eligibility (**gcheck**) 598
IdiomSet use of 121
initial, altering 70–72
local 59–61
reference by global VM object prohibited 60
reference by global VM object prohibited, **systemdict** exceptions 66
save and **restore** operations 61–62
snapshot of state, as **save** operation 61
name values occupy space in 40
operators *(list)* 514
resources use of 89
save and **restore** impact on 72
state
restoring (**restore**) 648
saving (**save**) 653
status, retrieving (**vmstatus**) 717
storage allocation in, by composite object constructors 58
Type 4 CIDFont advantages 380
unique ID benefits 335
use 742–743
user path caching 203
VMerror error 716
memory exhaustion 62
vmreclaim operator 716
VMReclaim user parameter
value *(table)* 750
vmstatus operator 717
VMThreshold user parameter
value *(table)* 750
VSamples entry
as **DCTEncode** dictionary entry *(table)* 148

W

W3C (WWW Consortium)
PNG standard 139
wcheck operator 56, 718
weight
media (**MediaWeight**) 401
Weight entry
as entry in **FontInfo** dictionary *(table)* 327
WeightVector array
as Type 1 font dictionary entry *(table)* 327
where operator 43, 54, 718
white-space
characters 26
numeric values *(table)* 27
WhitePoint array
calculation of 226
as **CIEBasedA** dictionary entry *(table)* 230
as **CIEBasedABC** dictionary entry *(table)* 224
as type 1 CIE-based CRD entry *(table)* 464
width
See also length; size
glyph 320, 331
adjusting (**widthshow**) 718
retrieving (**stringwidth**) 699
setting (**setcharwidth**) 661
line
retrieving (**currentlinewidth**) 559
setting (**setlinewidth**) 193, 674
Width entry
as image data dictionary entry *(table)* 305
as image mask dictionary entry *(table)* 306
as type 1 image dictionary entry *(table)* 298
as type 3 halftone dictionary entry *(table)* 490
as type 4 image dictionary entry *(table)* 307
as type 6 halftone dictionary entry *(table)* 491
as type 16 halftone dictionary entry *(table)* 496
Width2 entry
as type 16 halftone dictionary entry *(table)* 496
widthshow operator 320, 718
wildcards
filenameforall use 78
resourceforall use 646
winding number
nonzero, rule 195
WMode entry
as **CIDFont** dictionary entry *(table)* 371
as CMap dictionary entry *(table)* 384
as font dictionary entry *(table)* 325
as Type 0 font dictionary entry 362
write access attribute
testing (**wcheck**) 718
write operator 720
EOL transparent to 75
writing, characters to an output file 73
writehexstring operator 720
writing, hexadecimal strings to an output file 73
writeobject operator 721
binary object sequence writing 173

writestring operator 721
EOL transparent to 75
writing, strings to an output file 73
writing
binary object sequences
to standard output (**printobject**) 634
writeobject 721
characters
to an output file (**write**) 73
to standard output (**print**) 633
write 720
files
access convention 79
in arbitrary order (**setfileposition**) 80
hexadecimal strings, to an output file (**writehexstring**) 73, 720
operand stack contents
to standard output (**stack**) 692
using == (**pstack**) 87
using = (**stack**) 87
strings
to an output file (**writestring**) 73
writestring 721
text representation of an object
to standard output (==) 87
to standard output (=) 87
vertical, font metrics for 332

X

xcheck operator 56, 721
xor (exclusive or) operator 55, 722
xshow operator 320, 321, 722
encoded number string use 172
Xsquare entry
as type 10 halftone dictionary entry *(table)* 495
XStep entry
as type 1 pattern dictionary entry *(table)* 251
XUID array
as base font dictionary entry *(table)* 326
as **CIDFont** dictionary entry *(table)* 371
as CMap dictionary entry *(table)* 383
components of 337
as form dictionary entry *(table)* 208
as LanguageLevel 2 font feature 336
as type 1 pattern dictionary entry *(table)* 251
as type 2 pattern dictionary entry *(table)* 260
XUID entry
as form dictionary entry *(table)* 208
xyshow operator 321, 722

Y

yshow operator 320, 321, 723
encoded number string use 172
Ysquare entry
as type 10 halftone dictionary entry *(table)* 495
YStep entry
as type 1 pattern dictionary entry *(table)* 251

Z

zlib/deflate compressed format
Flate encoding of 137–142
FlateDecode filter use, overview *(overview table)* 86
FlateEncode filter use, overview *(overview table)* 85
zone theory
of color vision, **CIEBasedABC** as model of 221
zones
-specific colorant details 454
trapping 445–446
setting (**settrapzone**) 68

Colophon

THIS BOOK WAS PRODUCED using Adobe FrameMaker®, Adobe Illustrator®, Adobe Photoshop®, and other application software packages that support the PostScript language and Type 1 fonts. The type used is from the Minion® and Myriad® families. Heads are set in Myriad MM 565 Semibold, 600 Normal, and the body text is set in 10.5-on-13-point Minion.

Authors—Ed Taft, Steve Chernicoff, Caroline Rose

Key Contributors—Steven Kelley Amerige, Rob Babcock, Doug Brotz, Jo Davies, Richard Dermer, Matt Foley, Linda Gass, Ron Gentile, Peter Hibbard, Jim King, Ken Lent, Deborah MacKay, Jim Meehan, Carl Orthlieb, Paul Rovner, Mike Schuster, Scott Seltz, Andy Shore, John Warnock

Reviewers—Ken Anderson, Rob Babcock, Ned Batchelder, Perry Caro, Holly Cochran, L. Peter Deutsch, David Gelphman, Deborah MacKay, Henry McGilton, John Nash, Jim Sandman, Norin Saxe, Lydia Stang, and numerous others at Adobe Systems and elsewhere.

Editing and Book Production—Steve Chernicoff, Caroline Rose

Index—Rosemary Simpson

Illustrations—Carl Yoshihara, Wendy Bell, Dayna Porterfield, Lisa Ferdinandsen, Carol Keller

Book Design—Sharon Anderson

Publication Management—Robin Sterling, Richard Compeau, Sholom Surges, Joan Delfino

Project Management—Caroline Rose, Ed Taft