PDF Reference

fourth edition

Adobe® Portable Document Format

Version 1.5

Adobe Systems Incorporated

Contents

Figures

Tables

Preface

THE ORIGINS OF THE Portable Document Format and the Adobe® Acrobat® product family date to early 1990. At that time, the PostScript® page description language was rapidly becoming the worldwide standard for the production of the printed page. PDF builds on the PostScript page description language by layering a document structure and interactive navigation features on PostScript's underlying imaging model, providing a convenient, efficient mechanism enabling documents to be reliably viewed and printed anywhere.

The PDF specification was first published at the same time the first Acrobat products were introduced in 1993. Since then, updated versions of the specification have been and continue to be available from Adobe via the World Wide Web. This book is the third professionally published edition of the specification. Like its predecessor, it is completely self-contained, including the precise documentation of the underlying imaging model from PostScript along with the PDF-specific features that are combined in version 1.5 of the PDF standard.

Over the past ten years, aided by the explosive growth of the Internet, PDF has become the *de facto* standard for the electronic exchange of documents. Well over 500 million copies of the free Adobe Reader® application have been distributed around the world, facilitating efficient sharing of digital content. In addition, PDF is now the industry standard for the intermediate representation of printed material in electronic prepress systems for conventional printing applications. As major corporations, government agencies, and educational institutions streamline their operations by replacing paper-based workflow with electronic exchange of information, the impact and opportunity for the application of PDF will continue to grow at a rapid pace.

PDF is the file format that underlies Adobe ePaper® Solutions, a family of products supporting Adobe's vision for Network Publishing—the process of creating, managing, and accessing digital content on diverse platforms and devices. ePaper fulfills a set of requirements related to business process needs for the global desktop user, including:

- Preservation of document fidelity across the enterprise, independently of the device, platform, and software
- Merging of content from diverse sources—Web sites, word processing and spreadsheet programs, scanned documents, photos, and graphics—into one self-contained document while maintaining the integrity of all original source documents
- Real-time collaborative editing of documents from multiple locations or platforms
- Digital signatures to certify authenticity
- Security and permissions to allow the creator to retain control of the document and associated rights
- Accessibility of content to those with disabilities
- Extraction and reuse of content using other file formats and applications

A significant number of third-party developers and systems integrators offer customized enhancements and extensions to Adobe's core family of products. Adobe publishes the PDF specification in order to encourage the development of such third-party applications.

The emergence of PDF as a standard for electronic information exchange is the result of concerted effort by many individuals in both the private and public sectors. Without the dedication of Adobe employees, our industry partners, and our customers, the widespread acceptance of PDF could not have been achieved. We thank all of you for your continuing support and creative contributions to the success of PDF.

Chuck Geschke and John Warnock
May 2001

CHAPTER 1

Introduction

THE ADOBE PORTABLE DOCUMENT FORMAT (PDF) is the native file format of the Adobe® Acrobat® family of products. The goal of these products is to enable users to exchange and view electronic documents easily and reliably, independently of the environment in which they were created. PDF relies on the same imaging model as the PostScript® page description language to describe text and graphics in a device-independent and resolution-independent manner. To improve performance for interactive viewing, PDF defines a more structured format than that used by most PostScript language programs. PDF also includes objects, such as annotations and hypertext links, that are not part of the page itself but are useful for interactive viewing and document interchange.

1.1 About This Book

This book provides a description of the PDF file format and is intended primarily for application developers wishing to develop *PDF producer* applications that create PDF files directly. It also contains enough information to allow developers to write *PDF consumer* applications that read existing PDF files and interpret or modify their contents.

Although the PDF specification is independent of any particular software implementation, some PDF features are best explained by describing the way they are processed by a typical application program. In such cases, this book uses the Adobe Acrobat family of PDF viewer applications as its model. (The prototypical viewer is the fully capable Acrobat product, not the limited Adobe Reader® product.) Similarly, Appendix C discusses some implementation limits in the Acrobat viewer applications, even though these limits are not part of the file format itself. To provide guidance to implementors of PDF producer and consumer applications, compatibility and implementation notes in Appendix H describe the behavior of Acrobat viewer applications when they encounter newer features they do not understand, as well as areas in which the Acrobat products diverge from the specification presented in this book.

This fourth edition of the *PDF Reference* describes version 1.5 of PDF. (See implementation note 1 in Appendix H.) Throughout the book, information specific to particular versions of PDF is marked as such—for example, with indicators like *(PDF 1.3)* or *(PDF 1.4)*. Features so marked may be new in the indicated version or may have been substantially redefined in that version. Features designated *(PDF 1.0)* have generally been superseded in later versions; unless otherwise stated, features identified as specific to other versions are understood to be available in later versions as well. (PDF viewer applications designed for a specific PDF version generally ignore newer features they do not recognize; implementation notes in Appendix H point out exceptions.)

The rest of the book is organized as follows:

- Chapter 2, "Overview," briefly introduces the overall architecture of PDF and the design considerations behind it, compares it with the PostScript language, and describes the underlying imaging model that they share.
- Chapter 3, "Syntax," presents the syntax of PDF at the object, file, and document level. It sets the stage for subsequent chapters, which describe how that information is interpreted as page descriptions, interactive navigational aids, and application-level logical structure.
- Chapter 4, "Graphics," describes the graphics operators used to describe the appearance of pages in a PDF document.
- Chapter 5, "Text," discusses PDF's special facilities for presenting text in the form of character shapes, or *glyphs*, defined by fonts.
- Chapter 6, "Rendering," considers how device-independent content descriptions are matched to the characteristics of a particular output device.
- Chapter 7, "Transparency," discusses the operation of the *transparent imaging model*, introduced in PDF 1.4, in which objects can be painted with varying degrees of opacity, allowing the previous contents of the page to show through.
- Chapter 8, "Interactive Features," describes those features of PDF that allow a user to interact with a document on the screen, using the mouse and keyboard.
- Chapter 9, "Multimedia Features," describes those features of PDF that support embedding and playing multimedia content.

- Chapter 10, "Document Interchange," shows how PDF documents can incorporate higher-level information that is useful for the interchange of documents among applications.

  Note: *Chapter 9 is new in this edition; it contains information on the new multimedia features (see Section 1.2, "Introduction to PDF 1.5 Features" below), as well as some material that was previously in Chapter 8. The material that was in Chapter 9 of the previous edition is now in Chapter 10.*

The appendices contain useful tables and other auxiliary information.

- Appendix A, "Operator Summary," lists all the operators used in describing the visual content of a PDF document.
- Appendix B, "Operators in Type 4 Functions," summarizes the PostScript operators that can be used in PostScript calculator functions, which contain code written in a small subset of the PostScript language.
- Appendix C, "Implementation Limits," describes typical size and quantity limits imposed by the Acrobat viewer applications.
- Appendix D, "Character Sets and Encodings," lists the character sets and encodings that are assumed to be predefined in any PDF viewer application.
- Appendix E, "PDF Name Registry," discusses a registry, maintained for developers by Adobe Systems, that contains private names and formats used by PDF producers or Acrobat plug-in extensions.
- Appendix F, "Linearized PDF," describes a special form of PDF file organization designed to work efficiently in network environments.
- Appendix G, "Example PDF Files," presents several examples showing the structure of actual PDF files, ranging from one containing a minimal one-page document to one showing how the structure of a PDF file evolves over the course of several revisions.
- Appendix H, "Compatibility and Implementation Notes," provides details on the behavior of Acrobat viewer applications and describes how viewer applications should handle PDF files containing features that they do not recognize.
- Appendix I, "Computation of Object Digests," describes in detail an algorithm for calculating an object digest (discussed in Section 8.7, "Digital Signatures").

A color plate section provides illustrations of some of PDF's color-related features. References in the text of the form "see Plate 1" refer to the contents of this section.

The book concludes with a Bibliography and an Index.

1.2 Introduction to PDF 1.5 Features

Several features have been introduced or modified in PDF 1.5. The following is a list of the most significant additions, along with the primary sections where the material is discussed:

- The ability to display images using JPEG2000 compression (Section 3.3.8, "JPXDecode Filter"), and to allow 16-bit images (Section 4.8.4, "Image Dictionaries")
- Additional options for the encryption of documents (Section 3.5, "Encryption"). Major new features include crypt filters (Section 3.3.9, "Crypt Filter" and "Section 3.5.4, "Crypt Filters") and syntax for public-key security handlers (Section 3.5.3, "Public-Key Security Handlers", which contains information introduced in PDF 1.3 but not documented in the *PDF Reference* until this edition)
- An extension to the use of streams to allow greater compression of PDF files (Section 3.4.6, "Object Streams" and Section 3.4.7, "Cross-Reference Streams")
- The ability to selectively view or hide content in a PDF document (Section 4.10, "Optional Content" and "Set-OCG-State Actions" on page 608)
- New predefined CMaps and character collections ("Predefined CMaps" on page 404)
- Enhancements to interactive presentations (Section 8.3.3, "Presentations"), including navigation between pages ("Sub-page Navigation" on page 555) and a new action type ("Transition Actions" on page 611)
- Additional annotation types (Section 8.4.5, "Annotation Types") and other enhancements to annotations (Section 8.4, "Annotations")
- Miscellaneous enhancements to interactive forms (Section 8.6, "Interactive Forms"), including support for forms based on Adobe's XML Forms Architecture (Section 8.6.7, "XFA Forms") and the ability to use styled text in form fields and markup annotations ("Rich Text Strings" on page 620)

- Enhancements related to digital signatures and signature fields, including the ability to compute object signatures ("Signature Fields" on page 636, Section 8.7, "Digital Signatures," and Appendix I, "Computation of Object Digests")
- Greatly enhanced support for embedding and playback of multimedia (Section 9.1, "Multimedia", "Screen Annotations" on page 588 and "Rendition Actions" on page 609)
- The ability to allow the display of a PDF file as a slideshow (Section 9.4, "Alternate Presentations"). (This feature is considered part of PDF 1.4, although it was not previously documented.)
- Enhancements to Tagged PDF (Section 10.7, "Tagged PDF") and accessibility features (Section 10.8, "Accessibility Support"). Related updated information is found in Section 5.7, "Font Descriptors" and Section 5.9, "Extraction of Text Content."

1.3 Related Publications

PDF and the PostScript page description 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, PostScript includes a general-purpose programming language framework not present in PDF. The *PostScript Language Reference* is the comprehensive reference for the PostScript language and its imaging model.

PDF and PostScript support several standard formats for font programs, including Adobe Type 1, CFF (Compact Font Format), TrueType, and CID-keyed fonts. The PDF manifestations of these 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 #5177, *The Type 2 Charstring Format*
- Adobe Technical Note #5014, *Adobe CMap and CID Font Files Specification*

See the Bibliography for additional publications related to PDF and the contents of this book.

1.4 Intellectual Property

The general idea of using an interchange format for electronic documents is in the public domain. Anyone is free to devise a set of unique data structures and operators that define an interchange format for electronic documents. However, Adobe Systems Incorporated owns the copyright for the particular data structures and operators and the written specification constituting the interchange format called the Portable Document Format. Thus, these elements of the Portable Document Format may not be copied without Adobe's permission.

Adobe will enforce its copyright. Adobe's intention is to maintain the integrity of the Portable Document Format standard. This enables the public to distinguish between the Portable Document Format and other interchange formats for electronic documents. However, Adobe desires to promote the use of the Portable Document Format for information interchange among diverse products and applications. Accordingly, Adobe gives anyone copyright permission, subject to the conditions stated below, to:

- Prepare files whose content conforms to the Portable Document Format
- Write drivers and applications that produce output represented in the Portable Document Format
- Write software that accepts input in the form of the Portable Document Format and displays, prints, or otherwise interprets the contents
- Copy Adobe's copyrighted list of data structures and operators, as well as the example code and PostScript language function definitions in the written specification, to the extent necessary to use the Portable Document Format for the purposes above

The conditions of such copyright permission are:

- Authors of software that accepts input in the form of the Portable Document Format must make reasonable efforts to ensure that the software they create respects the access permissions and permissions controls listed in Table 3.20 of this specification, to the extent that they are used in any particular document.

- Anyone who uses the copyrighted list of data structures and operators, as stated above, must include an appropriate copyright notice.
- Accessing the document in ways not permitted by the document's access permissions is a violation of the document author's copyright.

This limited right to use the copyrighted list of data structures and operators does not include the right to copy this book, other copyrighted material from Adobe, or the software in any of Adobe's products that use the Portable Document Format, in whole or in part, nor does it include the right to use any Adobe patents, except as may be permitted by an official Adobe Patent Clarification Notice (see the Bibliography).

Acrobat, Acrobat Capture, Adobe Reader, ePaper, the "Get Adobe Reader" Web logo, the "Adobe PDF" Web logo, and all other trademarks, service marks, and logos used by Adobe (the "Marks") are the registered trademarks or trademarks of Adobe Systems Incorporated in the United States and other countries. Nothing in this book is intended to grant you any right or license to use the Marks for any purpose.

CHAPTER 2

Overview

THE ADOBE *PORTABLE DOCUMENT FORMAT (PDF)* is a file format for representing documents in a manner independent of the application software, hardware, and operating system used to create them and of the output device on which they are to be displayed or printed. A *PDF document* consists of a collection of *objects* that together describe the appearance of one or more *pages*, possibly accompanied by additional interactive elements and higher-level application data. A *PDF file* contains the objects making up a PDF document along with associated structural information, all represented as a single self-contained sequence of bytes.

A document's pages (and other visual elements) may contain any combination of text, graphics, and images. A page's appearance is described by a PDF *content stream*, which contains a sequence of *graphics objects* to be painted on the page. This appearance is fully specified; all layout and formatting decisions have already been made by the application generating the content stream.

In addition to describing the static appearance of pages, a PDF document may contain interactive elements that are possible only in an electronic representation. PDF supports *annotations* of many kinds for such things as text notes, hypertext links, markup, file attachments, sounds, and movies. A document can define its own user interface; keyboard and mouse input can trigger *actions* that are specified by PDF objects. The document can contain *interactive form* fields to be filled in by the user, and can export the values of these fields to or import them from other applications.

Finally, a PDF document can contain higher-level information that is useful for interchange of content among applications. In addition to specifying appearance, a document's content can include identification and logical structure information that allows it to be searched, edited, or extracted for reuse elsewhere. PDF is particularly well suited for representing a document as it moves through successive stages of a prepress production workflow.

2.1 Imaging Model

At the heart of PDF is its ability to describe the appearance of sophisticated graphics and typography. This is achieved through the use of the *Adobe imaging model*, the same high-level, device-independent representation used in the PostScript page description language.

Although application programs could theoretically describe any page as a full-resolution pixel array, the resulting file would be bulky, device-dependent, and impractical for high-resolution devices. A high-level *imaging model* enables applications to describe the appearance of pages containing text, graphical shapes, and sampled images in terms of abstract graphical elements rather than directly in terms of device pixels. Such a description is economical and device-independent, and can be used to produce high-quality output on a broad range of printers, displays, and other output devices.

2.1.1 Page Description Languages

Among its other roles, PDF serves as a *page description language*: a language for describing the graphical appearance of pages with respect to an imaging model. An application program produces 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 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 the compact, device-independent transmission and storage of printable or displayable documents.

2.1.2 Adobe Imaging Model

The Adobe imaging model is a simple and unified view of two-dimensional graphics borrowed from the graphic arts. In this model, "paint" is placed on a page in selected areas.

- The painted figures may be in the form of character shapes *(glyphs)*, geometric shapes, lines, or sampled images such as digital representations of photographs.
- The paint may be in color or in black, white, or any shade of gray; it may also take the form of a repeating *pattern (PDF 1.2)* or a smooth transition between colors *(PDF 1.3)*.
- Any of these elements may be *clipped* to appear within other shapes as they are placed onto the page.

A page's content stream contains *operands* and *operators* describing a sequence of graphics objects. A PDF viewer application maintains an implicit *current page* that accumulates the marks made by the painting operators. Initially, the current page is completely blank. For each graphics object encountered in the content stream, the viewer places marks on the current page, which replace or combine with any previous marks they may overlay. Once the page has been completely composed, the accumulated marks are rendered on the output medium and the current page is cleared to blank again.

Versions of PDF up to and including PDF 1.3 use an *opaque imaging model* in which each new graphics object painted onto a page completely obscures the previous contents of the page at those locations (subject to the effects of certain optional parameters that may modify this behavior; see Section 4.5.6, "Overprint Control"). No matter what color an object has—white, black, gray, or color—it is placed on the page as if it were applied with opaque paint. PDF 1.4 introduces a new *transparent imaging model* in which objects painted on the page are not required to be fully opaque. Instead, newly painted objects are *composited* with the previously existing contents of the page, producing results that combine the colors of the object and its backdrop according to their respective opacity characteristics. The transparent imaging model is described in Chapter 7.

The principal graphics objects (among others) are as follows:

- A *path object* 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 *path construction operators*, each of which appends one or more new elements. The path object is ended by a *path-painting operator*, which paints the path on the page in some way. The principal path-painting operators are **S** (stroke), which paints a line along the path, and **f** (fill), which paints the interior of the path.

- A *text object* consists of one or more glyph shapes representing characters of text. The glyph shapes for the characters are described in a separate data structure called a *font*. Like path objects, text objects can be stroked or filled.
- An *image object* is a rectangular array of *sample values*, each representing a color at a particular position within the rectangle. Such objects are typically used to represent photographs.

The painting operators require various parameters, some explicit and others implicit. Implicit parameters include the current color, current line width, current font (typeface and size), and many others. Together, these implicit parameters make up the *graphics state*. There are operators for setting the value of each implicit parameter in the graphics state; painting operators use the values currently in effect at the time they are invoked.

One additional implicit parameter in the graphics state modifies the results of painting graphics objects. The *current clipping path* outlines the area of the current page within which paint can be placed. 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. Initially, the current clipping path encompasses the entire imageable area of the page. It can temporarily be reduced to the shape defined by a path or text object, or to the intersection of multiple such shapes. Marks placed by subsequent painting operators will then be confined within that boundary.

2.1.3 Raster Output Devices

Much of the power of the Adobe imaging model derives from its ability to deal with the general class of *raster output devices.* These encompass such technologies 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, or *raster*, of dots called *pixels* (picture elements) that can be addressed individually. On a typical *bilevel* output device, each pixel can be made either black or white. On some devices, pixels can be set to intermediate shades of gray or to some color. The ability to set the colors of individual pixels makes it possible to generate printed or displayed output that can include text, arbitrary graphical shapes, and reproductions of sampled images.

The *resolution* of a raster output device measures 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 achieve medium-level resolutions 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.

2.1.4 Scan Conversion

An abstract graphical element (such as a line, a circle, a character glyph, 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 to those pixels to achieve the most faithful rendition possible at the available device resolution.

The pixels on a page can be represented by a two-dimensional array of pixel values in computer memory. For an output device whose pixels can only be black or white, a single bit suffices to represent each pixel. For a device that can reproduce gray levels 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 representa-*

tions, 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 *render* this representation (make it visible) on a printed page or display screen.

Scan-converting a graphical shape, such as a rectangle or circle, entails determining which device pixels lie inside the shape and setting their values appropriately (for example, to black). Because the edges of a shape do not always fall precisely on the boundaries between pixels, some policy is required for deciding how to set the pixels along the edges. Scan-converting a glyph representing a text character is conceptually the same as scan-converting an arbitrary graphical shape; however, character glyphs 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 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 location on the page. When viewed from a sufficient distance, the individual dots become imperceptible and the perceived 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.

2.2 Other General Properties

This section describes other notable general properties of PDF, aside from its imaging model.

2.2.1 Portability

PDF files are represented as sequences of 8-bit binary bytes. A PDF file is designed to be portable across all platforms and operating systems. The binary representation is intended to be generated, transported, and consumed directly, without translation between native character sets, end-of-line representations, or other conventions used on various platforms.

Any PDF file can also be represented in a form that uses only 7-bit ASCII (American Standard Code for Information Interchange) character codes. This is useful for the purpose of exposition, as in this book. However, this representation is not recommended for actual use, since it is less efficient than the normal binary representation. Regardless of which representation is used, PDF files must be transported and stored as *binary* files, not as *text* files; inadvertent changes, such as conversion between text end-of-line conventions, will damage the file and may render it unusable.

2.2.2 Compression

To reduce file size, PDF supports a number of industry-standard compression filters:

- JPEG and (in PDF 1.5) JPEG2000 compression of color and grayscale images
- CCITT (Group 3 or Group 4), run-length, and (in PDF 1.4) JBIG2 compression of monochrome images
- LZW (Lempel-Ziv-Welch) and (beginning with PDF 1.2) Flate compression of text, graphics, and images

Using JPEG compression, color and grayscale images can be compressed by a factor of 10 or more. Effective compression of monochrome images depends on the compression filter used and the properties of the image, but reductions of 2:1 to 8:1 are common (or 20:1 to 50:1 for JBIG2 compression of an image of a page full of text). LZW or Flate compression of the content streams describing all other text and graphics in the document results in compression ratios of approximately 2:1. All of these compression filters produce binary data, which can then be further converted to ASCII base-85 encoding if a 7-bit ASCII representation is desired.

2.2.3 Font Management

Managing fonts is a fundamental challenge in document interchange. Generally, the receiver of a document must have the same fonts that were originally used to create it. If a different font is substituted, its character set, glyph shapes, and metrics may differ from those in the original font. This can produce unexpected and undesirable results, such as lines of text extending into margins or overlapping with graphics.

PDF provides various means for dealing with font management:

- The original font programs can be embedded in the PDF file. PDF supports various font formats, including Type 1, TrueType®, and CID-keyed fonts. This ensures the most predictable and dependable results.
- To conserve space, a font subset can be embedded, containing just the glyph descriptions for those characters that are actually used in the document. Also, Type 1 fonts can be represented in a special compact format.
- PDF prescribes a set of 14 standard fonts that can be used without prior definition. These include four faces each of three Latin text typefaces (Courier, Helvetica*, and Times*), as well as two symbolic fonts (Symbol and ITC Zapf Dingbats®). These fonts, or suitable substitute fonts with the same metrics, are required to be available in all PDF viewer applications.
- A PDF file can refer by name to fonts that are not embedded in the PDF file. In this case, a viewer application will use those fonts if they are available in the viewer's environment. This approach suffers from the uncertainties noted above.
- A PDF file contains a *font descriptor* for each font that it uses (other than the standard 14). The font descriptor includes font metrics and style information, enabling a viewer application to select or synthesize a suitable substitute font if necessary. Although the glyphs' shapes will differ from those intended, their placement will be accurate.

Font management is primarily concerned with producing the correct appearance of text—that is, the shape and placement of glyphs. However, it is sometimes necessary for a PDF application to extract the meaning of the text, represented in some standard information encoding such as Unicode. In some cases, this information can be deduced from the encoding used to represent the text in the PDF file. Otherwise, the PDF producer application should specify the mapping explicitly by including a special object, the ***ToUnicode*** *CMap*.

2.2.4 Single-Pass File Generation

Because of system limitations and efficiency considerations, it may be necessary or desirable for an application program to generate a PDF file in a single pass. For example, the program may have limited memory available or be unable to open temporary files. For this reason, PDF supports single-pass generation of files. Although some PDF objects must specify their length in bytes, a mechanism is provided allowing the length to follow the object itself in the PDF file. In addition, information such as the number of pages in the document can be written into the file after all pages have been generated.

A PDF file that is generated in a single pass is generally not ordered for most efficient viewing, particularly when accessing the contents of the file over a network. When generating a PDF file that is intended to be viewed many times, it is worthwhile to perform a second pass to optimize the order in which objects occur in the file. PDF specifies a particular file organization, *Linearized PDF*, which is documented in Appendix F. Other optimizations are also possible, such as detecting duplicated sequences of graphics objects and collapsing them to a single shared sequence that is specified only once.

2.2.5 Random Access

A PDF file should be thought of as a flattened representation of a data structure consisting of a collection of objects that can refer to each other in any arbitrary way. The order of the objects' occurrence in the PDF file has no semantic significance. In general, a viewer application should process a PDF file by following references from object to object, rather than by processing objects sequentially. This is particularly important for interactive document viewing or for any application in which pages or other objects in the PDF file are accessed out of sequence.

To support such random access to individual objects, every PDF file contains a *cross-reference table* that can be used to locate and directly access pages and other important objects within the file. The cross-reference table is stored at the end of the file, allowing applications that generate PDF files in a single pass to store it easily and those that read PDF files to locate it easily. Using the cross-reference table makes the time needed to locate a page or other object nearly independent of the length of the document. This allows PDF documents containing hundreds or thousands of pages to be accessed efficiently.

2.2.6 Security

PDF has two security features that can be used, separately or together, in any document:

- The document can be *encrypted* so that only authorized users can access it. There is separate authorization for the owner of the document and for all other users; the users' access can be selectively restricted to allow only certain operations, such as viewing, printing, or editing.

- The document can be digitally *signed* to certify its authenticity. The signature may take many forms, including a document digest that has been encrypted with a public/private key, a biometric signature such as a fingerprint, and others. Any subsequent changes to a signed PDF file will invalidate the signature.

2.2.7 Incremental Update

Applications may allow users to modify PDF documents. Users should not have to wait for the entire file—which can contain hundreds of pages or more—to be rewritten each time modifications to the document are saved. PDF allows modifications to be appended to a file, leaving the original data intact. The addendum appended when a file is incrementally updated contains only those objects that were actually added or modified, and includes an update to the cross-reference table. Incremental update allows an application to save modifications to a PDF document in an amount of time proportional to the size of the modification rather than the size of the file.

In addition, because the original contents of the document are still present in the file, it is possible to undo saved changes by deleting one or more addenda. The ability to recover the exact contents of an original document is critical when digital signatures have been applied and subsequently need to be verified.

2.2.8 Extensibility

PDF is designed to be extensible. Not only can new features be added, but applications based on earlier versions of PDF can behave reasonably when they encounter newer features that they do not understand. Appendix H describes how a PDF viewer application should behave in such cases.

Additionally, PDF provides means for applications to store their own private information in a PDF file. This information can be recovered when the file is imported by the same application, but is ignored by other applications. This allows PDF to serve as an application's native file format while allowing its documents to be viewed and printed by other applications. Application-specific data can be stored either as *marked content* annotating the graphics objects in a PDF content stream or as entirely separate objects unconnected with the PDF content.

2.3 Creating PDF

PDF files may be produced either directly by application programs or indirectly by conversion from other file formats or imaging models. As PDF documents and applications that process them become more prevalent, new ways of creating and using PDF will be invented. One of the goals of this book is to make the file format accessible so that application developers can expand on the ideas behind PDF and the applications that initially support it.

Many applications can generate PDF files directly, and some can import them as well. This is the most desirable approach, since it gives the application access to the full capabilities of PDF, including the imaging model and the interactive and document interchange features. Alternatively, existing applications that do not generate PDF directly can still be used to produce PDF output by indirect methods. There are two principal ways of doing this:

- The application describes its printable output by making calls to an application programming interface (API) such as GDI in Microsoft® Windows® or QuickDraw in the Apple® Mac® OS. A software component called a *printer driver* intercepts these calls and interprets them to generate output in PDF form.
- The application produces printable output directly in some other file format, such as PostScript, PCL, HPGL, or DVI, which is then converted into PDF by a separate translation program.

Note, however, that while these indirect strategies are often the easiest way to obtain PDF output from an existing application, the resulting PDF files may not make the best use of the high-level Adobe imaging model. This is because the information embodied in the application's API calls or in the intermediate output file often describes the desired results at too low a level; any higher-level information maintained by the original application has been lost and is not available to the printer driver or translator.

Figures 2.1 and 2.2 show how Adobe Acrobat products support these indirect approaches. The Adobe PDF printer (Figure 2.1), available on the Windows and Mac OS platforms, acts as a printer driver, intercepting graphics and text operations generated by a running application program through the operating system's API. Instead of converting these operations into printer commands and transmitting them directly to a printer, Adobe PDF converts them to equivalent PDF operators and embeds them in a PDF file. The result is a platform-independent file that can be viewed and printed by a PDF viewer application, such as Adobe Acrobat, running on any supported platform—even a different platform from the one on which the file was originally generated.

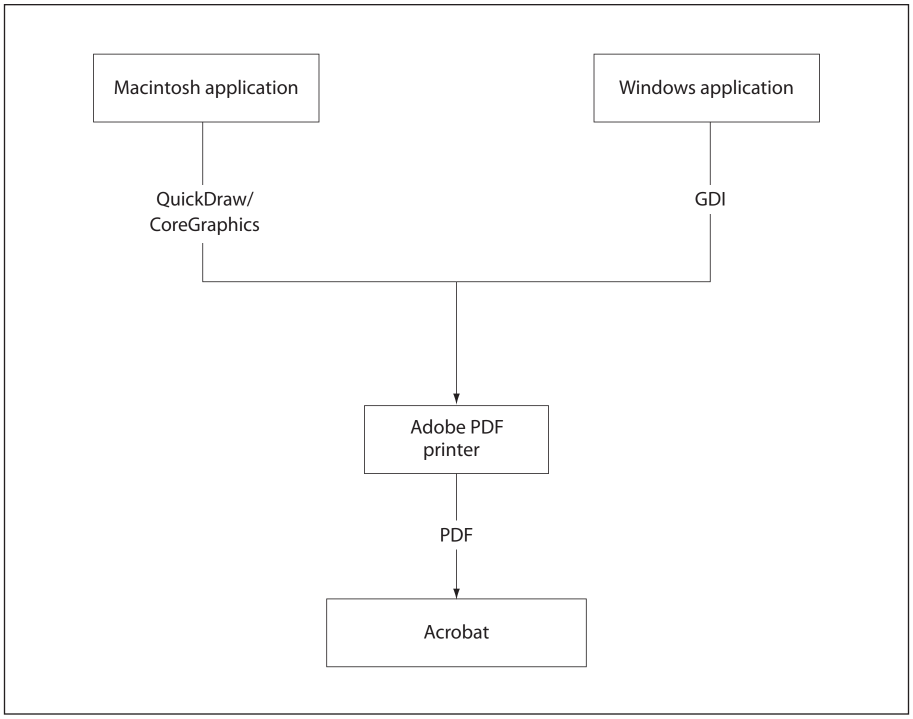


FIGURE 2.1 *Creating PDF files using the Adobe PDF printer*

Instead of describing their printable output via API calls, some applications produce PostScript page descriptions directly—either because of limitations in the QuickDraw or GDI imaging models or because the applications run on platforms such as DOS or UNIX®, where no system-level printer driver exists. PostScript files generated by such applications can be converted into PDF files using the Acrobat Distiller® application (see Figure 2.2). Because PostScript and PDF share the same Adobe imaging model, Acrobat Distiller can preserve the exact graphical content of the PostScript file in the translation to PDF. Additionally, Distiller supports a PostScript language extension, called **pdfmark**, that allows the producing application to embed instructions in the PostScript file for creating hypertext links, logical structure, and other interactive and document interchange features of PDF. Again, the resulting PDF file can be viewed with a viewer application, such as Adobe Acrobat, on any supported platform.

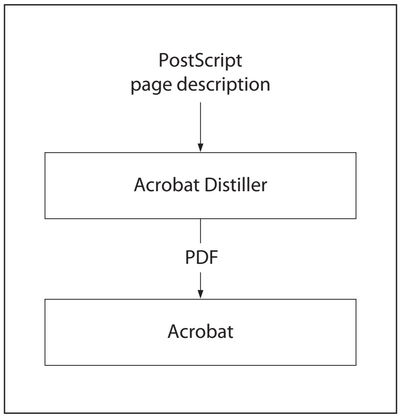


FIGURE 2.2 *Creating PDF files using Acrobat Distiller*

2.4 PDF and the PostScript Language

The PDF operators for setting the graphics state and painting graphics objects are similar to the corresponding operators in the PostScript language. Unlike PostScript, however, PDF is not a full-scale programming language; it trades reduced flexibility for improved efficiency and predictability. PDF therefore differs from PostScript in the following significant ways:

- PDF enforces a strictly defined file structure that allows an application to access parts of a document in arbitrary order.
- To simplify the processing of content streams, PDF does not include common programming language features such as procedures, variables, and control constructs.
- PDF files contain information such as font metrics to ensure viewing fidelity.
- A PDF file may contain additional information that is not directly connected with the imaging model, such as hypertext links for interactive viewing and logical structure information for document interchange.

Because of these differences, a PDF file generally cannot be transmitted directly to a PostScript output device for printing (although a few such devices do also support PDF directly). An application printing a PDF document to a PostScript device must carry out the following steps:

1. Insert *procedure sets* containing PostScript procedure definitions to implement the PDF operators.
2. Extract the content for each page. Each content stream is essentially the script portion of a traditional PostScript program using very specific procedures, such as **m** for **moveto** and **l** for **lineto**.
3. Decode compressed text, graphics, and image data as necessary. The compression filters used in PDF are compatible with those used in PostScript; they may or may not be supported, depending on the LanguageLevel of the target output device.
4. Insert any needed resources, such as fonts, into the PostScript file. These can be either the original fonts themselves or suitable substitute fonts based on the font metrics in the PDF file. Fonts may need to be converted to a format that the PostScript interpreter recognizes, such as Type 1 or Type 42.
5. Put the information in the correct order. The result is a traditional PostScript program that fully represents the visual aspects of the document but no longer contains PDF elements such as hypertext links, annotations, and bookmarks.
6. Transmit the PostScript program to the output device.

CHAPTER 3

Syntax

THIS CHAPTER COVERS everything about the syntax of PDF at the object, file, and document level. It sets the stage for subsequent chapters, which describe how the contents of a PDF file are interpreted as page descriptions, interactive navigational aids, and application-level logical structure.

PDF syntax is best understood by thinking of it in four parts, as shown in Figure 3.1:

- *Objects*. A PDF document is a data structure composed from a small set of basic types of data object. Section 3.1, "Lexical Conventions," describes the character set used to write objects and other syntactic elements. Section 3.2, "Objects," describes the syntax and essential properties of the objects themselves. Section 3.2.7, "Stream Objects," provides complete details of the most complex data type, the stream object.
- *File structure*. The PDF file structure determines how objects are stored in a PDF file, how they are accessed, and how they are updated. This structure is independent of the semantics of the objects. Section 3.4, "File Structure," describes the file structure. Section 3.5, "Encryption," describes a file-level mechanism for protecting a document's contents from unauthorized access.
- *Document structure*. The PDF document structure specifies how the basic object types are used to represent components of a PDF document: pages, fonts, annotations, and so forth. Section 3.6, "Document Structure," describes the overall document structure; later chapters address the detailed semantics of the components.
- *Content streams*. A PDF *content stream* contains a sequence of instructions describing the appearance of a page or other graphical entity. These instructions, while also represented as objects, are conceptually distinct from the objects that

represent the document structure and are described separately. Section 3.7, “Content Streams and Resources,” discusses PDF content streams and their associated resources.

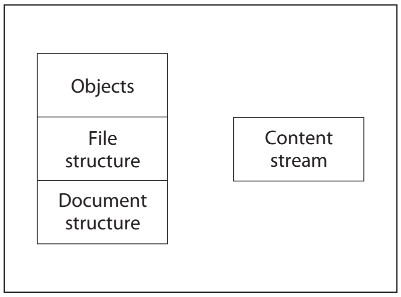


FIGURE 3.1 *PDF components*

In addition, this chapter describes some data structures, built from basic objects, that are so widely used that they can almost be considered basic object types in their own right. These objects are covered in Sections 3.8, “Common Data Structures”; 3.9, “Functions”; and 3.10, “File Specifications.”

PDF’s object and file syntax is also used as the basis for other file formats. These include the Forms Data Format (FDF), described in Section 8.6.6, “Forms Data Format,” and the Portable Job Ticket Format (PJTF), described in Adobe Technical Note #5620, *Portable Job Ticket Format*.

3.1 Lexical Conventions

At the most fundamental level, a PDF file is a sequence of 8-bit bytes. These bytes can be grouped into *tokens* according to the syntax rules described below. One or more tokens are then assembled to form higher-level syntactic entities, principally *objects*, which are the basic data values from which a PDF document is constructed.

PDF can be entirely represented using byte values corresponding to the visible printable subset of the ASCII character set, plus characters that appear as “white space,” such as space, tab, carriage return, and line feed characters. ASCII is the American Standard Code for Information Interchange, a widely used convention for encoding a specific set of 128 characters as binary numbers. However, a PDF file is not restricted to the ASCII character set; it can contain arbitrary 8-bit bytes, subject to the following considerations:

- The tokens that delimit objects and that describe the structure of a PDF file are all written in the ASCII character set, as are all the reserved words and the names used as keys in standard dictionaries.
- The data values of certain types of object—strings and streams—can be but need not be written entirely in ASCII. For the purpose of exposition (as in this book), ASCII representation is preferred. However, in actual practice, data that is naturally binary, such as sampled images, is represented directly in binary for the sake of compactness and efficiency.
- A PDF file containing binary data must be transported and stored by means that preserve all bytes of the file faithfully; that is, as a binary file rather than a text file. Such a file is not portable to environments that impose reserved character codes, maximum line lengths, end-of-line conventions, or other restrictions.

Note: *In this chapter, the term* character *is synonymous with* byte *and merely refers to a particular 8-bit value. This is entirely independent of any logical meaning that the value may have when it is treated as data in specific contexts, such as representing human-readable text or selecting a glyph from a font.*

3.1.1 Character Set

The PDF character set is divided into three classes, called *regular*, *delimiter*, and *white-space* characters. This classification determines the grouping of characters into tokens, except within strings, streams, and comments; different rules apply in those contexts.

White-space characters (see Table 3.1) separate syntactic constructs such as names and numbers from each other. All white-space characters are equivalent, except in comments, strings, and streams. In all other contexts, PDF treats any sequence of consecutive white-space characters as if there were just one.

TABLE 3.1 White-space characters

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

The carriage return (CR) and line feed (LF) characters, also called *newline characters*, are treated as *end-of-line* (EOL) markers. The combination of a carriage return followed immediately by a line feed is treated as one EOL marker. For the most part, EOL markers are treated the same as any other white-space characters. However, there are certain instances in which an EOL marker is required or recommended—that is, the following token must appear at the beginning of a line.

***Note:** The examples in this book illustrate a recommended convention for arranging tokens into lines. However, the examples' use of white space for indentation is purely for clarity of exposition and is not recommended for practical use.*

The *delimiter characters* (,), <, >, [,], {, }, /, and % are special. They delimit syntactic entities such as strings, arrays, names, 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 8-bit binary characters that are outside the ASCII character set. A sequence of consecutive regular characters comprises a single token.

***Note:** PDF is case-sensitive; corresponding uppercase and lowercase letters are considered distinct.*

3.1.2 Comments

Any occurrence of the percent sign character (%) outside a string or stream introduces a *comment*. The comment consists of all characters between the percent sign and the end of the line, including regular, delimiter, space, and tab characters. PDF ignores comments, treating them as if they were single white-space characters. That is, a comment separates the token preceding it from the one following; thus the PDF fragment

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

is syntactically equivalent to just the tokens abc and 123.

Comments (other than the %PDF–1.4 and %%EOF comments described in Section 3.4, "File Structure") have no semantics. They are not necessarily preserved by applications that edit PDF files (see implementation note 2 in Appendix H). In particular, there is no PDF equivalent of the PostScript document structuring conventions (DSC).

3.2 Objects

PDF supports eight basic types of object:

- Boolean values
- Integer and real numbers
- Strings
- Names
- Arrays
- Dictionaries
- Streams
- The null object

Objects may be labeled so that they can be referred to by other objects. A labeled object is called an *indirect object*.

The following sections describe each object type, as well as how to create and refer to indirect objects.

3.2.1 Boolean Objects

PDF provides *boolean objects* identified by the keywords **true** and **false**. Boolean objects can be used as the values of array elements and dictionary entries, and can also occur in PostScript calculator functions as the results of boolean and relational operators and as operands to the conditional operators **if** and **ifelse** (see Section 3.9.4, "Type 4 (PostScript Calculator) Functions").

3.2.2 Numeric Objects

PDF 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, but with limited range and precision; they are typically represented in fixed-point, rather than floating-point, form. The range and precision of numbers are limited by the internal representations used in the machine on which the PDF viewer application is running; Appendix C gives these limits for typical implementations.

An integer is written one or more decimal digits optionally preceded by a sign:

```
123    43445    +17    −98    0
```

The value 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.

A real value is written as one or more decimal digits with an optional sign and a leading, trailing, or embedded period (decimal point):

```
34.5    −3.62    +123.6    4.    −.002    0.0
```

The value is interpreted as a real number and is converted to a real object. If it exceeds the implementation limit for real numbers, an error occurs.

Note: *PDF does not support the PostScript syntax for numbers with nondecimal radices (such as 16#FFFE) or in exponential format (such as 6.02E23).*

Throughout this book, the term *number* refers to an object whose type may be either integer or real. Wherever a real number is expected, an integer may be used instead and will be automatically converted to an equivalent real value. For example, it is not necessary to write the number 1.0 in real format; the integer 1 will suffice.

3.2.3 String Objects

A *string object* consists of a series of bytes—unsigned integer values in the range 0 to 255. The string elements are not integer objects, but are stored in a more compact format. The length of a string is subject to an implementation limit; see Appendix C.

There are two conventions, described in the following sections, for writing a string object in PDF:

- As a sequence of literal characters enclosed in parentheses ( )
- As hexadecimal data enclosed in angle brackets < >

This section describes only the basic syntax for writing a string as a sequence of bytes. Strings can be used for many purposes and can be formatted in a variety of ways. When a string is used for a specific purpose (to represent a date, for example), it is useful to have a standard format for that purpose (see Section 3.8.3, "Dates"). Such formats are merely conventions for interpreting the contents of a string and are not in themselves separate object types. The use of a particular format is described with the definition of the string object that uses that format.

Literal Strings

A *literal string* is written as an arbitrary number of characters enclosed in parentheses. Any characters may appear in a string except unbalanced parentheses and the backslash, which must be treated specially. Balanced pairs of parentheses within a string require no special treatment.

The following are valid literal strings:

```
(This is a string)
(Strings may contain newlines
and such.)
```

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

Within a literal string, the backslash (\) is used as an escape character for various purposes, such as to include newline characters, nonprinting ASCII characters, unbalanced parentheses, or the backslash character itself in the string. The character immediately following the backslash determines its precise interpretation (see Table 3.2). If the character following the backslash is not one of those shown in the table, the backslash is ignored.

TABLE 3.2 Escape sequences in literal strings

SEQUENCE	MEANING
\n	Line feed (LF)
\r	Carriage return (CR)
\t	Horizontal tab (HT)
\b	Backspace (BS)
\f	Form feed (FF)
\(	Left parenthesis
\)	Right parenthesis
\\	Backslash
\ddd	Character code *ddd* (octal)

If a string is too long to be conveniently placed on a single line, it may be split across multiple lines by using the backslash character at the end of a line to indicate that the string continues on the following line. The backslash and the end-of-line marker following it are not considered part of the string. For example:

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

If an end-of-line marker appears within a literal string without a preceding backslash, the result is equivalent to \n (regardless of whether the end-of-line marker itself was a carriage return, a line feed, or both). For example:

```
(This string has an end–of–line at the end of it.
)
(So does this one.\n)
```

The *ddd* escape sequence provides a way to represent characters outside the printable ASCII character set. For example:

```
(This string contains \245two octal characters\307.)
```

The number *ddd* may consist of one, two, or three octal digits, with high-order overflow ignored. It is required that three octal digits be used, with leading zeros as needed, if the next character of the string is also a digit. For example, the literal

```
(\0053)
```

denotes a string containing two characters, \005 (Control-E) followed by the digit 3, whereas both

```
(\053)
```

and

```
(\53)
```

denote strings containing the single character \053, a plus sign (+).

This notation provides a way to specify characters outside the 7-bit ASCII character set using ASCII characters only. However, any 8-bit value may appear in a string. In particular, when a document is encrypted (see Section 3.5, “Encryption”), all of its strings are encrypted and often contain arbitrary 8-bit values. Note that the backslash character is still required as an escape to specify unbalanced parentheses or the backslash character itself.

Hexadecimal Strings

Strings may also be written in hexadecimal form; this is useful for including arbitrary binary data in a PDF file. A hexadecimal string is written as a sequence of hexadecimal digits (0–9 and either A–F or a–f) enclosed within angle brackets (< and >):

```
<4E6F762073686D6F7A206B6120706F702E>
```

Each pair of hexadecimal digits defines one byte of the string. White-space characters (such as space, tab, carriage return, line feed, and form feed) are ignored.

If the final digit of a hexadecimal string is missing—that is, if there is an odd number of digits—the final digit is assumed to be 0. For example,

```
<901FA3>
```

is a 3-byte string consisting of the characters whose hexadecimal codes are 90, 1F, and A3, but

```
<901FA>
```

is a 3-byte string containing the characters whose hexadecimal codes are 90, 1F, and A0.

3.2.4 Name Objects

A *name object* is an atomic symbol uniquely defined by a sequence of characters. *Uniquely defined* means that any two name objects made up of the same sequence of characters are identically the same object. *Atomic* means that a name has no internal structure; although it is defined by a sequence of characters, those characters are not "elements" of the name.

A slash character (/) introduces a name. The slash is not part of the name itself, but a prefix indicating that the following sequence of characters constitutes a name. There can be no white-space characters between the slash and the first character in the name. The name may include any regular characters, but not delimiter or white-space characters (see Section 3.1, "Lexical Conventions").

Uppercase and lowercase letters are considered distinct: /A and /a are different names. The following are examples of valid literal names:

```
/Name1
/ASomewhatLongerName
/A;Name_With–Various***Characters?
/1.2
/$$
/@pattern
/.notdef
```

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

Beginning with PDF 1.2, any character except null (character code 0) may be included in a name by writing its 2-digit hexadecimal code, preceded by the number sign character (#); see implementation notes 3 and 4 in Appendix H. This syntax is required in order to represent any of the delimiter or white-space characters or the number sign character itself; it is recommended but not required for characters whose codes are outside the range 33 (!) to 126 (~). The examples shown in Table 3.3 are valid literal names in PDF 1.2 and higher.

TABLE 3.3 Examples of literal names using the # character

LITERAL NAME	RESULT
/Adobe#20Green	Adobe Green
/PANTONE#205757#20CV	PANTONE 5757 CV
/paired#28#29parentheses	paired()parentheses
/The_Key_of_F#23_Minor	The_Key_of_F#_Minor
/A#42	AB

The length of a name is subject to an implementation limit; see Appendix C. The limit applies to the number of characters in the name's internal representation. For example, the name /A#20B has four characters (/, A, space, B), not six.

As stated above, name objects are treated as atomic symbols within a PDF file. Ordinarily, the bytes making up the name are never treated as text to be presented to a human user or to an application external to a PDF viewer. However, occasionally the need arises to treat a name object as text, such as one that represents a font name (see the **BaseFont** entry in Table 5.8 on page 375) or a structure type (see Section 10.6.2, "Structure Types").

In such situations, it is recommended that the sequence of bytes (after expansion of # sequences, if any) be interpreted according to UTF-8, a variable-length byte-encoded representation of Unicode in which the printable ASCII characters have the same representations as in ASCII. This enables a name object to represent text in any natural language, subject to the implementation limit on the length of a name. (See implementation note 5 in Appendix H.)

Note: *PDF does not prescribe what UTF-8 sequence to choose for representing any given piece of externally specified text as a name object. In some cases, there are multiple UTF-8 sequences that could represent the same logical text. Name objects defined by different sequences of bytes constitute distinct name objects in PDF, even though the UTF-8 sequences might have identical external interpretations.*

In PDF, name objects always begin with the slash character (/), unlike keywords such as **true**, **false**, and **obj**. This book follows a typographic convention of writing names without the leading slash when they appear in running text and tables. For example, **Type** and FullScreen denote names that would actually be written in a PDF file (and in code examples in this book) as /Type and /FullScreen.

3.2.5 Array Objects

An *array object* is a one-dimensional collection of objects arranged sequentially. Unlike arrays in many other computer languages, PDF arrays may be heterogeneous; that is, an array's elements may be any combination of numbers, strings, dictionaries, or any other objects, including other arrays. The number of elements in an array is subject to an implementation limit; see Appendix C.

An array is written as a sequence of objects enclosed in square brackets ([and]):

```
[549 3.14 false (Ralph) /SomeName]
```

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

3.2.6 Dictionary Objects

A *dictionary object* is an associative table containing pairs of objects, known as the dictionary's *entries*. The first element of each entry is the *key* and the second element is the *value*. The key must be a name (unlike dictionary keys in PostScript, which may be objects of any type). The value can be any kind of object, including another dictionary. A dictionary entry whose value is **null** (see Section 3.2.8, "Null Object") is equivalent to an absent entry. (Note that this differs from PostScript, where **null** behaves like any other object as the value of a dictionary entry.) The number of entries in a dictionary is subject to an implementation limit; see Appendix C.

Note: *No two entries in the same dictionary should have the same key. If a key does appear more than once, its value is undefined.*

A dictionary is written as a sequence of key-value pairs enclosed in double angle brackets (<<…>>). For example:

```
<< /Type /Example
   /Subtype /DictionaryExample
   /Version 0.01
   /IntegerItem 12
   /StringItem (a string)
   /Subdictionary << /Item1 0.4
                     /Item2 true
                     /LastItem (not!)
                     /VeryLastItem (OK)
                  >>
>>
```

Note: *Do not confuse the double angle brackets with single angle brackets (< and >), which delimit a hexadecimal string (see "Hexadecimal Strings" on page 32).*

Dictionary objects are the main building blocks of a PDF document. They are commonly used to collect and tie together the attributes of a complex object, such as a font or a page of the document, with each entry in the dictionary specifying the name and value of an attribute. By convention, the **Type** entry of such a dictionary identifies the type of object the dictionary describes. In some cases, a **Subtype** entry (sometimes abbreviated **S**) is used to further identify a specialized subcategory of the general type. The value of the **Type** or **Subtype** entry is always a name. For example, in a font dictionary, the value of the **Type** entry is always **Font**, whereas that of the **Subtype** entry may be **Type1**, **TrueType**, or one of several other values.

The value of the **Type** entry can almost always be inferred from context. The operand of the **Tf** operator, for example, must be a font object, so the **Type** entry in a font dictionary serves primarily as documentation and as information for error checking. The **Type** entry is not required unless so stated in its description; however, if the entry is present, it must have the correct value. In addition, the value of the **Type** entry in any dictionary, even in private data, must be either a name defined in this book or a registered name; see Appendix E for details.

3.2.7 Stream Objects

A *stream object*, like a string object, is a sequence of bytes. However, a PDF application can read a stream incrementally, while a string must be read in its entirety. Furthermore, a stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams.

***Note:** As with strings, this section describes only the syntax for writing a stream as a sequence of bytes. What those bytes represent is determined by the context in which the stream is referenced.*

A stream consists of a dictionary that describes a sequence of bytes, followed by zero or more bytes bracketed between the keywords **stream** and **endstream**:

dictionary
stream
…Zero or more bytes…
endstream

All streams must be indirect objects (see Section 3.2.9, "Indirect Objects") and the stream dictionary must be a direct object. The keyword **stream** that follows the stream dictionary should be followed by an end-of-line marker consisting of either a carriage return and a line feed or just a line feed, and not by a carriage return alone. The sequence of bytes that make up a stream lie between the **stream** and **endstream** keywords; the stream dictionary specifies the exact number of bytes. It is recommended that there be an end-of-line marker after the data and before **endstream**; this marker is not included in the stream length.

Alternatively, in PDF 1.2 the bytes may be contained in an external file, in which case the stream dictionary specifies the file and any bytes between **stream** and **endstream** are ignored. (See implementation note 6 in Appendix H.)

Note: *Without the restriction against following the keyword* ***stream*** *by a carriage return alone, it would be impossible to differentiate a stream that uses carriage return as its end-of-line marker and has a line feed as its first byte of data from one that uses a carriage return–line feed sequence to denote end-of-line.*

Table 3.4 lists the entries common to all stream dictionaries; certain types of stream may have additional dictionary entries, as indicated where those streams are described. The optional entries regarding *filters* for the stream indicate whether and how the data in the stream must be transformed ("decoded") before it is used. Filters are described further in Section 3.3, "Filters."

Stream Extent

Every stream dictionary has a **Length** entry that indicates how many bytes of the PDF file are used for the stream's data. (If the stream has a filter, **Length** is the number of bytes of *encoded* data.) In addition, most filters are defined so that the data is self-limiting; that is, they use an encoding scheme in which an explicit *end-of-data* (EOD) marker delimits the extent of the data. Finally, streams are used to represent many objects from whose attributes a length can be inferred. *All of these constraints must be consistent.*

For example, an image with 10 rows and 20 columns, using a single color component and 8 bits per component, requires exactly 200 bytes of image data. If the stream uses a filter, there must be enough bytes of encoded data in the PDF file to produce those 200 bytes. An error occurs if **Length** is too small, if an explicit EOD marker occurs too soon, or if the decoded data does not contain 200 bytes.

It is also an error if the stream contains too *much* data, with the exception that there may be an extra end-of-line marker in the PDF file before the keyword **endstream**.

TABLE 3.4 Entries common to all stream dictionaries

KEY	TYPE	VALUE
Length	integer	*(Required)* The number of bytes from the beginning of the line following the keyword **stream** to the last byte just before the keyword **endstream**. (There may be an additional EOL marker, preceding **endstream**, that is not included in the count and is not logically part of the stream data.) See "Stream Extent," above, for further discussion.
Filter	name or array	*(Optional)* The name of a filter to be applied in processing the stream data found between the keywords **stream** and **endstream**, or an array of such names. Multiple filters should be specified in the order in which they are to be applied.
DecodeParms	dictionary or array	*(Optional)* A parameter dictionary, or an array of such dictionaries, used by the filters specified by **Filter**. If there is only one filter and that filter has parameters, **DecodeParms** must be set to the filter's parameter dictionary unless all the filter's parameters have their default values, in which case the **DecodeParms** entry may be omitted. If there are multiple filters and any of the filters has parameters set to nondefault values, **DecodeParms** must be an array with one entry for each filter: either the parameter dictionary for that filter, or the null object if that filter has no parameters (or if all of its parameters have their default values). If none of the filters have parameters, or if all their parameters have default values, the **DecodeParms** entry may be omitted. (See implementation note 7 in Appendix H.)
F	file specification	*(Optional; PDF 1.2)* The file containing the stream data. If this entry is present, the bytes between **stream** and **endstream** are ignored, the filters are specified by **FFilter** rather than **Filter**, and the filter parameters are specified by **FDecodeParms** rather than **DecodeParms**. However, the **Length** entry should still specify the number of those bytes. (Usually there are no bytes and **Length** is 0.)
FFilter	name or array	*(Optional; PDF 1.2)* The name of a filter to be applied in processing the data found in the stream's external file, or an array of such names. The same rules apply as for **Filter**.
FDecodeParms	dictionary or array	*(Optional; PDF 1.2)* A parameter dictionary, or an array of such dictionaries, used by the filters specified by **FFilter**. The same rules apply as for **DecodeParms**.

KEY	TYPE	VALUE
DL	integer	*(Optional; PDF 1.5)* A non-negative integer representing the number of bytes in the decoded (defiltered) stream. It can be used to determine, for example, whether enough disk space is available to write a stream to a file. This value should be considered a hint only; for some stream filters, it may not be possible to determine this value precisely.

3.2.8 Null Object

The *null object* has a type and value that are unequal to those of any other object. There is only one object of type null, denoted by the keyword **null**. An indirect object reference (see Section 3.2.9, "Indirect Objects") to a nonexistent object is treated the same as a null object; specifying the null object as the value of a dictionary entry (Section 3.2.6, "Dictionary Objects") is equivalent to omitting the entry entirely.

3.2.9 Indirect Objects

Any object in a PDF file may be labeled as an *indirect object*. This gives the object a unique *object identifier* by which other objects can refer to it (for example, as an element of an array or as the value of a dictionary entry). The object identifier consists of two parts:

- A positive integer *object number*. Indirect objects are often numbered sequentially within a PDF file, but this is not required; object numbers may be assigned in any arbitrary order.
- A nonnegative integer *generation number*. In a newly created file, all indirect objects have generation numbers of 0. Nonzero generation numbers may be introduced when the file is later updated; see Sections 3.4.3, "Cross-Reference Table," and 3.4.5, "Incremental Updates."

Together, the combination of an object number and a generation number uniquely identifies an indirect object. The object retains the same object number and generation number throughout its existence, even if its value is modified.

The definition of an indirect object in a PDF file consists of its object number and generation number, followed by the value of the object itself bracketed between the keywords **obj** and **endobj**. For example, the definition

```
12  0  obj
    (Brillig)
endobj
```

defines an indirect string object with an object number of 12, a generation number of 0, and the value Brillig.

The object can then be referred to from elsewhere in the file by an *indirect reference* consisting of the object number, the generation number, and the keyword **R**:

```
12  0  R
```

Beginning with PDF 1.5, indirect objects may reside in object streams (see Section 3.4.6, "Object Streams"). They are referred to in the same way; however, their definition does not include the keywords **obj** and **endobj**.

An indirect reference to an undefined object is not an error; it is simply treated as a reference to the null object. For example, if a file contains the indirect reference

```
17  0  R
```

but does not contain the corresponding definition

```
17  0  obj
    …
endobj
```

then the indirect reference is considered to refer to the null object.

***Note:** In the data structures that make up a PDF document, certain values are required to be specified as indirect object references. Except where this is explicitly called out, any object (other than a stream) may be specified either directly or as an indirect object reference; the semantics are entirely equivalent. Note in particular that content streams, which define the visible contents of the document, may not contain indirect references (see Section 3.7.1, "Content Streams"). Also, see implementation note 8 in Appendix H.*

Example 3.1 shows the use of an indirect object to specify the length of a stream. The value of the stream's **Length** entry is an integer object that follows the stream itself in the file. This allows applications that generate PDF in a single pass to defer specifying the stream's length until after its contents have been generated.

Example 3.1

```
7  0  obj
   << /Length  8 0 R  >>                     % An indirect reference to object 8
stream
   BT
      /F1  12  Tf
      72  712  Td
      (A stream with an indirect length)  Tj
   ET
endstream
endobj

8  0  obj
   77                                        % The length of the preceding stream
endobj
```

3.3 Filters

Stream filters are introduced in Section 3.2.7, "Stream Objects." A *filter* is an optional part of the specification of a stream, indicating how the data in the stream must be decoded before it is used. For example, if a stream has an **ASCIIHexDecode** filter, an application reading the data in that stream will transform the ASCII hexadecimal-encoded data in the stream into binary data.

An application program that produces a PDF file can encode certain information (for example, data for sampled images) to compress it or to convert it to a portable ASCII representation. Then an application that reads ("consumes") the PDF file can invoke the corresponding decoding filter to convert the information back to its original form.

The filter or filters for a stream are specified by the **Filter** entry in the stream's dictionary (or the **FFilter** entry if the stream is external). Filters can be cascaded to form a *pipeline* that passes the stream through two or more decoding transformations in sequence. For example, data encoded using LZW and ASCII base-85 encoding (in that order) can be decoded using the following entry in the stream dictionary:

```
/Filter [/ASCII85Decode /LZWDecode]
```

Some filters may take parameters to control how they operate. These optional parameters are specified by the **DecodeParms** entry in the stream's dictionary (or the **FDecodeParms** entry if the stream is external).

PDF supports a standard set of filters that fall into two main categories:

- *ASCII filters* enable decoding of arbitrary 8-bit binary data that has been encoded as ASCII text. (See Section 3.1, "Lexical Conventions," for an explanation of why this type of encoding might be useful.) Note that ASCII filters serve no useful purpose in a PDF file that is encrypted; see Section 3.5, "Encryption."
- *Decompression filters* enable decoding of data that has been compressed. The compressed data is always in 8-bit binary format, even if the original data happens to be ASCII text. (Compression is particularly valuable for large sampled images, since it reduces storage requirements and transmission time. Note that some types of compression are *lossy*, meaning that some data is lost during the encoding, resulting in a loss of quality when the data is decompressed; compression in which no loss of data occurs is called *lossless*.)

The standard filters are summarized in Table 3.5, which also indicates whether they accept any optional parameters. The following sections describe these filters and their parameters (if any) in greater detail, including specifications of encoding algorithms for some filters. (See also implementation notes 9 and 10 in Appendix H.)

TABLE 3.5 Standard filters

FILTER NAME	PARAMETERS?	DESCRIPTION
ASCIIHexDecode	no	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
ASCII85Decode	no	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
LZWDecode	yes	Decompresses data encoded using the LZW (Lempel-Ziv-Welch) adaptive compression method, reproducing the original text or binary data.

FILTER NAME	**PARAMETERS?**	**DESCRIPTION**
FlateDecode	yes	*(PDF 1.2)* Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
RunLengthDecode	no	Decompresses data encoded using a byte-oriented run-length encoding algorithm, reproducing the original text or binary data (typically monochrome image data, or any data that contains frequent long runs of a single byte value).
CCITTFaxDecode	yes	Decompresses data encoded using the CCITT facsimile standard, reproducing the original data (typically monochrome image data at 1 bit per pixel).
JBIG2Decode	yes	*(PDF 1.4)* Decompresses data encoded using the JBIG2 standard, reproducing the original monochrome (1 bit per pixel) image data (or an approximation of that data).
DCTDecode	yes	Decompresses data encoded using a DCT (discrete cosine transform) technique based on the JPEG standard, reproducing image sample data that approximates the original data.
JPXDecode	no	*(PDF 1.5)* Decompresses data encoded using the wavelet-based JPEG2000 standard, reproducing the original image data.
Crypt	yes	*(PDF 1.5)* Decrypts data encrypted by a security handler, reproducing the original data as it was prior to encryption.

Example 3.2 shows a stream, containing the marking instructions for a page, that was compressed using the LZW compression method and then encoded in ASCII base-85 representation. Example 3.3 shows the same stream without any encoding. (The stream's contents are explained in Section 3.7.1, "Content Streams," and the operators used there are further described in Chapter 5.)

Example 3.2

```
1 0 obj
   << /Length 534
       /Filter [/ASCII85Decode /LZWDecode]
   >>
stream
J..)6T`?p&<!J9%_[umg"B7/Z7KNXbN'S+,*Q/&"OLT'F
LIDK#!n`$"<Atdi`\Vn%b%)&'cA*VnK\CJY(sF>c!Jnl@
RM]WM;jjH6Gnc75idkL5]+cPZKEBPWdR>FF(kj1_R%W_d
&/jS!;iuad7h?[L–F$+]]0A3Ck*$I0KZ?;<)CJtqi65Xb
```

```
Vc3\n5ua:Q/=0$W<#N3U;H,MQKqfg1?:lUpR;6oN[C2E4
ZNr8Udn.'p+?#X+1>0Kuk$bCDF/(3fL5]Oq)^kJZ!C2H1
'TO]Rl?Q:&'<5&iP!$Rq;BXRecDN[IJB`,)o8XJOSJ9sD
S]hQ;Rj@!ND)bD_q&C\g:inYC%)&u#:u,M6Bm%lY!Kb1+
":aAa'S`ViJglLb8<W9k6Yl\\0McJQkDeLWdPN?9A'jX*
al>iG1p&i;eVoK&juJHs9%;Xomop"5KatWRT"JQ#qYuL,
JD?M$0QP)lKn06l1apKDC@\qJ4B!!(5m+j.7F790m(Vj8
8l8Q:_CZ(Gm1%X\N1&u!FKHMB~>
endstream
endobj
```

Example 3.3

```
1 0 obj
   << /Length 568 >>
stream
2 J
BT
/F1 12 Tf
0 Tc
0 Tw
72.5 712 TD
[(Unencoded streams can be read easily) 65 (, )] TJ
0 −14 TD
[(b) 20 (ut generally tak) 10 (e more space than \311)] TJ
T* (encoded streams.) Tj
0 −28 TD
[(Se) 25 (v) 15 (eral encoding methods are a) 20 (v) 25 (ailable in PDF) 80 (.)] TJ
0 −14 TD
(Some are used for compression and others simply) Tj
T* [(to represent binary data in an ) 55 (ASCII format.)] TJ
T* (Some of the compression encoding methods are \
suitable ) Tj
T* (for both data and images, while others are \
suitable only ) Tj
T* (for continuous−tone images.) Tj
ET
endstream
endobj
```

3.3.1 ASCIIHexDecode Filter

The **ASCIIHexDecode** filter decodes data that has been encoded in ASCII hexadecimal form. ASCII hexadecimal encoding and ASCII base-85 encoding (described in the next section) convert binary data, such as image data, to 7-bit ASCII characters. In general, ASCII base-85 encoding is preferred to ASCII hexadecimal encoding because it is more compact: it expands the data by a factor of 4:5, compared with 1:2 for ASCII hexadecimal encoding.

For each pair of ASCII hexadecimal digits (0–9 and A–F or a–f), the **ASCIIHex-Decode** filter produces one byte of binary data. All white-space characters (see Section 3.1, "Lexical Conventions") are ignored. A right angle bracket character (>) indicates EOD. Any other characters will cause an error. If the filter encounters the EOD marker after reading an odd number of hexadecimal digits, it will behave as if a 0 followed the last digit.

3.3.2 ASCII85Decode Filter

The **ASCII85Decode** filter decodes data that has been encoded in ASCII base-85 encoding and produces binary data. The following paragraphs describe the process for encoding binary data in ASCII base-85; the **ASCII85Decode** filter reverses this process.

The ASCII base-85 encoding uses the characters ! through u and the character z, with the 2-character sequence ~> as its EOD marker. The **ASCII85Decode** filter ignores all white-space characters (see Section 3.1, "Lexical Conventions"). Any other characters, and any character sequences that represent impossible combinations in the ASCII base-85 encoding, will cause an error.

Specifically, ASCII base-85 encoding produces 5 ASCII characters for every 4 bytes of binary data. Each group of 4 binary input bytes, $(b_1\ b_2\ b_3\ b_4)$, is converted to a group of 5 output bytes, $(c_1\ c_2\ c_3\ c_4\ c_5)$, using the relation

$$\begin{aligned}&(b_1 \times 256^3) + (b_2 \times 256^2) + (b_3 \times 256^1) + b_4 = \\ &\quad (c_1 \times 85^4) + (c_2 \times 85^3) + (c_3 \times 85^2) + (c_4 \times 85^1) + c_5\end{aligned}$$

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 the base-85 number are then converted to ASCII characters by adding 33 (the ASCII code for the character !) to each. The resulting encoded data contains only printable ASCII characters with codes in the range 33 (!) to 117 (u). As a special case, if all five digits are 0, they are represented by the character with code 122 (z) instead of by five exclamation points (!!!!!).

If the length of the binary data to be encoded is not a multiple of 4 bytes, the last, partial group of 4 is used to produce a last, partial group of 5 output characters. Given n (1, 2, or 3) bytes of binary data, the encoder first appends $4 - n$ zero bytes to make a complete group of 4. It then encodes this group in the usual way, but without applying the special z case. Finally, it writes only the first $n + 1$ characters of the resulting group of 5. These characters are immediately followed by the ~> EOD marker.

The following conditions (which never occur in a correctly encoded byte sequence) will cause errors during decoding:

- The value represented by a group of 5 characters is greater than $2^{32} - 1$.
- A z character occurs in the middle of a group.
- A final partial group contains only one character.

3.3.3 LZWDecode and FlateDecode Filters

The **LZWDecode** and (in PDF 1.2) **FlateDecode** filters have much in common and so are discussed together in this section. They decode data that has been encoded using the LZW or Flate data compression method, respectively.

- LZW (Lempel-Ziv-Welch) 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. Details on LZW encoding follow in the next section.
- The Flate method 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. It is fully defined in Internet RFCs 1950, *ZLIB Compressed Data Format Specification*, and 1951, *DEFLATE Compressed Data Format Specification* (see the Bibliography).

Both of these methods compress either binary data or ASCII text but (like all compression methods) always produce binary data, even if the original data was text.

The LZW and Flate compression methods can discover and exploit many patterns in the input data, whether the data is text or images. As described later, both filters support optional transformation by a *predictor function*, which improves the compression of sampled image data. Thanks to its cascaded adaptive Huffman coding, Flate-encoded output is usually much 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, both Flate and LZW encodings compress their input substantially. However, in the worst case (in which no pair of adjacent characters appears twice), Flate encoding *expands* its input by no more than 11 bytes or a factor of 1.003 (whichever is larger), plus the effects of algorithm tags added by PNG predictors. For LZW encoding, the best case (all zeros) provides a compression approaching 1365:1 for long files, but the worst-case expansion is at least a factor of 1.125, which can increase to nearly 1.5 in some implementations (plus the effects of PNG tags as with Flate encoding).

Details of LZW Encoding

Data encoded using the LZW compression method consists of a sequence of codes that are 9 to 12 bits long. Each code represents a single character of input data (0–255), a clear-table marker (256), an EOD marker (257), or a table entry representing a multiple-character sequence that has been encountered previously in the input (258 or greater).

Initially, the code length is 9 bits and the LZW table contains only entries for the 258 fixed codes. As encoding proceeds, entries are appended to the table, associating new codes with longer and longer sequences of input characters. The encoder and the decoder maintain identical copies of this table.

Whenever both the encoder and the 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. The first output code that is 10 bits long is the one following the creation of table entry 511, and similarly 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 a sequence already present in the table. For maximum compression, the encoder looks for 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 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 3.6.

TABLE 3.6 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
66	66	–	–
–	257 (EOD)	–	–

Codes are packed into a continuous bit stream, high-order bit first. 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 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 0C 85 01
```

To adapt to changing input sequences, the encoder may at any point issue a clear-table code, which causes both the encoder and the decoder to restart with initial tables and a 9-bit code length. 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.

Note: *The LZW compression method is the subject of U.S. patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Adobe Systems has licensed this patent for use in its Acrobat products; however, independent software vendors (ISVs) may be required to license this patent directly from Unisys to develop software that uses the LZW method to compress data in PDF files. For information on Unisys licensing policies, send e-mail to <lzw_info@unisys.com>, or visit the Unisys Web site at <http://www.unisys.com>.*

LZWDecode and FlateDecode Parameters

The **LZWDecode** and **FlateDecode** filters accept optional parameters to control the decoding process. Most of these parameters are related to techniques that reduce the size of compressed sampled images (rectangular arrays of color values, described in Section 4.8, "Images"). For example, image data frequently changes very little from sample to sample; subtracting the values of adjacent samples (a process called *differencing*), and encoding the differences rather than the raw sample values, can reduce the size of the output data. Furthermore, when the image data contains several color components (red-green-blue or cyan-magenta-yellow-black) per sample, taking the difference between the values of corresponding components in adjacent samples, rather than between different color components in the same sample, often reduces the output data size.

Table 3.7 shows the parameters that can optionally be specified for **LZWDecode** and **FlateDecode** filters. Except where otherwise noted, all values supplied to the decoding filter for any optional parameters must match those used when the data was encoded.

TABLE 3.7 Optional parameters for LZWDecode and FlateDecode filters

KEY	TYPE	VALUE
Predictor	integer	A code that selects the predictor algorithm, if any. If the value of this entry is 1, the filter assumes that the normal algorithm was used to encode the data, without prediction. If the value is greater than 1, the filter assumes that the data was differenced before being encoded, and **Predictor** selects the predictor algorithm. For more information regarding **Predictor** values greater than 1, see "LZW and Flate Predictor Functions," below. Default value: 1.
Colors	integer	*(Used only if* ***Predictor*** *is greater than 1)* The number of interleaved color components per sample. Valid values are 1 to 4 in PDF 1.2 or earlier, and 1 or greater in PDF 1.3 or later. Default value: 1.
BitsPerComponent	integer	*(Used only if* ***Predictor*** *is greater than 1)* The number of bits used to represent each color component in a sample. Valid values are 1, 2, 4, 8, and (in PDF 1.5) 16. Default value: 8.
Columns	integer	*(Used only if* ***Predictor*** *is greater than 1)* The number of samples in each row. Default value: 1.
EarlyChange	integer	*(****LZWDecode*** *only)* An indication of when to increase the code length. If the value of this entry is 0, code length increases are postponed as long as possible. If it is 1, they occur one code early. This parameter is included because LZW sample code distributed by some vendors increases the code length one code earlier than necessary. Default value: 1.

LZW and Flate Predictor Functions

LZW and Flate encoding compress more compactly if their input data is highly predictable. One way of increasing the predictability of many continuous-tone sampled images is to replace each sample with the difference between that sample and a *predictor function* applied to earlier neighboring samples. If the predictor function works well, the postprediction data will cluster toward 0.

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 its 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 Internet RFC 2083, *PNG (Portable Network Graphics) Specification* (see the Bibliography). The term *predictors* is used here instead of *filters* to avoid confusion. There are five basic PNG predictor algorithms (and a sixth that chooses the optimum predictor function separately for each row):

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 predictor algorithm to be used, if any, is indicated by the **Predictor** filter parameter (see Table 3.7), which can have any of the values listed in Table 3.8.

For **LZWDecode** and **FlateDecode**, a **Predictor** value greater than or equal to 10 merely indicates that a PNG predictor is in use; the specific predictor function used is explicitly encoded in the incoming data. The value of **Predictor** supplied by the decoding filter need not match the value used when the data was encoded if they are both greater than or equal to 10.

TABLE 3.8 Predictor values

VALUE	MEANING
1	No prediction (the default value)
2	TIFF Predictor 2
10	PNG prediction (on encoding, PNG None on all rows)
11	PNG prediction (on encoding, PNG Sub on all rows)
12	PNG prediction (on encoding, PNG Up on all rows)
13	PNG prediction (on encoding, PNG Average on all rows)
14	PNG prediction (on encoding, PNG Paeth on all rows)
15	PNG prediction (on encoding, PNG optimum)

The two groups of predictor functions have some commonalities. Both assume the following:

- Data is presented in order, from the top row to the bottom row and, within a row, from left to right.
- A row occupies a whole number of bytes, rounded up if necessary.
- Samples and their components are packed into bytes from high-order to low-order bits.
- All color components of samples outside the image (which are necessary for predictions near the boundaries) are 0.

The predictor function groups also differ in significant ways:

- The postprediction data for each PNG-predicted row begins with an explicit algorithm tag, so different rows can be predicted with different algorithms to improve compression. TIFF Predictor 2 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 component, taking into account the number of bits per component and 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 at the cost of somewhat worse compression.

3.3.4 RunLengthDecode Filter

The **RunLengthDecode** filter decodes data that has been encoded in a simple byte-oriented format based on run length. The encoded data 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 (1 to 128) bytes are copied literally during decompression. If *length* is in the range 129 to 255, the following single byte is to be copied 257 − *length* (2 to 128) times during decompression. A *length* value of 128 denotes EOD.

The compression achieved by run-length encoding depends on the input data. In the best case (all zeros), a compression of approximately 64:1 is achieved for long files. The worst case (the hexadecimal sequence 00 alternating with FF) results in an expansion of 127:128.

3.3.5 CCITTFaxDecode Filter

The **CCITTFaxDecode** filter decodes image data that has been encoded using either Group 3 or Group 4 CCITT facsimile (fax) encoding. CCITT encoding is designed to achieve efficient compression of monochrome (1 bit per pixel) image data at relatively low resolutions, and so is useful only for bitmap image data, not for color images, grayscale images, or general data.

The CCITT encoding standard is defined by the International Telecommunications Union (ITU), formerly known as the Comité Consultatif International Téléphonique et Télégraphique (International Coordinating Committee for Telephony and Telegraphy). The encoding algorithm is not described in detail here, but can be found in ITU Recommendations T.4 and T.6 (see the Bibliography). For historical reasons, we refer to these documents as the CCITT standard.

CCITT encoding is bit-oriented, not byte-oriented. This means that, in principle, encoded or decoded data might not end at a byte boundary. This problem is dealt with 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 PDF convention for sampled image data.
- Encoded data is ordinarily treated as a continuous, unbroken bit stream. The **EncodedByteAlign** parameter (described in Table 3.9) can be used to cause each encoded scan line to be filled to a byte boundary; although this is not prescribed by the CCITT standard and fax machines never do this, 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.

If the **CCITTFaxDecode** filter encounters improperly encoded source data, an error will occur. The filter will not perform any error correction or resynchronization, except as noted for the **DamagedRowsBeforeError** parameter in Table 3.9.

Table 3.9 lists the optional parameters that can be used to control the decoding. Except where noted otherwise, all values supplied to the decoding filter by any of these parameters must match those used when the data was encoded.

TABLE 3.9 Optional parameters for the CCITTFaxDecode filter

KEY	TYPE	VALUE
K	integer	A code identifying the encoding scheme 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 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	A flag indicating whether end-of-line bit patterns are required to be present in the encoding. The **CCITTFaxDecode** filter always accepts end-of-line bit patterns, but requires them only if **EndOfLine** is **true**. Default value: **false**.
EncodedByteAlign	boolean	A flag indicating whether the filter expects extra 0 bits before each encoded line so that the line begins on a byte boundary. If **true**, the filter skips over encoded bits to begin decoding each line at a byte boundary. If **false**, the filter does not expect extra bits in the encoded representation. Default value: **false**.
Columns	integer	The width of the image in pixels. If the value is not a multiple of 8, the filter adjusts the width of the unencoded image to the next multiple of 8, so that each line starts on a byte boundary. Default value: 1728.
Rows	integer	The height of the image in scan lines. If the value is 0 or absent, the image's height is not predetermined, and the encoded data must be terminated by an end-of-block bit pattern or by the end of the filter's data. Default value: 0.

KEY	TYPE	VALUE
EndOfBlock	boolean	A flag indicating whether the filter expects the encoded data to be terminated by an end-of-block pattern, overriding the **Rows** parameter. If **false**, the filter stops when it has decoded the number of lines indicated by **Rows** or when its data has been exhausted, whichever occurs first. The end-of-block pattern is the CCITT end-of-facsimile-block (EOFB) or return-to-control (RTC) appropriate for the **K** parameter. Default value: **true**.
BlackIs1	boolean	A flag indicating whether 1 bits are to be interpreted as black pixels and 0 bits as white pixels, the reverse of the normal PDF convention for image data. Default value: **false**.
DamagedRowsBeforeError	integer	The number of damaged rows of data to be tolerated before an error occurs. This entry 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 and 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.

The compression achieved using CCITT encoding depends on the data, as well as on the value of various optional parameters. For Group 3 one-dimensional encoding, in the best case (all zeros), each scan line compresses to 4 bytes, and the compression factor depends on the length of a scan line. If the scan line is 300 bytes long, a compression ratio of approximately 75:1 is achieved. The worst case, an image of alternating ones and zeros, produces an expansion of 2:9.

3.3.6 JBIG2Decode Filter

The **JBIG2Decode** filter *(PDF 1.4)* decodes monochrome (1 bit per pixel) image data that has been encoded using JBIG2 encoding. JBIG stands for the Joint Bi-Level Image Experts Group, a group within the International Organization for Standardization (ISO) that developed the format; JBIG2 is the second version of a standard originally released as JBIG1 and was approaching standards approval at the time of publication of this book.

JBIG2 encoding, which provides for both lossy and lossless compression, is useful only for monochrome images, not for color images, grayscale images, or general data. The algorithms used by the encoder, and the details of the format, are not described here; a working draft of the JBIG2 specification can be found through the Web site for the JBIG and JPEG (Joint Photographic Experts Group) committees at <http://www.jpeg.org>.

In general, JBIG2 provides considerably better compression than the existing CCITT standard (discussed in Section 3.3.5). The compression it achieves depends strongly on the nature of the image. Images of pages containing text in any language will compress particularly well, with typical compression ratios of 20:1 to 50:1 for a page full of text. The JBIG2 encoder builds a table of unique symbol bitmaps found in the image, and other symbols found later in the image are matched against the table. Matching symbols are replaced by an index into the table, and symbols that fail to match are added to the table. The table itself is compressed using other means. This results in high compression ratios for documents in which the same symbol is repeated often, as is typical for images created by scanning text pages. This method also results in high compression of "white space" in the image, which does not need to be encoded because it contains no symbols.

While best compression is achieved for images of text, the JBIG2 standard also includes algorithms for compressing regions of an image that contain dithered halftone images (for example, photographs).

The JBIG2 compression method can also be used for encoding multiple images into a single JBIG2 bit stream. Typically, these images will be scanned pages of a multiple-page document. Since a single table of symbol bitmaps is used to match symbols across multiple pages, this type of encoding can result in higher compression ratios than if each of the pages had been individually encoded using JBIG2.

In general, an image may be specified in PDF as either an *image XObject* or an *inline image* (as described in Section 4.8, "Images"); however, the **JBIG2Decode** filter can be applied only to image XObjects.

This filter addresses both single-page and multiple-page JBIG2 bit streams, by representing each JBIG2 "page" as a PDF image, as follows:

- The filter uses the embedded file organization of JBIG2. (The details of this and the other types of file organization are provided in an annex of the ISO specification.) The optional 2-byte combination (marker) mentioned in the specification is not used in PDF. JBIG2 bit streams in random-access organization should be converted to the embedded file organization. Bit streams in sequential organization need no reorganization, except for the mappings described below.

- The JBIG2 file header, end-of-page segments, and end-of-file segment are not used in PDF. These should be removed before the PDF objects described below are created.
- The image XObject to which the **JBIG2Decode** filter is applied contains all segments that are associated with the JBIG2 page represented by that image—that is, all segments whose segment page association field contains the page number of the JBIG2 page represented by the image. In the image XObject, however, the segment's page number should always be 1—that is, when each such segment is written to the XObject, the value of its segment page association field should be set to 1.
- If the bit stream contains global segments (segments whose segment page association field contains 0), these must be placed in a separate PDF stream, and the filter parameter listed in Table 3.10 should refer to that stream. The stream can be shared by multiple image XObjects whose JBIG2 encodings use the same global segments.

TABLE 3.10 Optional parameter for the JBIG2Decode filter

KEY	TYPE	VALUE
JBIG2Globals	stream	A stream containing the JBIG2 global (page 0) segments. Global segments must be placed in this stream even if only a single JBIG2 image XObject refers to it.

Example 3.4 shows an image that was compressed using the JBIG2 compression method and then encoded in ASCII hexadecimal representation. Since the JBIG2 bit stream contains global segments, these are placed in a separate PDF stream, as indicated by the **JBIG2Globals** filter parameter.

Example 3.4

```
5 0 obj
   << /Type /XObject
      /Subtype /Image
      /Width 52
      /Height 66
      /ColorSpace /DeviceGray
      /BitsPerComponent 1
      /Length 224
      /Filter [/ASCIIHexDecode /JBIG2Decode]
      /DecodeParms [null << /JBIG2Globals 6 0 R >>]
   >>
stream
000000013000010000001300000034000000420000000000
000000400000000000020620000100000001e000000340000
00420000000000000000020010000000231db51ce51ffac>
endstream
endobj

6 0 obj
   << /Length 126
      /Filter /ASCIIHexDecode
   >>
stream
000000000001000000000032000003fffdff02fefefe000000
01000000012ae225aea9a5a538b4d9999c5c8e56ef0f872
7f2b53d4e37ef795cc5506dffac>
endstream
endobj
```

The JBIG2 bit stream for this example is as follows:

```
97 4A 42 32 0D 0A 1A 0A 01 00 00 00 01 00 00 00 00 00 01 00 00 00 00 32
00 00 03 FF FD FF 02 FE FE FE 00 00 00 01 00 00 00 01 2A E2 25 AE A9 A5
A5 38 B4 D9 99 9C 5C 8E 56 EF 0F 87 27 F2 B5 3D 4E 37 EF 79 5C C5 50 6D
FF AC 00 00 00 01 30 00 01 00 00 00 13 00 00 00 34 00 00 00 42 00 00 00
00 00 00 00 00 40 00 00 00 00 00 02 06 20 00 01 00 00 00 1E 00 00 00 34
00 00 00 42 00 00 00 00 00 00 00 00 02 00 10 00 00 00 02 31 DB 51 CE 51
FF AC 00 00 00 03 31 00 01 00 00 00 00 00 00 00 04 33 01 00 00 00 00
```

This bit stream is made up of the parts listed below (in the order listed).

1. The JBIG2 file header

   ```
   97 4A 42 32 0D 0A 1A 0A 01 00 00 00 01
   ```

   Since the JBIG2 file header is not used in PDF, this header is not placed in the JBIG2 stream object, and is discarded.

2. The first JBIG2 segment (segment 0)—in this case, the symbol dictionary segment

   ```
   00 00 00 00 00 01 00 00 00 00 32 00 00 03 FF FD FF 02 FE FE FE 00 00 0
   0
   01 00 00 00 01 2A E2 25 AE A9 A5 A5 38 B4 D9 99 9C 5C 8E 56 EF 0F 87
   27 F2 B5 3D 4E 37 EF 79 5C C5 50 6D FF AC
   ```

   This is a global segment (segment page association = 0) and so is placed in the **JBIG2Globals** stream.

3. The page information segment

   ```
   00 00 00 01 30 00 01 00 00 00 13 00 00 00 34 00 00 00 42 00 00 00 00
   00 00 00 00 40 00 00
   ```

   and the immediate text region segment

   ```
   00 00 00 02 06 20 00 01 00 00 00 1E 00 00 00 34 00 00 00 42 00 00 00
   00 00 00 00 00 02 00 10 00 00 00 02 31 DB 51 CE 51 FF AC
   ```

   These two segments constitute the contents of the JBIG2 page, and are placed in the PDF XObject representing this image.

4. The end-of-page segment

   ```
   00 00 00 03 31 00 01 00 00 00 00
   ```

   and the end-of-file segment

   ```
   00 00 00 04 33 01 00 00 00 00
   ```

   Since these are not used in PDF, they are discarded.

The resulting PDF image object, then, contains the page information segment and the immediate text region segment, and refers to a **JBIG2Globals** stream that contains the symbol dictionary segment.

3.3.7 DCTDecode Filter

The **DCTDecode** filter decodes grayscale or color image data that has been encoded in the JPEG baseline format. (JPEG stands for the Joint Photographic Experts Group, a group within the International Organization for Standardization that developed the format; DCT stands for discrete cosine transform, the primary technique used in the encoding.)

JPEG encoding is a lossy compression method, designed specifically for compression of sampled continuous-tone images and not for general data compression. Data to be encoded using JPEG consists of a stream of image samples, each consisting of one, two, three, or four color components. The color component values for a particular sample must appear consecutively. Each component value occupies an 8-bit byte.

During encoding, several parameters control the algorithm and the information loss. The values of these parameters, which include the dimensions of the image and the number of components per sample, are entirely under the control of the encoder and are stored in the encoded data. **DCTDecode** generally obtains the parameter values it requires directly from the encoded data. However, in one instance, the parameter might not be present in the encoded data but must be specified in the filter parameter dictionary; see Table 3.11.

The details of the encoding algorithm are not presented here but can be found in the ISO specification and in *JPEG: Still Image Data Compression Standard*, by Pennebaker and Mitchell (see the Bibliography). Briefly, the JPEG algorithm breaks an image up into blocks 8 samples wide by 8 high. Each color component in an image is treated separately. A two-dimensional DCT is performed on each block. This operation produces 64 coefficients, which are then quantized. Each coefficient may be quantized with a different step size. It is this quantization that results in the loss of information in the JPEG algorithm. The quantized coefficients are then compressed.

TABLE 3.11 Optional parameter for the DCTDecode filter

KEY	TYPE	VALUE
ColorTransform	integer	A code specifying the transformation to be performed on the sample values: 0 No transformation. 1 If the image has three color components, transform *RGB* values to *YUV* before encoding and from *YUV* to *RGB* after decoding. If the image has four components, transform *CMYK* values to *YUVK* before encoding and from *YUVK* to *CMYK* after decoding. This option is ignored if the image has one or two color components. ***Note:*** *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 the image has three components 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 encoding algorithm 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.

The encoding algorithm can reduce the information loss by making the step size in the quantization smaller at the expense of reducing the amount of compression achieved by the algorithm. The compression achieved by the JPEG algorithm depends on the image being compressed and the amount of loss that is acceptable. In general, a compression of 15:1 can be achieved without perceptible loss of information, and 30:1 compression causes little impairment of the image.

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.

The JPEG filter implementation in Adobe Acrobat products does not support features of the JPEG standard that are irrelevant to images. In addition, certain choices have been made regarding reserved marker codes and other optional features of the standard. For details, see Adobe Technical Note #5116, *Supporting the DCT Filters in PostScript Level 2*.

In addition to the baseline JPEG format, beginning with PDF 1.3 the **DCTDecode** filter supports the progressive JPEG extension. This extension does not add any entries to the **DCTDecode** parameter dictionary; the distinction between baseline and progressive JPEG is represented in the encoded data.

Note: *There is no benefit to using progressive JPEG for stream data that is embedded in a PDF file. Decoding progressive JPEG is slower and consumes more memory than baseline JPEG. The purpose of this feature is to enable a stream to refer to an external file whose data happens to be already encoded in progressive JPEG. (See also implementation note 11 in Appendix H.)*

3.3.8 JPXDecode Filter

The **JPXDecode** filter *(PDF 1.5)* decodes data that has been encoded using the JPEG2000 compression method, an international standard for the compression and packaging of image data. JPEG2000 defines a wavelet-based method for image compression that gives somewhat better size reduction than other methods such as regular JPEG or CCITT. Although the filter can reproduce samples that are losslessly compressed, it is recommended only for use with images and not for general data compression.

In PDF, this filter can be applied only to image XObjects, and not to inline images (see Section 4.8, “Images”). It is suitable both for images that have a single color component and for those with multiple color components. The color components in an image may have different numbers of bits per sample. Any value from 1 to 38 is allowed.

From a single JPEG2000 data stream, multiple versions of an image may be decoded. These different versions form progressions along four degrees of freedom: sampling resolution, color depth, band, and location. For example, with a resolution progression, a thumbnail version of the image may be decoded from the data, followed by a sequence of other versions of the image each with approximately 4 times as many samples (twice the width times twice the height) as the previous one. The last version is the full resolution image.

Viewing and printing applications may gain performance benefits by using the resolution progression. If the full-resolution image is densely sampled, the application may be able to select and decode only the data making up a lower-resolution version, thereby spending less time decoding. Fewer bytes need be processed, a particular benefit when viewing files over the Web. The tiling structure of the image may also provide benefits, if only certain areas of an image need to be displayed or printed.

***Note:** Information on these progressions is encoded in the data; there are no decode parameters needed to describe them. The decoder deals with any progressions it encounters to deliver the correct image data; progressions that are of no interest may simply have performance consequences.*

The JPEG2000 specifications define two widely used formats, JP2 and JPX, for packaging the compressed image data; JP2 is a subset of JPX. These packagings contain all the information needed to properly interpret the image data, including the color space, bits per component and image dimensions. In other words, they are complete descriptions of images (as opposed to image data that require outside parameters for correct interpretation). The **JPXDecode** filter expects to read a full JPX file structure—either internal to the PDF file or as an external file.

To promote interoperability, the specifications define a subset of JPX called *JPX baseline* (of which JP2 is also a subset). The complete details of the baseline set of JPX features are contained in ISO/IEC 15444-2, *Information Technology—JPEG 2000 Image Coding System: Extensions* (see the Bibliography). See also <http://www.jpeg.org/JPEG2000.html>.

Data used in PDF image XObjects should be limited to the JPX baseline set of features, except for enumerated color space 19 (CIEJab). In addition, enumerated color space 12 (CMYK), which is part of JPX but not JPX baseline, is supported in PDF.

A JPX file describes a collection of *channels* that are present in the image data. A channel may have one of three types:

- An *ordinary* channel contains values that, when decoded, become samples for a specified color component.
- An *opacity* channel provides samples that are to be interpreted as raw opacity information.

- A *premultiplied opacity* channel provides samples that have been multiplied into the color samples of those channels with which it is associated.

Opacity and premultiplied opacity channels are associated with specific color channels. There is never more than one opacity channel (of either type) associated with a given color channel. For example, it is possible for one opacity channel to apply to the red samples and another to apply to the green and blue color channels of an RGB image.

***Note:** The method by which the opacity information is to be used is explicitly not specified, although one possible method shows a normal blending mode.*

In addition to using opacity channels for describing transparency, JPX files also have the ability to specify chroma-key transparency. A single color is specified by giving an array of values, one value for each color channel. Any image location which matches this color is considered to be completely transparent.

In a JPX file, the color space for an image may be one of the following:

- A predefined color space, chosen from a list of *enumerated color spaces*. (Two of these are actually families of spaces and parameters are included.)
- A "restricted ICC profile". (These are the only sorts of ICC profiles which are allowed in JP2 files.)
- An input ICC profile of any sort defined by ICC-1.
- A *vendor-defined* color space.

More than one color space may be specified for an image, with each space being tagged with a precedence and an approximation value that indicates how well it represents the "correct" color space. In addition, the image's color space may serve as the foundation for a palette of colors that are selected using samples coming from the image's data channels: the equivalent of an **Indexed** color space in PDF.

There are other features in the JPX format beyond describing a simple image. These include provisions for describing layering and giving instructions on composition, specifying simple animation, and including generic XML metadata (along with JPEG2000-specific schemas for such data). It is recommended, but not required, that relevant metadata be replicated in the image dictionary's **Metadata** stream in XMP format (see Section 10.2.2, "Metadata Streams).

When using the **JPXDecode** filter with image XObjects, there are changes to and constraints on some entries in the image dictionary (see Section 4.8.4, “Image Dictionaries” for details on these entries):

- **Width** and **Height** must match the corresponding width and height values in the JPEG2000 data.
- **ColorSpace** is optional, since JPEG2000 data contain color space specifications. If it is present, it determines how the image samples are interpreted, and the color space specifications in the JPEG2000 data are ignored. The number of color channels contained in the JPEG2000 data must match the number of components in the color space; the producer of the PDF must ensure that the samples are consistent with the color space being used.

  Any color space other than **Pattern** may be specified. If an **Indexed** color space is used, it is subject to the PDF limit of 256 colors. (The analogous concept in the JPEG2000 color specifications is a *palette color space*, which has a limit of 1024 colors.) If the color space does not match one of JPX’s enumerated color spaces (for example, if it has 2 color components or more than 4) it can be specified as a vendor color space in the JPX data.

  If **ColorSpace** is not present in the image dictionary, the color space information in the JPEG2000 data is used. Viewer applications must support the JPX baseline set of enumerated color spaces; they are also responsible for dealing with the interaction between the color spaces and the bit depth of samples.

  If multiple color space specifications are given in the JPEG2000 data, a rendering application should attempt to use the one with the highest precedence and best approximation value. If the color space is given by an unsupported ICC profile, the next lower color space, in terms of precedence and approximation value, is used. If no supported color space is found, the color space used should be **DeviceGray**, **DeviceRGB** or **DeviceCMYK** depending on the number of color channels in the JPEG2000 data.
- **SMaskInData** specifies whether soft-mask information packaged with the image samples should be used (see “Soft-Mask Images” on page 512); if it is, the **SMask** entry is not needed. If **SMaskInData** is non-zero, there must be only one opacity channel in the JPEG2000 data and it must apply to all color channels.
- **Decode** is ignored, except in the case where the image is being treated as a mask; that is, when **ImageMask** is true. In this case, the JPEG2000 data must provide a single color channel with 1-bit samples.

3.3.9 Crypt Filter

The **Crypt** filter *(PDF 1.5)* allows the document-level security handler (see Section 3.5, "Encryption") to determine which algorithms should be used to decrypt the input data. The **Name** parameter in the decode parameters dictionary for this filter (see Table 3.12) specifies which of the named crypt filters in the document (see Section 3.5.4, "Crypt Filters") should be used.

TABLE 3.12 Optional parameters for Crypt filters

KEY	TYPE	VALUE
Type	name	*(Optional)* If present, must be **CryptFilterDecodeParms** for a crypt filter decode parameter dictionary.
Name	name	*(Optional)* The name of the crypt filter that is to be used to decrypt this stream. The name must correspond to an entry in the **CF** entry of the encryption dictionary (see Table 3.18) or one of the standard crypt filters (see Table 3.23). Default value: **Identity**.

In addition, the decode parameters dictionary may include entries that are private to the security handler. Security handlers may use information from both the crypt filter decode parameters dictionary and the crypt filter dictionaries (see Table 3.22) when decrypting data or providing a key to decrypt data.

Note: *When adding private data to the decode parameters dictionary, security handlers should name these entries in conformance with the PDF name registry (see Appendix E, "PDF Name Registry").*

3.4 File Structure

The preceding sections describe the syntax of individual objects. This section describes how objects are organized in a PDF file for efficient random access and incremental update. A canonical PDF file initially consists of four elements (see Figure 3.2):

- A one-line *header* identifying the version of the PDF specification to which the file conforms
- A *body* containing the objects that make up the document contained in the file

- A *cross-reference table* containing information about the indirect objects in the file
- A *trailer* giving the location of the cross-reference table and of certain special objects within the body of the file

This initial structure may be modified by later updates, which append additional elements to the end of the file; see Section 3.4.5, "Incremental Updates," for details.

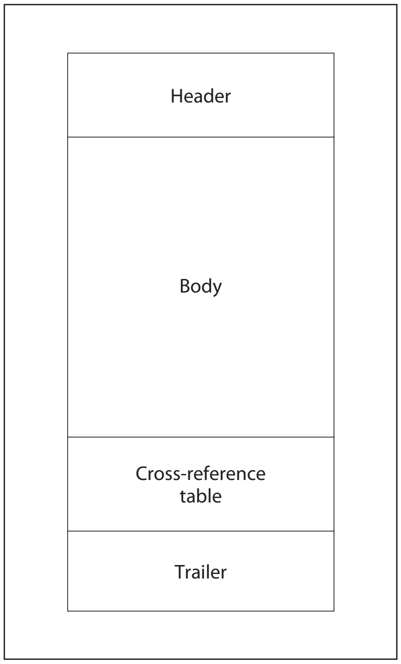


FIGURE 3.2 *Initial structure of a PDF file*

As a matter of convention, the tokens in a PDF file are arranged into lines; see Section 3.1, "Lexical Conventions." Each line is terminated by an end-of-line (EOL) marker, which may be a carriage return (character code 13), a line feed (character code 10), or both. PDF files with binary data may have arbitrarily long lines. However, to increase compatibility with other applications that process PDF files, lines that are not part of stream object data are limited to no more than 255 characters, with one exception: beginning with PDF 1.3, an exception is made to the restriction on line length in the case of the **Contents** string of a signature dictionary (see Section 8.7, "Digital Signatures"). See also implementation note 12 in Appendix H.

The rules described here are sufficient to produce a well-formed PDF file. However, there are some additional rules for organizing a PDF file to enable efficient incremental access to a document's components in a network environment. This form of organization, called *Linearized PDF*, is described in Appendix F.

3.4.1 File Header

The first line of a PDF file is a *header* identifying the version of the PDF specification to which the file conforms. For a file conforming to PDF version 1.5, the header should be

```
%PDF−1.5
```

However, since any file conforming to an earlier version of PDF also conforms to version 1.4, an application that processes PDF 1.4 can also accept files with any of the following headers:

```
%PDF−1.0
%PDF−1.1
%PDF−1.2
%PDF−1.3
```

(See also implementation notes 13 and 14 in Appendix H.)

In PDF 1.4, the version in the file header can be overridden by the **Version** entry in the document's catalog dictionary (located via the **Root** entry in the file's trailer, as described in Section 3.4.4, "File Trailer"). This enables a PDF producer application to update the version using an incremental update (see Section 3.4.5, "Incremental Updates").

Under some conditions, a viewer application may be able to process PDF files conforming to a later version than it was designed to accept. New PDF features are often introduced in such a way that they can safely be ignored by a viewer that does not understand them (see Section H.1, "PDF Version Numbers").

Note: *If a PDF file contains binary data, as most do (see Section 3.1, "Lexical Conventions"), it is recommended that the header line be immediately followed by a* *comment line containing at least four binary characters—that is, characters whose codes are 128 or greater. This will ensure proper behavior of file transfer applications that inspect data near the beginning of a file to determine whether to treat the file's contents as text or as binary.*

3.4.2 File Body

The *body* of a PDF file consists of a sequence of indirect objects representing the contents of a document. The objects, which are of the basic types described in Section 3.2, "Objects," represent components of the document such as fonts, pages, and sampled images. Starting with PDF 1.5, the body can also contain object streams, each of which in turn contains a sequence of indirect objects; see Section 3.4.6, "Object Streams"

3.4.3 Cross-Reference Table

The *cross-reference table* contains information that permits random access to indirect objects within the file, so that the entire file need not be read to locate any particular object. The table contains a one-line entry for each indirect object, specifying the location of that object within the body of the file. (Starting with PDF 1.5, some or all of the cross-reference information may alternatively be contained in cross-reference streams; see Section 3.4.7, "Cross-Reference Streams").

The cross-reference table is the only part of a PDF file with a fixed format; this permits entries in the table to be accessed randomly. The table comprises one or more *cross-reference sections*. Initially, the entire table consists of a single section (or two sections if the file is linearized; see Appendix F); one additional section is added each time the file is updated (see Section 3.4.5, "Incremental Updates").

Each cross-reference section begins with a line containing the keyword **xref**. Following this line are one or more *cross-reference subsections*, which may appear in any order. The subsection structure is useful for incremental updates, since it allows a new cross-reference section to be added to the PDF file, containing entries only for objects that have been added or deleted. For a file that has never been updated, the cross-reference section contains only one subsection, whose object numbering begins at 0.

Each cross-reference subsection contains entries for a contiguous range of object numbers. The subsection begins with a line containing two numbers, separated by a space: the object number of the first object in this subsection and the number of entries in the subsection. For example, the line

```
28 5
```

introduces a subsection containing five objects, numbered consecutively from 28 to 32.

Note: *A given object number must not have an entry in more than one subsection within a single section. However, see implementation note 15 in Appendix H.*

Following this line are the cross-reference entries themselves, one per line. Each entry is exactly 20 bytes long, including the end-of-line marker. There are two kinds of cross-reference entry: one for objects that are in use and another for objects that have been deleted and so are free. Both types of entry have similar basic formats, distinguished by the keyword **n** (for an in-use entry) or **f** (for a free entry). The format of an in-use entry is as follows:

nnnnnnnnnn ggggg **n** *eol*

where

nnnnnnnnnn is a 10-digit byte offset

ggggg is a 5-digit generation number

n is a literal keyword identifying this as an in-use entry

eol is a 2-character end-of-line sequence

The byte offset is a 10-digit number, padded with leading zeros if necessary, giving the number of bytes from the beginning of the file to the beginning of the object. It is separated from the generation number by a single space. The generation number is a 5-digit number, also padded with leading zeros if necessary. Following the generation number is a single space, the keyword **n**, and then a 2-character end-of-line sequence. If the file's end-of-line marker is a single character (either a carriage return or a line feed), it is preceded by a single space; if the marker is 2 characters (both a carriage return and a line feed), it is not preceded by a space. Thus the overall length of the entry is always exactly 20 bytes.

The cross-reference entry for a free object has essentially the same format, except that the keyword is **f** instead of **n** and the interpretation of the first item is different:

nnnnnnnnnn ggggg **f** *eol*

where

nnnnnnnnnn is the 10-digit object number of the next free object

ggggg is a 5-digit generation number

f is a literal keyword identifying this as a free entry

eol is a 2-character end-of-line sequence

The free entries in the cross-reference table form a linked list, with each free entry containing the object number of the next. The first entry in the table (object number 0) is always free and has a generation number of 65,535; it is the head of the linked list of free objects. The last free entry (the tail of the linked list) links back to object number 0. (In addition, the table may contain other free entries that link back to object number 0 and have a generation number of 65,535, even though these entries are not in the linked list itself.)

Except for object number 0, all objects in the cross-reference table initially have generation numbers of 0. When an indirect object is deleted, its cross-reference entry is marked free and it is added to the linked list of free entries. The entry's generation number is incremented by 1 to indicate the generation number to be used the next time an object with that object number is created. Thus each time the entry is reused, it is given a new generation number. The maximum generation number is 65,535; when a cross-reference entry reaches this value, it will never be reused.

The cross-reference table (comprising the original cross-reference section and all update sections) must contain one entry for each object number from 0 to the maximum object number used in the file, even if one or more of the object numbers in this range do not actually occur in the file. See

Example 3.5 shows a cross-reference section consisting of a single subsection with six entries: four that are in use (objects number 1, 2, 4, and 5) and two that are free (objects number 0 and 3). Object number 3 has been deleted, and the next object created with that object number will be given a generation number of 7.

Example 3.5

```
xref
0  6
0000000003  65535  f
0000000017  00000  n
0000000081  00000  n
0000000000  00007  f
0000000331  00000  n
0000000409  00000  n
```

Example 3.6 shows a cross-reference section with four subsections, containing a total of five entries. The first subsection contains one entry, for object number 0, which is free. The second subsection contains one entry, for object number 3, which is in use. The third subsection contains two entries, for objects number 23 and 24, both of which are in use. Object number 23 has been reused, as can be seen from the fact that it has a generation number of 2. The fourth subsection contains one entry, for object number 30, which is in use.

Example 3.6

```
xref
0  1
0000000000  65535  f
3  1
0000025325  00000  n
23  2
0000025518  00002  n
0000025635  00000  n
30  1
0000025777  00000  n
```

See Section G.6, "Updating Example," for a more extensive example of the structure of a PDF file that has been updated several times.

3.4.4 File Trailer

The *trailer* of a PDF file enables an application reading the file to quickly find the cross-reference table and certain special objects. Applications should read a PDF file from its end. The last line of the file contains only the end-of-file marker, %%EOF. (See implementation note 17 in Appendix H.) The two preceding lines contain the keyword **startxref** and the byte offset from the beginning of the file to the beginning of the **xref** keyword in the last cross-reference section. The **startxref** line is preceded by the *trailer dictionary*, consisting of the keyword **trailer** followed by a series of key-value pairs enclosed in double angle brackets (<<…>>). Thus the trailer has the following overall structure:

```
trailer
   << key1  value1
      key2  value2
      …
      keyn  valuen
   >>
startxref
Byte_offset_of_last_cross-reference_section
%%EOF
```

Table 3.13 lists the contents of the trailer dictionary.

TABLE 3.13 Entries in the file trailer dictionary

KEY	TYPE	VALUE
Size	integer	*(Required; must not be an indirect reference)* The total number of entries in the file's cross-reference table, as defined by the combination of the original section and all update sections. Equivalently, this value is 1 greater than the highest object number used in the file. ***Note:*** *Any object in a cross-reference section whose number is greater than this value is ignored and considered missing.*
Prev	integer	*(Present only if the file has more than one cross-reference section; must not be an indirect reference)* The byte offset from the beginning of the file to the beginning of the previous cross-reference section.
Root	dictionary	*(Required; must be an indirect reference)* The catalog dictionary for the PDF document contained in the file (see Section 3.6.1, "Document Catalog").
Encrypt	dictionary	*(Required if document is encrypted; PDF 1.1)* The document's encryption dictionary (see Section 3.5, "Encryption").
Info	dictionary	*(Optional; must be an indirect reference)* The document's information dictionary (see Section 10.2.1, "Document Information Dictionary").
ID	array	*(Optional; PDF 1.1)* An array of two strings constituting a file identifier (see Section 10.3, "File Identifiers") for the file.

Example 3.7 shows an example trailer for a file that has never been updated (as indicated by the absence of a **Prev** entry in the trailer dictionary).

Example 3.7

```
trailer
    << /Size 22
        /Root 2 0 R
        /Info 1 0 R
        /ID [ <81b14aafa313db63dbd6f981e49f94f4>
                <81b14aafa313db63dbd6f981e49f94f4>
            ]
    >>
startxref
18799
%%EOF
```

3.4.5 Incremental Updates

The contents of a PDF file can be updated incrementally, without rewriting the entire file. Changes are appended to the end of the file, leaving its original contents intact. The main advantage to updating a file in this way (as discussed in Section 2.2.7, "Incremental Update") is that it enables small changes to a large document to be saved quickly. Other advantages include the following:

- In some cases, incremental updating is the only way to save changes to a document. An accepted practice for minimizing the risk of data loss when saving a document is to write it to a new file and then rename the new file to replace the old one; however, in certain contexts, such as when editing a document across an HTTP connection or using OLE embedding (a Windows-specific technology), it is not possible to overwrite the contents of the original file in this manner. Incremental updates can be used to save changes to documents in these contexts.
- Once a document has been signed (see Section 2.2.6, "Security"), all changes made to the document must be saved using incremental updates, since altering any existing bytes in the file will invalidate existing signatures.

In an incremental update, any new or changed objects are appended to the file, a cross-reference section is added, and a new trailer is inserted. The resulting file has the structure shown in Figure 3.3. A complete example of an updated file is shown in Section G.6, "Updating Example."

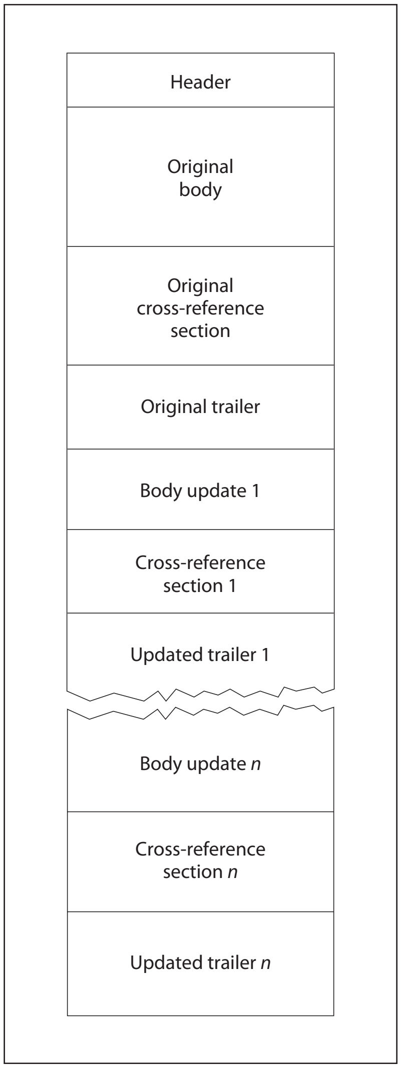


FIGURE 3.3 *Structure of an updated PDF file*

The cross-reference section added when a file is updated contains entries only for objects that have been changed, replaced, or deleted. Deleted objects are left unchanged in the file, but are marked as deleted via their cross-reference entries. The added trailer contains all the entries (perhaps modified) from the previous trailer, as well as a **Prev** entry giving the location of the previous cross-reference section (see Table 3.13 on page 73). As shown in Figure 3.3, a file that has been updated several times contains several trailers; note that each trailer is terminated by its own end-of-file (%%EOF) marker.

Because updates are appended to PDF files, it is possible to end up with several copies of an object with the same object identifier (object number and generation number). This can occur, for example, if a text annotation (see Section 8.4, "Annotations") is changed several times, with the file being saved between changes. Because the text annotation object is not deleted, it retains the same object number and generation number as before. An updated copy of the object is included in the new update section added to the file; the update's cross-reference section includes a byte offset to this new copy of the object, overriding the old byte offset contained in the original cross-reference section. When a viewer application reads the file, it must build its cross-reference information in such a way that the most recent copy of each object is the one accessed in the file.

In versions of PDF prior to 1.4, it was not possible to use an incremental update to alter the version of PDF to which the document conforms, since the version was specified only in the header at the beginning of the file (see Section 3.4.1, "File Header"). In PDF 1.4, it is possible for a **Version** entry in the document's catalog dictionary (see Section 3.6.1, "Document Catalog") to override the version specified in the header; this enables the version to be altered using an incremental update.

3.4.6 Object Streams

PDF 1.5 introduces a new kind of stream, an *object stream*, which contains a sequence of PDF objects. The purpose of object streams is to allow a greater number of PDF objects to be compressed, thereby allowing a substantial reduction in the size of PDF files. The objects in the stream are referred to as *compressed objects*. (This term is used whether or not the stream is actually encoded with a compression filter.)

Any PDF object can appear in an object stream, with the following exceptions:

- Stream objects
- Objects with a generation number other than zero
- A document's encryption dictionary (see Section 3.5, "Encryption")
- An object representing the value of the **Length** entry in an object stream dictionary

Note: *In addition, in linearized files (see Appendix F, "Linearized PDF"), the document catalog, the linearization dictionary and page objects may not appear in an object stream.*

Indirect references to objects inside object streams use the normal syntax: for example, `14 0 R`. Access to these objects requires a new way of storing cross-reference information; see Section 3.4.7, "Cross-Reference Streams." Although a PDF 1.5 viewer is needed in order to use compressed objects, there is a way to store them in a PDF file that is compatible with a PDF 1.4 viewer, which simply ignores the objects; see "Compatibility with PDF 1.4" on page 85.

In addition to the standard keys for streams shown in Table 3.4, the stream dictionary describing an object stream contains the following entries:

TABLE 3.14 Additional entries specific to an object stream dictionary

KEY	TYPE	DESCRIPTION
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **ObjStm** for an object stream.
N	integer	*(Required)* The number of compressed objects in the stream.
First	integer	*(Required)* The byte offset (in the decoded stream) of the first compressed object.
Extends	stream	*(Optional)* A reference to an object stream, of which the current object stream is considered an extension. Both streams are considered part of a *collection* of object streams (see below). A given collection consists of a set of streams whose **Extends** links form a directed acyclic graph.

The creator of a PDF file has flexibility in determining which objects, if any, to store in object streams. For example, it can be useful to store objects having common characteristics together, such as "fonts on page 1," or "Comments for draft #3." These objects may be said to form a *collection*.

However, the number of objects in an individual object stream should be limited, to avoid a degradation of performance, such as would occur when downloading and decompressing a large object stream to access a single compressed object (see implementation note 18 in Appendix H). This may require a group of object streams to be linked as a collection, which can be done by means of the **Extends** entry in the object stream dictionary.

Extends can also be used when a collection is being updated to include new objects. Rather than redefine the original object stream, which would require duplicating the stream data, the new objects can be stored in a new object stream. This is particularly important when adding an update section to a document.

The stream data in an object stream consists of the following:

- **N** pairs of integers, where the first integer in each pair represents the object number of a compressed object and the second integer represents the byte offset of that object, relative to the first one. The offsets must be in increasing order, but there is no restriction on the order of object numbers.

  Note: The byte offset in the decoded stream of the first object is the value of the **First** entry.

- The **N** objects stored consecutively. Only the object values are stored in the stream; the **obj** and **endobj** keywords are not used. A compressed dictionary or array may contain indirect references.

  Note: *It is illegal for a compressed object to consist of nothing but an indirect reference; for example, 3 0 R.*

  By contrast, dictionaries and arrays in content streams (Section 3.7.1) may not contain indirect references. In an encrypted file, strings occurring anywhere in an object stream must not be separately encrypted, since the entire object stream will be encrypted.

  Note: *The data for the first object is not required to immediately follow the last byte offset. Future extensions may place additional information between those two points in the stream.*

An object stream itself, like any stream, is an indirect object, and there must be an entry for it in a cross-reference table or cross-reference stream (see Section 3.4.7, "Cross-Reference Streams"), although there might not be any references to it (of the form 243 0 R).

The generation number of an object stream and of any compressed object is implicitly zero. If either an object stream or a compressed object is deleted and the object number is freed, that object number can be reused only for an ordinary (uncompressed) object other than an object stream. When new object streams and compressed objects are created, they must always be assigned new object numbers, not old ones taken from the free list.

Example 3.8 shows three objects (two fonts and a font descriptor) as they would be represented in a PDF 1.4 or earlier file, along with a cross-reference table. In Example 3.9, the same objects are stored in an object stream in a PDF 1.5 file, using a cross-reference stream.

Example 3.8

```
11 0 obj
    << /Type /Font
        /Subtype /TrueType
        ...other entries...
        /FontDescriptor 12 0 R
    >>
endobj

12 0 obj
    << /Type /FontDescriptor
        /Ascent 891
        ...other entries...
        /FontFile2 22 0 R
    >>
endobj

13 0 obj
    << /Type /Font
        /Subtype /Type0
        ...other entries...
        /ToUnicode 10 0 R
    >>
endobj

...
```

```
xref
0 32
0000000000 65535 f
...
0000001434 00000 n        % Cross-reference entry for object 11
0000001735 00000 n        % Cross-reference entry for object 12
0000002155 00000 n        % Cross-reference entry for object 13
...
trailer
    << /Size 32
        /Root ...
    >>
```

In Example 3.9, the cross-reference stream (see Section 3.4.7, "Cross-Reference Streams") contains entries for the fonts (objects 11 and 13) and the descriptor (object 12), which are compressed objects in an object stream. The first field of these entries is the entry type (2), the second field is the number of the object stream (15), and the third field is the position within the sequence of objects in the object stream (0, 1, and 2). The cross-reference stream also contains a type 1 entry for the object stream itself.

***Note:** For readability, the object stream has been shown unencoded. In a real PDF 1.5 file, Flate encoding would typically be used to gain the benefits of compression.*

Example 3.9

```
15 0 obj                  % The object stream
    << /Type /ObjStm
        /Length 1856
        /N 3              % The number of objects in the stream
        /First 24         % The byte offset of the first object
    >>
stream
    % The object numbers and offsets of the objects, relative to the first
    11 0 12 547 13 665
    << /Type /Font
        /Subtype /TrueType
        ...other keys...
        /FontDescriptor 12 0 R
    >>
    << /Type /FontDescriptor
        /Ascent 891
        ...other keys...
        /FontFile2 22 0 R
    >>
    << /Type /Font
        /Subtype /Type0
        ...other keys...
        /ToUnicode 10 0 R
    >>
...
endstream
endobj

99 0 obj                % The cross-reference stream
    << /Type /XRef
        /Index [0 32]           % This section has one subsection with 32 objects
        /W [1 2 2]              % Each entry has 3 fields: 1, 2 and 2 bytes in width,
                                % respectively
        /Filter /ASCIIHexDecode % For readability in this example
        /Size 32
        ...
    >>
stream
    00 0000 FFFF        % "0 65535 f" in a cross-reference table
    ...
    02 000F 0000        % The entry for object 11, the first font
    02 000F 0001        % The entry for object 12, the font descriptor
    02 000F 0002        % The entry for object 13, the second font
    ...
    01 BA5E 0000        % The entry for object 15, the object stream
    ...
endstream
endobj

startxref
    54321               % The offset of "99 0 obj"
%%EOF
```

3.4.7 Cross-Reference Streams

Beginning with PDF 1.5, cross-reference information may be stored in a *cross-reference stream*, instead of a cross-reference table. Cross-reference streams provide the following advantages:

- A more compact representation of cross-reference information.

- The ability to access compressed objects that are stored in object streams (see Section 3.4.6, “Object Streams”), and to allow new cross-reference entry types to be added in the future.

Cross-reference streams are stream objects (see Section 3.2.7, “Stream Objects”), and contain a dictionary and a data stream. Each cross-reference stream contains the information equivalent to the cross-reference table (see Section 3.4.3, “Cross-Reference Table”) and trailer (see Section 3.4.4, “File Trailer”) for one cross-reference section. The trailer dictionary entries are stored in the stream dictionary, and the cross-reference table entries are stored as the stream data, as shown in the following example:

Example 3.10

```
... objects ...

12 0 obj            % Cross-reference stream
   << /Type /XRef   % Cross-reference stream dictionary
      /Size ...
      /Root ...
   >>
stream
   ...              % Stream data containing cross-reference information
endstream
endobj

... more objects ...

startxref
byte_offset_of_cross-reference_stream   % Points to object 12
%%EOF
```

Note that the value following the **startxref** keyword is now the offset of the cross-reference stream rather than an **xref** keyword. For files that use cross-reference streams entirely (that is, PDF 1.5 files that are not hybrid-reference files; see “Compatibility with PDF 1.4” on page 85), the keywords **xref** and **trailer** are no longer used. Therefore, with the exception of the “**startxref** *address* **%%EOF**” segment and comments, a PDF 1.5 file is entirely a sequence of objects.

Note: *The use of object streams and cross-reference streams is permitted in linearized PDF, with minor modifications to the specification (see Section F.2, “Linearized PDF Document Structure”).*

Cross-Reference Stream Dictionary

Cross-reference streams contain the entries shown in Table 3.15, in addition to the entries common to all streams (Table 3.4) and trailer dictionaries (Table 3.13). Since some of the information in the cross-reference stream is needed by the viewer application to construct the index that allows indirect references to be resolved, the entries in cross-reference streams are subject to the following restrictions:

- The value of all entries shown in Table 3.15 must be direct objects; indirect references are not permitted. For arrays (the **Index** and **W** entries), all their elements must be direct objects as well. If the stream is encoded, the **Filter** and **DecodeParms** entries in Table 3.4 must also be direct objects.

  Note: *Other cross-reference stream entries not listed in Table 3.15 may be indirect; in fact, some (such as* **Root** *in Table 3.13) are required to be indirect.*

- The cross-reference stream itself must not be encrypted, nor may any strings appearing in the cross-reference stream dictionary. It must not have a **Filter** entry that specifies a **Crypt** filter (see 3.3.9, "Crypt Filter").

TABLE 3.15 Additional entries specific to a cross-reference stream dictionary

KEY	TYPE	DESCRIPTION
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **XRef** for a cross-reference stream.
Size	integer	*(Required)* The number one greater than the highest object number used in this section, or in any section for which this is an update. It is equivalent to the **Size** entry in a trailer dictionary.
Index	array	*(Optional)* An array containing a pair of integers for each subsection in this section. Each pair consists of: • The first object number in the subsection. • The number of entries in the subsection. The array is sorted in ascending order by object number. There may be no overlaps between subsections; an object number may have at most one entry in a section. Default value: [0 **Size**].

KEY	TYPE	DESCRIPTION
Prev	integer	*(Present only if the file has more than one cross-reference stream; not meaningful in hybrid-reference files; see "Compatibility with PDF 1.4" on page 85)* The byte offset from the beginning of the file to the beginning of the previous cross-reference stream. It has the same function as the **Prev** entry in the trailer dictionary (Table 3.13).
W	array	*(Required)* An array of integers representing the size of the fields in a single cross-reference entry. Table 3.16 describes the types of entries and their fields. For PDF 1.5, **W** always contains three integers; the value of each integer is the number of bytes (in the decoded stream) of the corresponding field. For example, [1 2 1] means that the fields are one byte, two bytes, and one byte, respectively. A value of zero for an element in the **W** array indicates that the corresponding field is not present in the stream, and the default value is used, if there is one. If the first element is zero, the type field is not present, and it defaults to type 1. The sum of the items is the total length of each entry, and can be used with the **Index** array to determine the starting position of each subsection. ***Note:*** *Different cross-reference streams in a PDF file may use different values for* ***W****.*

Cross-Reference Stream Data

Each entry in a cross-reference stream has one or more fields, the first of which designates the entry's type (see Table 3.16). In PDF 1.5, only types 0, 1 and 2 are allowed. Any other value is interpreted as a reference to the null object, thus permitting new entry types to be defined in the future.

The fields are written in increasing order of field number; the length of each field is determined by the corresponding value in the **W** entry (see Table 3.15). Fields requiring more than one byte are stored with the high-order byte first.

TABLE 3.16 Entries in a cross-reference stream

TYPE	FIELD	DESCRIPTION
0	1	The type of this entry, which must be 0. Type 0 entries define the linked list of free objects (corresponding to **f** entries in a cross-reference table).
	2	The object number of the next free object.
	3	The generation number to use if this object number is used again.

TYPE	FIELD	DESCRIPTION
1	1	The type of this entry, which must be 1. Type 1 entries define objects that are in used but are not compressed (corresponding to **n** entries in a cross-reference table).
	2	The byte offset of the object, starting from the beginning of the file.
	3	The generation number of the object. Default value: 0.
2	1	The type of this entry, which must be 2. Type 2 entries define compressed objects.
	2	The object number of the object stream in which this object is stored. (The generation number of the object stream is implicitly 0).
	3	The index of this object within the object stream.

Like any stream, a cross-reference stream is an indirect object; therefore, there must be an entry for it in the file in either a cross-reference stream (usually itself), or in a cross-reference table (in hybrid-reference files; see "Compatibility with PDF 1.4" on page 85).

Compatibility with PDF 1.4

PDF 1.4 and earlier viewers cannot access objects that are referenced by cross-reference streams; if a file uses cross-reference streams exclusively, it cannot be opened by a PDF 1.4 viewer.

However, it is possible to construct a file that is readable by a PDF 1.4 viewer, containing objects referenced by standard cross-reference tables, which also contains objects in object streams that are referenced by cross-reference streams. Such files are referred to as *hybrid-reference* files.

In these files, the trailer dictionary may contain, in addition to the entry for trailers shown in Table 3.13, an additional entry, as shown in Table 3.17. This entry is ignored by PDF 1.4 viewers, which therefore have no access to entries in the cross-reference stream it refers to.

TABLE 3.17 Additional entries in a hybrid-reference file's trailer dictionary

KEY	TYPE	VALUE
XRefStm	integer	*(Optional)* The byte offset from the beginning of the file of a cross-reference stream.

The **Size** entry of the trailer must be large enough to include all objects, including those defined in the cross-reference stream referenced by the **XRefStm** entry. However, in a main cross-reference section, entries for all objects numbered 0 through **Size** - 1 must be present to allow random access (see Section 3.4.3, "Cross-Reference Table"). Therefore, the **XRefStm** entry cannot be used in the trailer dictionary of the main cross-reference section, but only in an update cross-reference section.

When a hybrid-reference file is opened in a PDF 1.5 viewer, objects with entries in cross-reference streams are no longer hidden. When a PDF 1.5 viewer searches for an object, if an entry is not found in any given standard cross-reference section, the search proceeds to a cross-reference stream specified by the **XRefStm** entry, before looking in the previous cross-reference section (the **Prev** entry in the trailer).

Hidden objects, then, have two cross-reference entries. One is in the cross-reference stream. The other is a free entry in some previous section, typically the one referenced by the **Prev** entry. Since a PDF 1.5 viewer looks in the cross-reference stream first, finds the object there and ignores the free entry in the previous section. A PDF 1.4 viewer ignores the cross-reference stream and looks in the previous section, where it finds the free entry. The free entry must have a next-generation number of 65535, so that the object number is never reused.

There are limitations on which objects in a hybrid-reference file can be hidden without making the file appear invalid to PDF 1.4 and earlier viewers. In particular, the root of the PDF file, the document catalog (see Section 3.6.1, "Document Catalog"), must not be hidden, nor any object that is *visible from the root*. Such objects can be determined by starting from the root and working recursively:

- In any dictionary that is visible, direct objects are visible. The value of any required key-value pair is visible.
- In any array that is visible, every element is visible.

- Resource dictionaries in content streams are visible. Although a resource dictionary is not required, strictly speaking, the content stream to which it is attached is assumed to contain references to the resources.

In general, those objects that may be hidden are optional objects that are specified by indirect references. In a PDF 1.5 viewer, those references can be resolved by processing the cross-reference streams. In a PDF 1.4 viewer, the objects appear to be free, and the references are treated as references to the null object.

For example, the **Outlines** entry in the catalog dictionary is optional. Therefore, its value may be an indirect reference to a hidden object. A PDF 1.4 viewer treats it as a reference to the null object, which is equivalent to having omitted the entry entirely; a PDF 1.5 viewer recognizes it. On the other hand, if the value of the **Outlines** entry is an indirect reference to a visible object, the entire outline tree must then be visible, because nodes in the outline tree contain required pointers to other nodes.

Following this logic, items that must be visible include the entire page tree, fonts, font descriptors, and width tables. Objects that may be hidden in a hybrid-reference file include the structure tree, the outline tree, article threads, annotations, destinations, Web Capture information, and page labels,.

Example 3.11 shows a hybrid-reference file, containing a main cross-reference section and an update cross-reference section with an **XRefStm** entry that points to a cross-reference stream (object 11), which in turn has references to an object stream (object 2).

In this example, the catalog (object 1) contains an indirect reference (`3 0 R`) to the root of the structure tree. The search for the object starts at the update cross-reference table, which has no objects in it. The search proceeds depending on the version of the viewer application:

- In a PDF 1.4 viewer, the search continues by following the **Prev** pointer to the main cross-reference table. That table defines object 3 as a free object, which is treated as the **null** object. Therefore, the entry is considered missing, and the document has no structure tree.
- In a PDF 1.5 viewer, the search continues by following the **XRefStm** pointer to the cross-reference stream (object 11). It defines object 3 as a compressed object, stored at index 0 in the object stream (`2 0 obj`). Therefore, the document has a structure tree.

Example 3.11

```
1 0 obj                          % The document root, at offset 23.
    << /Type /Catalog
        /StructTreeRoot 3 0 R
        …
    >>
endobj

12 0 obj
…
endobj
…
99 0 obj
…
endobj

xref                             % The main xref section, at offset 2664
0 100                            % This subsection has entries for objects 0 - 99.
0000000002 65535 f               % Entry for object 0
0000000023 00000 n               % Entry for object 1, the root
0000000003 65535 f               % Entry for object 2 (object stream), marked free in this table
0000000004 65535 f               % Entry for object 3, marked free in this table
0000000005 65535 f               % …
0000000006 65535 f
0000000007 65535 f
0000000008 65535 f
0000000009 65535 f
0000000010 65535 f
0000000011 65535 f
0000000000 65535 f               % Entry for object 11 (xref stream), marked free in this table.
0000000045 00000 n               % Entry for object 12, in use.
0000000179 00000 n               % Entry for object 13, in use.
…
0000002201 00000 n               % Entry for object 99, in use.
trailer
    << /Size 100
        /Root 1 0 R
        /ID …
    >>
startxref
    2264                         % Offset of the main xref section
%%EOF
```

```
2 0 obj                         % The object stream, at offset 3722
    << /Length ...
        /N 8                    % This stream contains 8 objects.
        /First 47               % The stream-offset of the first object
    >>
stream
    3 0 4 50 5 72 …             % The numbers and stream-offsets of the 8 objects
    << /Type /StructTreeRoot            % This is object 3.
        /K 4 0 R
        /RoleMap 5 0 R
        /ClassMap 6 0 R
        /ParentTree 7 0 R
        /ParentTreeNextKey 8
    >>
    << /S /Workbook             % This is object 4 (K value from StructTreeRoot).
        /P 8 0 R
        /K 9 0 R
    >>
    << /Workbook /Div           % This is object 5 (RoleMap).
        /Worksheet /Sect
        /TextBox /Figure
        /Shape /Figure
    >>
    …                           % Objects 6 through 10 are defined here.
endstream
endobj

11 0 obj                        % The cross-reference stream, at offset 4899
    << /Type /XRefStm
        /Index [2 9]            % This stream contains entries for objects 2 through 10.
        /W [1 2 1]              % The byte-widths of each field
        /Filter /ASCIIHexDecode
        /Length …
    >>
stream
    01 0E8A 0                   % Entry for object 2 (0x0E8A = 3722)
    02 0002 00                  % Entry for object 3 (in object stream 2, index 0)
    02 0002 01                  % Entry for object 4 (in object stream 2, index 1)
    02 0002 02                  % …
    02 0002 03
    02 0002 04
    02 0002 05
    02 0002 06
    02 0002 07              % Entry for object 9 (in object stream 2, index 7)
    01 1323 0               % Entry for object 10 (0x1323 = 4899)
endstream
endobj

xref                            % The update xref section, at offset 5640
0 0                             % There are no entries in this section.
trailer
    << /Size 100
        /Prev 2664              % Offset of previous xref section
        /XRefStm 4899
        /Root 1 0 R
        /ID …
    >>
startxref
5640
%%EOF
```

Additional comments on the example:

- The object stream is unencoded, and the cross-reference stream uses an ASCII hexadecimal encoding, for purposes of clarity. In practice, both streams would be Flate-encoded. Also, the comments shown in the cross-reference table in the above example are for illustrative purposes; PDF comments are not legal in a cross-reference table.
- The hidden objects, 2 through 11, are numbered consecutively. In practice, there is no such requirement, nor is there a requirement that free items in a cross-reference table be linked in ascending order until the end.
- The update cross-reference table contains no entries. This is not a requirement, but it is reasonable. A PDF creator that uses the hybrid-reference format will create the main cross-reference table, the update cross-reference table, and the cross-reference stream at the same time. Objects 12 and 13, for example, are not compressed. They might have entries in the update table. Since objects 2 and 11, the object stream and the cross-reference stream, are not compressed, they might also be defined in the update table. Since they're part of the "hidden" section, however, it makes sense to define them in the cross-reference stream.
- The update cross-reference section must appear at the end of the file, but otherwise, there are no ordering restrictions on any of the objects or on the main cross-reference section. However, a file that uses both the hybrid-reference for-

mat and the linearized format has ordering requirements (see Appendix F, “Linearized PDF”).

3.5 Encryption

A PDF document can be *encrypted (PDF 1.1)* to protect its contents from unauthorized access. Encryption applies to all strings and streams in the document’s PDF file, but not to other object types such as integers and boolean values, which are used primarily to convey information about the document’s structure rather than its content. Leaving these values unencrypted allows random access to the objects within a document, while encrypting the strings and streams protects the document’s substantive contents.

Note: *When a PDF stream object (see Section 3.2.7, “Stream Objects”) refers to an external file, the stream’s contents are not encrypted, since they are not part of the PDF file itself. However, if the contents of the stream are embedded within the PDF file (see Section 3.10.3, “Embedded File Streams”), they are encrypted like any other stream in the file. In PDF 1.5, embedded files may be encrypted in an otherwise unencrypted document (see Section 3.5.4, “Crypt Filters”).*

Encryption-related information is stored in a document’s *encryption dictionary*, which is the value of the **Encrypt** entry in the document’s trailer dictionary (see Table 3.13 on page 73). The absence of this entry from the trailer dictionary means that the document is not encrypted. The entries shown in Table 3.18 are common to all encryption dictionaries.

The encryption dictionary’s **Filter** entry identifies the file’s *security handler*, a software module that implements various aspects of the encryption process and controls access to the contents of the encrypted document. PDF specifies a standard password-based security handler that all viewer applications are expected to support, but applications may optionally substitute security handlers of their own.

The **SubFilter** entry specifies the syntax of the encryption dictionary contents. It allows interoperability between handlers; that is, a document may be decrypted by a handler other than the preferred one (the **Filter** entry), if they both support the format specified by **SubFilter**.

The **V** entry, in specifying which algorithm to use, determines the length or lengths allowed for the encryption key, on which the encryption (and decryption) of data in a PDF file is based. If more than one length is allowed (that is, for **V** values 2 and 3), the **Length** entry specifies the exact length of the encryption key. In PDF 1.5, a value of 4 for **V** is permitted, allowing the security handler to use its own encryption and decryption algorithms and to specify *crypt filters* to use on specific streams (see Section 3.5.4, "Crypt Filters").

The remaining contents of the encryption dictionary are determined by the security handler, and may vary from one handler to another. Those for the standard security handler are described in Section 3.5.2, "Standard Security Handler." Those for public-key security handlers are described in Section 3.5.3, "Public-Key Security Handlers."

TABLE 3.18 Entries common to all encryption dictionaries

KEY	TYPE	VALUE
Filter	name	*(Required)* The name of the preferred *security handler* for this document; typically it is the name of the security handler that was used to encrypt the document. If **SubFilter** is not present, only this security handler should be used when opening the document. If it is present, viewer applications are free to use any security handler that implements the format specified by **SubFilter**. **Standard** is the name of the built-in password-based security handler. Names for other security handlers can be registered using the procedure described in Appendix E. ***Note:*** *The definition of this entry has been clarified since the previous version of this document.*
SubFilter	name	*(Optional; PDF 1.3)* A name that completely specifies the format and interpretation of the contents of the encryption dictionary. It is needed in order to allow security handlers other than the one specified by **Filter** to decrypt the document. If it is absent, other security handlers will not be able to decrypt the document. ***Note:*** *This entry was introduced in PDF 1.3 to support the use of public-key cryptography in PDF files (see Section 3.5.3, "Public-Key Security Handlers"); however, it was not incorporated into the* PDF Reference *until the fourth edition (PDF 1.5).*

KEY	TYPE	VALUE
V	number	*(Optional but strongly recommended)* A code specifying the algorithm to be used in encrypting and decrypting the document: 0 An algorithm that is undocumented and no longer supported, and whose use is strongly discouraged. 1 Algorithm 3.1 on page 94, with an encryption key length of 40 bits; see below. 2 *(PDF 1.4)* Algorithm 3.1 on page 94, but allowing encryption key lengths greater than 40 bits. 3 *(PDF 1.4)* An unpublished algorithm allowing encryption key lengths ranging from 40 to 128 bits. (This algorithm is unpublished as an export requirement of the U.S. Department of Commerce.) 4 *(PDF 1.5)* The security handler defines the use of encryption and decryption in the document, using the rules specified by the **CF**, **StmF**, and **StrF** entries. The default value if this entry is omitted is 0, but a value of 1 or greater is strongly recommended. (See implementation note 19 in Appendix H.)
Length	integer	*(Optional; PDF 1.4; only if **V** is 2 or 3)* The length of the encryption key, in bits. The value must be a multiple of 8, in the range 40 to 128. Default value: 40.
CF	dictionary	*(Optional; meaningful only when the value of **V** is 4; PDF 1.5)* A dictionary whose keys are crypt filter names and whose values are the corresponding crypt filter dictionaries (see Table 3.22). Every crypt filter used in the document must have an entry in this dictionary, except for the standard crypt filter names (see Table 3.23). ***Note:** An attempt to redefine any of the standard names in Table 3.23 is ignored.*
StmF	name	*(Optional; meaningful only when the value of **V** is 4; PDF 1.5)* The name of the crypt filter that is used by default when encrypting streams; it must correspond to a key in the **CF** dictionary or a standard crypt filter name specified in Table 3.23. All streams in the document, except for cross-reference streams (see Section 3.4.7, "Cross-Reference Streams") or those that have a **Crypt** entry in their **Filter** array (see Table 3.5), are decrypted by the security handler, using this crypt filter. Default value: **Identity**.
StrF	name	*(Optional; meaningful only when the value of **V** is 4; PDF 1.5)* The name of the crypt filter that is used when decrypting all strings in the document; it must correspond to a key in the **CF** dictionary or a standard crypt filter name specified in Table 3.23. Default value: **Identity**.

Unlike strings within the body of the document, those in the encryption dictionary must be direct objects and are *not* encrypted by the usual methods (the algorithm specified by the **V** entry. The security handler itself is responsible for encrypting and decrypting strings in the encryption dictionary, using whatever encryption algorithm it chooses.

Note: *Document creators have two choices if the encryption methods and syntax provided by PDF are not sufficient for their needs: they can provide an alternate security handler or they can encrypt whole PDF documents themselves, not making use of PDF security.*

3.5.1 General Encryption Algorithm

PDF's standard encryption methods use the MD5 message-digest algorithm (described in Internet RFC 1321, *The MD5 Message-Digest Algorithm*; see the Bibliography) and a proprietary encryption algorithm known as RC4. RC4 is a symmetric stream cipher: the same algorithm is used for both encryption and decryption, and the algorithm does not change the length of the data.

Note: *RC4 is a copyrighted, proprietary algorithm of RSA Security, Inc. Adobe Systems has licensed this algorithm for use in its Acrobat products. Independent software vendors may be required to license RC4 in order to develop software that encrypts or decrypts PDF documents. For further information, visit the RSA Web site at <http://www.rsasecurity.com> or send e-mail to <products@rsasecurity.com>.*

The encryption of data in a PDF file is based on the use of an *encryption key* computed by the security handler. Different security handlers can compute the encryption key in a variety of ways, more or less cryptographically secure. Regardless of how the key is computed, its use in the encryption of data is always the same (see Algorithm 3.1). Because the RC4 algorithm is symmetric, this same sequence of steps can be used both to encrypt and to decrypt data.

Algorithm 3.1 ***Encryption of data using an encryption key***

1. Obtain the object number and generation number from the object identifier of the string or stream to be encrypted (see Section 3.2.9, "Indirect Objects"). If the string is a direct object, use the identifier of the indirect object containing it.
2. Treating the object number and generation number as binary integers, extend the original n-byte encryption key to $n + 5$ bytes by appending the low-order 3 bytes

of the object number and the low-order 2 bytes of the generation number in that order, low-order byte first. (*n* is 5 unless the value of **V** in the encryption dictionary is greater than 1, in which case *n* is the value of **Length** divided by 8.)

3. Initialize the MD5 hash function and pass the result of step 2 as input to this function.
4. Use the first (*n* + 5) bytes, up to a maximum of 16, of the output from the MD5 hash as the key for the RC4 encryption function, along with the string or stream data to be encrypted. The output is the encrypted data to be stored in the PDF file.

Stream data is encrypted after applying all stream encoding filters, and is decrypted before applying any stream decoding filters; the number of bytes to be encrypted or decrypted is given by the **Length** entry in the stream dictionary. Decryption of strings (other than those in the encryption dictionary) is done after escape-sequence processing and hexadecimal decoding as appropriate to the string representation described in Section 3.2.3, "String Objects."

3.5.2 Standard Security Handler

PDF's standard security handler allows *access permissions* and up to two passwords to be specified for a document: an *owner password* and a *user password*. An application's decision to encrypt a document is based on whether the user creating the document specifies any passwords or access restrictions (for example, in a security settings dialog that the user can invoke before saving the PDF file); if so, the document is encrypted, and the permissions and information required to validate the passwords are stored in the encryption dictionary. (An application may also create an encrypted document without any user interaction, if it has some other source of information about what passwords and permissions to use.)

If a user attempts to open an encrypted document that has a user password, the viewer application should prompt for a password. Correctly supplying either password allows the user to open the document, decrypt it, and display it on the screen. If the document does not have a user password, no password is requested; the viewer application can simply open, decrypt, and display the document. Whether additional operations are allowed on a decrypted document depends on which password (if any) was supplied when the document was opened and on any access restrictions that were specified when the document was created:

- Opening the document with the correct owner password (assuming it is not the same as the user password) allows full (owner) access to the document. This unlimited access includes the ability to change the document's passwords and access permissions.

- Opening the document with the correct user password (or opening a document that does not have a user password) allows additional operations to be performed according to the user access permissions specified in the document's encryption dictionary.

Access permissions are specified in the form of flags corresponding to the various operations, and the set of operations to which they correspond depends in turn on the security handler's revision number (also stored in the encryption dictionary). If the revision number is 2 or greater, the operations to which user access can be controlled are as follows:

- Modifying the document's contents
- Copying or otherwise extracting text and graphics from the document, including extraction for accessibility purposes (that is, to make the contents of the document accessible through assistive technologies such as screen readers or Braille output devices; see Section 10.8, "Accessibility Support")
- Adding or modifying text annotations (see "Text Annotations" on page 574) and interactive form fields (Section 8.6, "Interactive Forms")
- Printing the document

If the security handler's revision number is 3 or greater, user access to the following operations can be controlled more selectively:

- Filling in forms (that is, filling in existing interactive form fields) and signing the document (which amounts to filling in existing signature fields, a type of interactive form field)
- Assembling the document: inserting, rotating, or deleting pages and creating navigation elements such as bookmarks or thumbnail images (see Section 8.2, "Document-Level Navigation")
- Printing to a representation from which a faithful digital copy of the PDF content could be generated. Disallowing such printing may result in degradation of output quality (a feature implemented as "Print As Image" in Acrobat)

In addition, revision 3 enables the extraction of text and graphics (in support of accessibility to disabled users or for other purposes) to be controlled separately.

Beginning with revision 4, the standard security handler supports crypt filters (see Section 3.5.4, "Crypt Filters"). The support is limited to the **Identity** crypt filter (see Table 3.23) and crypt filters named **StdCF** whose dictionaries contain a **CFM** value of **V2** and an **AuthEvent** value of **DocOpen**.

Note: *Once the document has been opened and decrypted successfully, the viewer application has access to the entire contents of the document. There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. It is up to the implementors of PDF viewer applications to respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.*

Note: *PDF 1.5 introduces a new set of access permissions that do not require the document to be encrypted; see Section 8.7.3, "Permissions."*

Standard Encryption Dictionary

Table 3.19 shows the encryption dictionary entries for the standard security handler (in addition to those in Table 3.18 on page 92).

TABLE 3.19 Additional encryption dictionary entries for the standard security handler

KEY	TYPE	VALUE
R	number	*(Required)* A number specifying which revision of the standard security handler should be used to interpret this dictionary. The revision number should be • 2 if the document is encrypted with a **V** value less than 2 (see Table 3.18) and does not have any of the access permissions set (via the **P** entry, below) that are designated "Revision 3" in Table 3.20. • 3 if the document is encrypted with a **V** value of 2 or 3, or has any "Revision 3" access permissions set. • 4 if the document is encrypted with a **V** value of 4.
O	string	*(Required)* A 32-byte string, based on both the owner and user passwords, that is used in computing the encryption key and in determining whether a valid owner password was entered. For more information, see "Encryption Key Algorithm" on page 99 and "Password Algorithms" on page 100.
U	string	*(Required)* A 32-byte string, based on the user password, that is used in determining whether to prompt the user for a password and, if so, whether a valid user or owner password was entered. For more information, see "Password Algorithms" on page 100.

KEY	TYPE	VALUE
P	integer	*(Required)* A set of flags specifying which operations are permitted when the document is opened with user access (see Table 3.20).

The values of the **O** and **U** entries in this dictionary are used to determine whether a password entered when the document is opened is the correct owner password, user password, or neither.

The value of the **P** entry is an unsigned 32-bit integer containing a set of flags specifying which access permissions should be granted when the document is opened with user access. Table 3.20 shows the meanings of these flags. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order); a 1 bit in any position enables the corresponding access permission. Which bits are meaningful, and in some cases how they are interpreted, depends on the security handler's revision number (specified in the encryption dictionary's **R** entry).

Note: *PDF integer objects in fact are represented internally in signed twos-complement form. Since all the reserved high-order flag bits in the encryption dictionary's* ***P*** *value are required to be 1, the value must be specified as a negative integer. For example, assuming revision 2 of the security handler, the value -44 allows printing and copying but disallows modifying the contents and annotations.*

TABLE 3.20 User access permissions

BIT POSITION	MEANING
1–2	Reserved; must be 0.
3	*(Revision 2)* Print the document. *(Revision 3)* Print the document (possibly not at the highest quality level, depending on whether bit 12 is also set).
4	Modify the contents of the document by operations other than those controlled by bits 6, 9, and 11.
5	*(Revision 2)* Copy or otherwise extract text and graphics from the document, including extracting text and graphics (in support of accessibility to disabled users or for other purposes). *(Revision 3)* Copy or otherwise extract text and graphics from the document by operations other than that controlled by bit 10.

BIT POSITION	MEANING
6	Add or modify text annotations, fill in interactive form fields, and, if bit 4 is also set, create or modify interactive form fields (including signature fields).
7–8	Reserved; must be 1.
9	*(Revision 3 only)* Fill in existing interactive form fields (including signature fields), even if bit 6 is clear.
10	*(Revision 3 only)* Extract text and graphics (in support of accessibility to disabled users or for other purposes).
11	*(Revision 3 only)* Assemble the document (insert, rotate, or delete pages and create bookmarks or thumbnail images), even if bit 4 is clear.
12	*(Revision 3 only)* Print the document to a representation from which a faithful digital copy of the PDF content could be generated. When this bit is clear (and bit 3 is set), printing is limited to a low-level representation of the appearance, possibly of degraded quality. (See implementation note 20 in Appendix H.)
13–32	*(Revision 3 only)* Reserved; must be 1.

Encryption Key Algorithm

As noted earlier, one function of a security handler is to generate an encryption key for use in encrypting and decrypting the contents of a document. Given a password string, the standard security handler computes an encryption key as shown in Algorithm 3.2.

Algorithm 3.2 ***Computing an encryption key***

1. Pad or truncate the password string to exactly 32 bytes. If the password string is more than 32 bytes long, use only its first 32 bytes; if it is less than 32 bytes long, pad it by appending the required number of additional bytes from the beginning of the following padding string:

```
< 28 BF 4E 5E 4E 75 8A 41 64 00 4E 56 FF FA 01 08
  2E 2E 00 B6 D0 68 3E 80 2F 0C A9 FE 64 53 69 7A >
```

   That is, if the password string is *n* bytes long, append the first 32 – *n* bytes of the padding string to the end of the password string. If the password string is empty (zero-length), meaning there is no user password, substitute the entire padding string in its place.

2. Initialize the MD5 hash function and pass the result of step 1 as input to this function.
3. Pass the value of the encryption dictionary's **O** entry to the MD5 hash function. (Algorithm 3.3 shows how the **O** value is computed.)
4. Treat the value of the **P** entry as an unsigned 4-byte integer and pass these bytes to the MD5 hash function, low-order byte first.
5. Pass the first element of the file's file identifier array (the value of the **ID** entry in the document's trailer dictionary; see Table 3.13 on page 73) to the MD5 hash function.
6. *(Revision 3 only)* If document metadata is not being encrypted, pass 4 bytes with the value 0xFFFFFFFF to the MD5 hash function.
7. Finish the hash.
8. *(Revision 3 only)* Do the following 50 times: Take the output from the previous MD5 hash and pass it as input into a new MD5 hash.
9. Set the encryption key to the first *n* bytes of the output from the final MD5 hash, where *n* is always 5 for revision 2 but for revision 3 depends on the value of the encryption dictionary's **Length** entry.

This algorithm, when applied to the user password string, produces the encryption key used to encrypt or decrypt string and stream data according to Algorithm 3.1 on page 94. Parts of this algorithm are also used in the algorithms described below.

Password Algorithms

In addition to the encryption key, the standard security handler must provide the contents of the encryption dictionary (Tables 3.18 on page 92 and 3.19 on page 97). The values of the **Filter**, **V**, **Length**, **R**, and **P** entries are straightforward, but the computation of the **O** (owner password) and **U** (user password) entries requires further explanation. Algorithms 3.3 through 3.5 show how the values of the owner password and user password entries are computed (with separate versions of the latter for revisions 2 and 3 of the security handler).

Algorithm 3.3 ***Computing the encryption dictionary's O (owner password) value***

1. Pad or truncate the owner password string as described in step 1 of Algorithm 3.2. If there is no owner password, use the user password instead. (See implementation note 21 in Appendix H.)
2. Initialize the MD5 hash function and pass the result of step 1 as input to this function.
3. *(Revision 3 only)* Do the following 50 times: Take the output from the previous MD5 hash and pass it as input into a new MD5 hash.
4. Create an RC4 encryption key using the first *n* bytes of the output from the final MD5 hash, where *n* is always 5 for revision 2 but for revision 3 depends on the value of the encryption dictionary's **Length** entry.
5. Pad or truncate the user password string as described in step 1 of Algorithm 3.2.
6. Encrypt the result of step 5, using an RC4 encryption function with the encryption key obtained in step 4.
7. *(Revision 3 only)* Do the following 19 times: Take the output from the previous invocation of the RC4 function and pass it as input to a new invocation of the function; use an encryption key generated by taking each byte of the encryption key obtained in step 4 and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 1 to 19).
8. Store the output from the final invocation of the RC4 function as the value of the **O** entry in the encryption dictionary.

Algorithm 3.4 ***Computing the encryption dictionary's U (user password) value (Revision 2)***

1. Create an encryption key based on the user password string, as described in Algorithm 3.2.
2. Encrypt the 32-byte padding string shown in step 1 of Algorithm 3.2, using an RC4 encryption function with the encryption key from the preceding step.
3. Store the result of step 2 as the value of the **U** entry in the encryption dictionary.

Algorithm 3.5 ***Computing the encryption dictionary's U (user password) value (Revision 3)***

1. Create an encryption key based on the user password string, as described in Algorithm 3.2.
2. Initialize the MD5 hash function and pass the 32-byte padding string shown in step 1 of Algorithm 3.2 as input to this function.

3. Pass the first element of the file's file identifier array (the value of the **ID** entry in the document's trailer dictionary; see Table 3.13 on page 73) to the hash function and finish the hash.
4. Encrypt the 16-byte result of the hash, using an RC4 encryption function with the encryption key from step 1.
5. Do the following 19 times: Take the output from the previous invocation of the RC4 function and pass it as input to a new invocation of the function; use an encryption key generated by taking each byte of the original encryption key (obtained in step 1) and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 1 to 19).
6. Append 16 bytes of arbitrary padding to the output from the final invocation of the RC4 function and store the 32-byte result as the value of the **U** entry in the encryption dictionary.

The standard security handler uses Algorithms 3.6 and 3.7 to determine whether a supplied password string is the correct user or owner password. Note too that Algorithm 3.6 can be used to determine whether a document's user password is the empty string, and therefore whether to suppress prompting for a password when the document is opened.

Algorithm 3.6 ***Authenticating the user password***

1. Perform all but the last step of Algorithm 3.4 *(Revision 2)* or Algorithm 3.5 *(Revision 3)* using the supplied password string.
2. If the result of step 1 is equal to the value of the encryption dictionary's **U** entry (comparing on the first 16 bytes in the case of Revision 3), the password supplied is the correct user password. The key obtained in step 1 (that is, in the first step of Algorithm 3.4 or 3.5) can be used to decrypt the document using Algorithm 3.1 on page 94.

Algorithm 3.7 ***Authenticating the owner password***

1. Compute an encryption key from the supplied password string, as described in steps 1 to 4 of Algorithm 3.3.
2. *(Revision 2 only)* Decrypt the value of the encryption dictionary's **O** entry, using an RC4 encryption function with the encryption key computed in step 1.

   (Revision 3 only) Do the following 20 times: Decrypt the value of the encryption dictionary's **O** entry (first iteration) or the output from the previous iteration (all subsequent iterations), using an RC4 encryption function with a different encryption key at each iteration. The key is generated by taking the original key (ob-

tained in step 1) and performing an XOR (exclusive or) operation between each byte of the key and the single-byte value of the iteration counter (from 19 to 0).

3. The result of step 2 purports to be the user password. Authenticate this user password using Algorithm 3.6. If it is found to be correct, the password supplied is the correct owner password.

3.5.3 Public-Key Security Handlers

Security handlers may use *public-key* encryption technology to encrypt a document (or streams within a document). When doing so, it is possible to specify one or more lists of recipients, where each list has its own unique access permissions. Only specified recipients can open the encrypted document or content, unlike the standard security handler, where a password determines access. The permissions defined for public-key security handlers are identical to those defined for the standard security handler (see Section 3.5.2, "Standard Security Handler").

Public-key security handlers use the industry standard Public Key Cryptographic Standard Number 7 (PKCS#7) binary encoding syntax to encode recipient list, decryption key and access permission information. The PKCS#7 specification can be found at <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html>.

When encrypting the data, each recipient's X.509 public key certificate (as described in ITU-T Recommendation X.509; see the Bibliography) must be available. When decrypting the data, the viewer application scans the recipient list for which the content is encrypted and attempts to find a match with a certificate that belongs to the user. If a match is found, the user will require access to the corresponding private key, which may require authentication, possibly using a password. Once access is obtained, the private key is used to decrypt the encrypted data.

Public-Key Encryption Dictionary

Encryption dictionaries for public-key security handlers contain the common entries shown in Table 3.18, whose values are described below. In addition, they may contain the entries shown in Table 3.21.

- The **Filter** entry must be the name of a known public-key security handler. Examples of existing security handlers that support public-key encryption are **Entrust.PPKEF**, **Adobe.PPKLite** and **Adobe.PubSec**.
- Permitted values of the **SubFilter** entry for use with conforming public-key security handlers are **adbe.pkcs7.s3**, **adbe.pkcs7.s4**, which are used when not using crypt filters (see Section 3.5.4, "Crypt Filters") and **adbe.pkcs7.s5**, which is used when using crypt filters.
- The **CF**, **StmF** and **StrF** entries may be present when **SubFilter** is **adbe.pkcs7.s5**.

TABLE 3.21 Additional encryption dictionary entries for public-key security handlers

KEY	TYPE	VALUE
Recipients	array	*(Required when* ***SubFilter*** *is* ***adbe.pkcs7.s3*** *or* ***adbe.pkcs7.s4****; PDF 1.3)* An array of strings, where each string is a PKCS#7 object listing recipients that have been granted equal access rights to the document. The data contained in the PKCS#7 object includes both a cryptographic key that is used to decrypt the encrypted data and the access permissions (see Table 3.20) that apply to the recipient list. There should be only one object per unique set of access permissions; if a recipient appears in more than one list, the permissions used will be those found in the first matching list. ***Note:*** *When* ***SubFilter*** *is* ***adbe.pkcs7.s5****, recipient lists are specified in the crypt filter dictionary; see Table 3.24.*

Note: *The specification for encryption dictionary entries for public-key handlers was introduced in PDF 1.3 but not documented in the* PDF Reference *until the fourth edition (PDF 1.5).*

Public-Key Encryption Algorithms

Figure 3.4 illustrates how PKCS#7 objects are used when encrypting PDF files. A PKCS#7 object is designed to encapsulate a single copy of some encrypted data, referred to as the *enveloped data*. The enveloped data in the PKCS#7 object contains keying material that must be used to decrypt the document (or individual strings or streams in the document, when crypt filters are used; see Section 3.5.4, “Crypt Filters”).

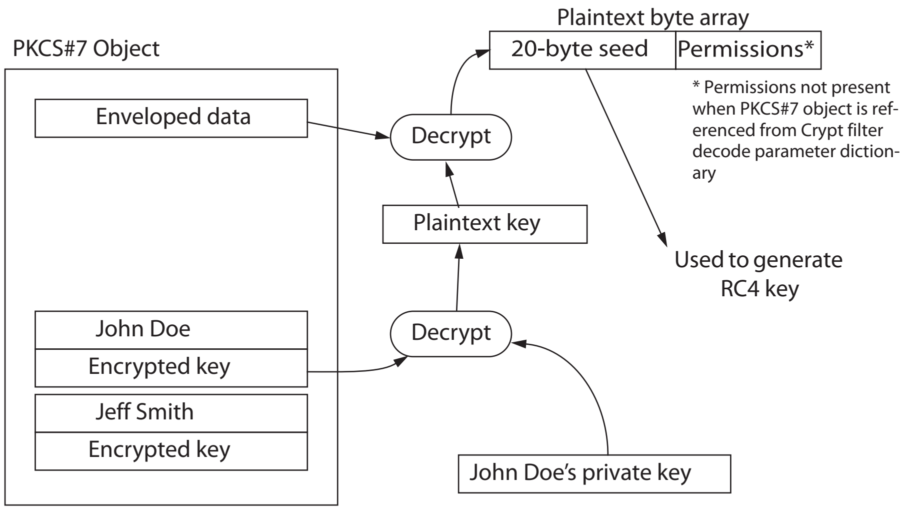


FIGURE 3.4 *Public-key encryption algorithm*

A key is used to encrypt (and decrypt) the enveloped data. This key (referred to as the “Plaintext key” in Figure 3.4) is encrypted for each recipient, using that recipient’s public key, and stored in the PKCS#7 object (as the “Encrypted key” for each recipient). To decrypt the document, that key is decrypted using the recipient’s private key, which yields a decrypted (plaintext) key. That key, in turn, is used to decrypt the enveloped data in the PKCS#7 object, resulting in a byte array that includes the following information:

- A 20-byte seed that is used to create the RC4 crypt key that is used to encrypt the document (or stream). It should be a unique random number generated by the security handler that encrypted the document.
- A 4-byte value defining the permissions, least significant byte first. See Table 3.20 for the possible permission values.
  - When **SubFilter** is **adbe.pkcs7.s3**, the relevant permissions are restricted to those specified for revision 2 of the standard security handler.
  - For **adbe.pkcs7.s4**, revision 3 permissions apply.

- For **adbe.pkcs7.s5**, which supports the use of crypt filters, the permissions are the same as **adbe.pkcs7.s4** when the crypt filter is referenced from the **StmF** or **StrF** entries of the encryption dictionary. When referenced from the **Crypt** filter decode parameter dictionary of a stream object (see Table 3.12), the 4 bytes of permissions are not included.

The RC4 key that is passed to the viewer application in order to decrypt the document contents is calculated by means of an SHA-1 message digest operation that digests the following data, in order:

1. The 20 bytes of seed.
2. The bytes of each item in the **Recipients** array of PKCS#7 objects, in the order in which they appear in the array.
3. 4 bytes with the value 0xFF, if the key being generated is intended for use in document-level encryption and the document metadata is being left as plaintext.

The first *n*/8 bytes of the resulting digest is used as the RC4 key that can decrypt data in the PDF file, where *n* is the bit length of the RC4 key.

3.5.4 Crypt Filters

PDF 1.5 introduces *crypt filters,* which provide finer granularity control of encryption. The use of crypt filters involves the following structures in a PDF file:

- The encryption dictionary (see Table 3.18) contains new entries that enumerate the crypt filters in the document (**CF**) and specify which ones are used by default to decrypt all the streams (**StmF**) and strings (**StrF**) in the document. In addition, there is a new value for the **V** entry (4) which must be specified in order to use crypt filters.
- Each crypt filter specified in the **CF** entry of the encryption dictionary is represented by a *crypt filter dictionary*, whose entries are shown in Table 3.22.
- A new stream filter type, the **Crypt** filter (see Section 3.3.9, "Crypt Filter") can be specified for any stream in the document to override the default filter for streams. A standard **Identity** filter is provided (see Table 3.23) to allow specific streams, such as document metadata, to be unencrypted in an otherwise encrypted document. The stream's **DecodeParms** entry must contain a crypt filter decode parameters dictionary (see Table 3.12) whose **Name** entry specifies the

particular crypt filter to be used (if missing, **Identity** is used). Different streams may specify different crypt filters; however, see implementation notes 22 and 23 in Appendix H.

TABLE 3.22 Entries common to all crypt filter dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* If present, must be **CryptFilter** for a crypt filter dictionary.
CFM	name	*(Optional)* The method used, if any, by the viewer application to decrypt data. In PDF 1.5, the following values are supported: • **None**: the viewer application does not decrypt data, but directs the input stream to the security handler for decryption. • **V2**: the viewer application asks the security handler for the decryption key and implicitly decrypts data using Algorithm 3.1. A viewer application may ask once for this decryption key, then cache the key for subsequent use for streams that use the same crypt filter; therefore, there must be a one-to-one relationship between a crypt filter name and the corresponding decryption key. Default value: **None**.
Length	integer	*(Optional)* When the value of **CFM** is **V2**, this entry is used to indicate the bit length of the decryption key. It must be a multiple of 8 in the range of 40 to 128. Default value: 128. When the value of **CFM** is **None**, security handlers can define their own use of this entry, but are encouraged to follow the usage conventions defined for **V2**.
AuthEvent	name	*(Optional)* The event to be used to trigger the authorization that is required to access decryption keys used by this filter. If authorization fails, the event should fail. Acceptable values are: • **DocOpen**: authorization is required when a document is opened. • **EFOpen**: authorization is required when about to access embedded files. Default value: **DocOpen**. If this filter is used as the value of **StrF** or **StmF** in the encryption dictionary (see Table 3.18), the viewer application should ignore this key and behave as if the value is **DocOpen**.

Authorization to decrypt a stream must always be obtained before the stream can be accessed. This typically occurs when the document is opened, as specified by a value of **DocOpen** for the **AuthEvent** entry in the crypt filter dictionary. PDF viewer applications and security handlers should treat any attempt to access a stream for which authorization has failed as an error. **AuthEvent** may also be **EFOpen**, indicating the presence of an encrypted embedded file in an otherwise unencrypted document; see implementation note 24 in Appendix H.

The security handler may do its own decryption by specifying a value of **None** for the **CFM** entry in the crypt filter dictionary. This allows the handler to tightly control key management and use any encryption algorithm it wishes. A value of **V2** for **CFM** is equivalent to using the standard decryption algorithm.

Security handlers can add their own private data to crypt filter dictionaries; names for private data entries must conform to the PDF name registry (see Appendix E, "PDF Name Registry").

TABLE 3.23 Standard crypt filter names

NAME	DESCRIPTION
Identity	Input data is passed through without any processing.

Table 3.24 lists the additional crypt filter dictionary entries used by public-key security handlers (see Section 3.5.3, "Public-Key Security Handlers"). When these entries are present, the value of **CFM** must be **V2**.

TABLE 3.24 Additional crypt filter dictionary entries for public-key security handlers

KEY	TYPE	VALUE
Recipients	array or string	*(Required)* If the crypt filter is referenced from **StmF** or **StrF** in the encryption dictionary, this entry is an array of strings, where each string is a binary-encoded PKCS#7 object listing recipients that have been granted equal access rights to the document. The enveloped data contained in the PKCS#7 object includes both a 20-byte seed value used to compute the decryption key (see "Public-Key Encryption Algorithms" on page 104) followed by 4 bytes of permissions settings (see Table 3.20) that apply to the recipient list. There should be only one object per unique set of access permissions; if a recipient appears in more than one list, the permissions used will be those found in the first matching list. If the crypt filter is referenced from a crypt filter decode parameter dictionary (see Table 3.12), this entry is a string that is a binary-encoded PKCS#7 object, containing a list of all recipients that are permitted to access the corresponding encrypted stream. The enveloped data contained in the PKCS#7 object is a 20-byte seed value used to create the cryptographic key that is used to decrypt the stream data.
EncryptMetadata	boolean	*(Optional; used only by crypt filters that are referenced from* ***StmF*** *or* ***StrF*** *in an encryption dictionary)* Indicates whether the document-level metadata stream (see Section 10.2.2, "Metadata Streams") is to be encrypted. Default value: **true**.

Example 3.12 shows the use of crypt filters in an encrypted document containing an unencrypted XML metadata stream. The metadata stream is left as is by applying the **Identity** crypt filter; the remaining streams and strings are decrypted using the default filters.

Example 3.12

```
%PDF1.5
1 0 obj                        % Document catalog
   << /Type /Catalog
      /Pages 2 0 R
      /Metadata 6 0 R
   >>
endobj
2 0 obj                        % Page tree
   << /Type /Pages
      /Kids [3 0 R]
```

```
        /Count 1
    >>
endobj
3 0 obj                         % 1s t page
    << /Type /Page
        /Parent 2 0 R
        /MediaBox [0 0 612 792]
        /Contents 4 0 R
    >>
endobj
4 0 obj                         % Page contents
    << /Length 35 >>
    stream
        *** Encrypted Page-marking operators ***
    endstream
endobj
5 0 obj
    << /Title ($#*#%*$#^&##) >> % Info dictionary: encrypted text string
endobj
6 0 obj
    << /Type /Metadata
        /Subtype /XML
        /Length 15
        /Filter [/Crypt]        % Uses a crypt filter
        /DecodeParms [7 0 R]    % with these parameters
    >>
    stream
        </XML Metadata stuff /> % Unencrypted metadata
    endstream
endobj
7 0 R
    << /Type /CryptFilterDecodeParms       %Parameters for metadata stream filter
        /Name /Identity                    % /Identity = no encryption
    >>
endobj
8 0 obj                                    % Encryption dictionary
    << /Filter /MySecurityHandlerName
        /V 4                               % Version 4: allow crypt filters
        /CF                                % List of crypt filters
        << /MyFilter0
            << /Type /CryptFilter
                /CFM V2                    % Uses the standard algorithm
            >>
            >>
            /StrF /MyFilter0                    % Strings are decrypted using /MyFilter0
            /StmF /MyFilter0                    % Streams are decrypted using /MyFilter0
                ...                             % Private data for /MySecurityHandlerName
            /MyUnsecureKey (12345678)
        >>
    endobj
    xref
        ....
    trailer
        << /Size 8
            /Root 1 0 R
            /Info 5 0 R
            /Encrypt 8 0 R
        >>
    startxref
    495
    %%EOF
```

3.6 Document Structure

A PDF document can be regarded as a hierarchy of objects contained in the body section of a PDF file. At the root of the hierarchy is the document's *catalog* dictionary (see Section 3.6.1, "Document Catalog"). Most of the objects in the hierarchy are dictionaries. For example, each page of the document is represented by a *page object*—a dictionary that includes references to the page's contents and other attributes, such as its thumbnail image (Section 8.2.3, "Thumbnail Images") and any annotations (Section 8.4, "Annotations") associated with it. The individual page objects are tied together in a structure called the *page tree* (described in Section 3.6.2, "Page Tree"), which in turn is located via an indirect reference in the document catalog. Parent, child, and sibling relationships within the hierarchy are defined by dictionary entries whose values are indirect references to other dictionaries. Figure 3.5 illustrates the structure of the object hierarchy.

***Note:** The data structures described in this section, particularly the catalog and page dictionaries, combine entries describing document structure with ones dealing with the detailed semantics of documents and pages. All entries are listed here, but many of their descriptions are deferred to subsequent chapters.*

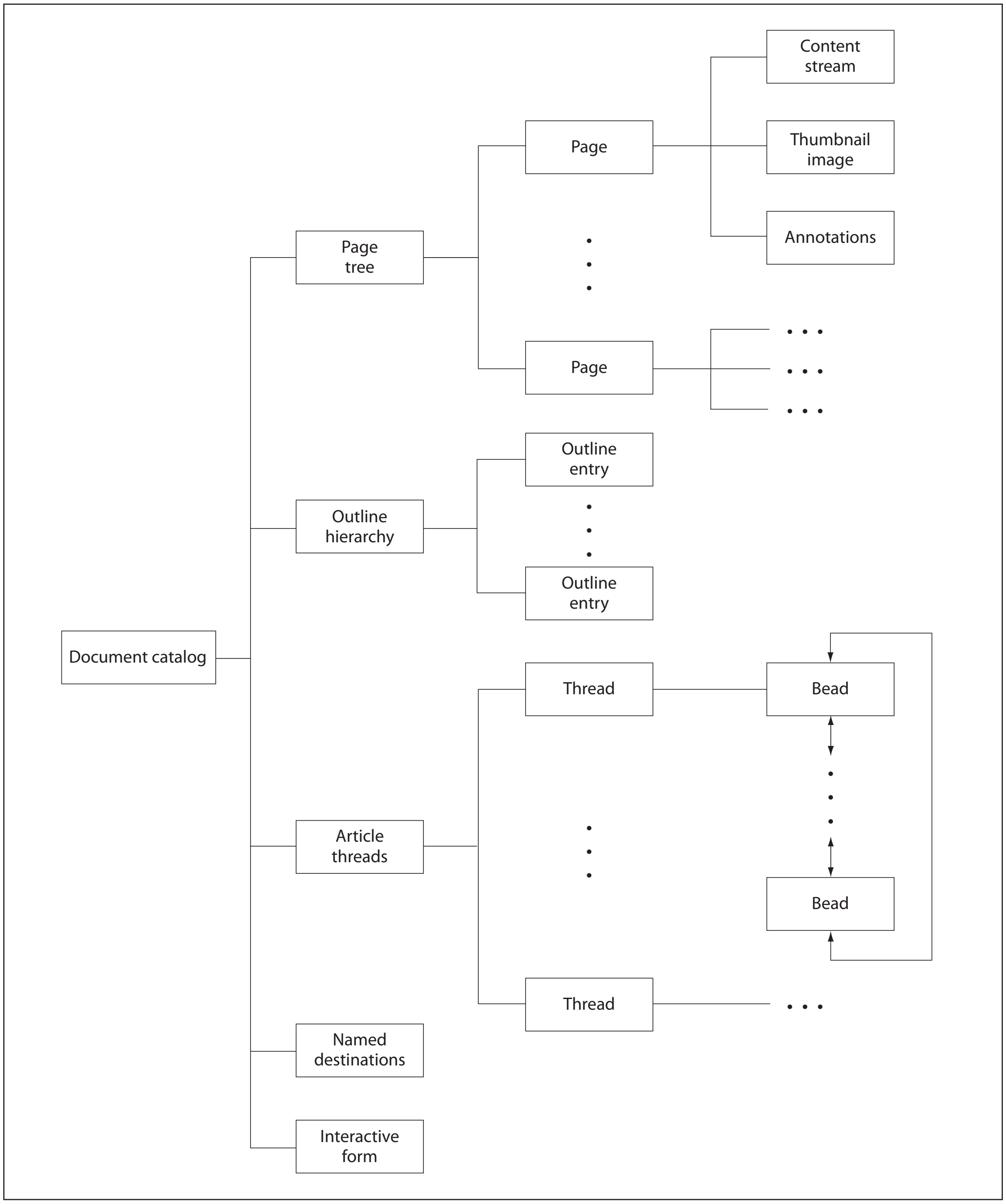


FIGURE 3.5 *Structure of a PDF document*

3.6.1 Document Catalog

The root of a document's object hierarchy is the *catalog* dictionary, located via the **Root** entry in the trailer of the PDF file (see Section 3.4.4, "File Trailer"). The catalog contains references to other objects defining the document's contents, outline, article threads *(PDF 1.1)*, named destinations, and other attributes. In addition, it contains information about how the document should be displayed on the screen, such as whether its outline and thumbnail page images should be displayed automatically and whether some location other than the first page should be shown when the document is opened. Table 3.25 shows the entries in the catalog dictionary.

TABLE 3.25 Entries in the catalog dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Catalog** for the catalog dictionary.
Version	name	*(Optional; PDF 1.4)* The version of the PDF specification to which the document conforms (for example, 1.4), if later than the version specified in the file's header (see Section 3.4.1, "File Header"). If the header specifies a later version, or if this entry is absent, the document conforms to the version specified in the header. This entry enables a PDF producer application to update the version using an incremental update; see Section 3.4.5, "Incremental Updates." (See implementation note 25 in Appendix H.) ***Note:*** *The value of this entry is a name object, not a number, and so must be preceded by a slash character (/) when written in the PDF file (for example, /1.4).*
Pages	dictionary	*(Required; must be an indirect reference)* The *page tree node* that is the root of the document's *page tree* (see Section 3.6.2, "Page Tree").
PageLabels	number tree	*(Optional; PDF 1.3)* A number tree (see Section 3.8.6, "Number Trees") defining the page labeling for the document. The keys in this tree are page indices; the corresponding values are *page label dictionaries* (see Section 8.3.1, "Page Labels"). Each page index denotes the first page in a *labeling range* to which the specified page label dictionary applies. The tree must include a value for page index 0.
Names	dictionary	*(Optional; PDF 1.2)* The document's *name dictionary* (see Section 3.6.3, "Name Dictionary").

KEY	TYPE	VALUE
Dests	dictionary	*(Optional; PDF 1.1; must be an indirect reference)* A dictionary of names and corresponding *destinations* (see "Named Destinations" on page 542).
ViewerPreferences	dictionary	*(Optional; PDF 1.2)* A *viewer preferences dictionary* (see Section 8.1, "Viewer Preferences") specifying the way the document is to be displayed on the screen. If this entry is absent, viewer applications should use their own current user preference settings.
PageLayout	name	*(Optional)* A name object specifying the page layout to be used when the document is opened: SinglePage — Display one page at a time. OneColumn — Display the pages in one column. TwoColumnLeft — Display the pages in two columns, with odd-numbered pages on the left. TwoColumnRight — Display the pages in two columns, with odd-numbered pages on the right. (See implementation note 26 in Appendix H.) Default value: SinglePage.
PageMode	name	*(Optional)* A name object specifying how the document should be displayed when opened: UseNone — Neither document outline nor thumbnail images visible UseOutlines — Document outline visible UseThumbs — Thumbnail images visible FullScreen — Full-screen mode, with no menu bar, window controls, or any other window visible UseOC — Optional content group panel visible Default value: UseNone.
Outlines	dictionary	*(Optional; must be an indirect reference)* The *outline dictionary* that is the root of the document's *outline hierarchy* (see Section 8.2.2, "Document Outline").
Threads	array	*(Optional; PDF 1.1; must be an indirect reference)* An array of *thread dictionaries* representing the document's *article threads* (see Section 8.3.2, "Articles").

KEY	TYPE	VALUE
OpenAction	array or dictionary	*(Optional; PDF 1.1)* A value specifying a *destination* to be displayed or an *action* to be performed when the document is opened. The value is either an array defining a destination (see Section 8.2.1, "Destinations") or an *action dictionary* representing an action (Section 8.5, "Actions"). If this entry is absent, the document should be opened to the top of the first page at the default magnification factor.
AA	dictionary	*(Optional; PDF 1.4)* An *additional-actions dictionary* defining the actions to be taken in response to various *trigger events* affecting the document as a whole (see "Trigger Events" on page 593). (See also implementation note 27 in Appendix H.)
URI	dictionary	*(Optional; PDF 1.1)* A *URI dictionary* containing document-level information for *URI (uniform resource identifier) actions* (see "URI Actions" on page 602).
AcroForm	dictionary	*(Optional; PDF 1.2)* The document's *interactive form (AcroForm) dictionary* (see Section 8.6.1, "Interactive Form Dictionary").
Metadata	stream	*(Optional; PDF 1.4; must be an indirect reference)* A *metadata stream* containing metadata for the document (see Section 10.2.2, "Metadata Streams").
StructTreeRoot	dictionary	*(Optional; PDF 1.3)* The document's *structure tree root* dictionary (see Section 10.6.1, "Structure Hierarchy").
MarkInfo	dictionary	*(Optional; PDF 1.4)* A *mark information dictionary* containing information about the document's usage of Tagged PDF conventions (see Section 10.7.1, "Mark Information Dictionary").
Lang	text string	*(Optional; PDF 1.4)* A *language identifier* specifying the natural language for all text in the document except where overridden by language specifications for structure elements or marked content (see Section 10.8.1, "Natural Language Specification"). If this entry is absent, the language is considered unknown.
SpiderInfo	dictionary	*(Optional; PDF 1.3)* A *Web Capture information dictionary* containing state information used by the Acrobat Web Capture (AcroSpider) plug-in extension (see Section 10.9.1, "Web Capture Information Dictionary").
OutputIntents	array	*(Optional; PDF 1.4)* An array of *output intent dictionaries* describing the color characteristics of output devices on which the document might be rendered (see "Output Intents" on page 841).

KEY	TYPE	VALUE
PieceInfo	dictionary	*(Optional; PDF 1.4)* A *page-piece dictionary* associated with the document (see Section 10.4, "Page-Piece Dictionaries")
OCProperties	dictionary	*(Optional; PDF 1.5; required if a document contains optional content)* The document's *optional content properties dictionary* (see Section 4.10.3, "Configuring Optional Content").
Perms	dictionary	*(Optional; PDF 1.5)* A *permissions dictionary* that specifies user access permissions for the document. Section 8.7.3, "Permissions," describes this dictionary and how it is used.
Legal	dictionary	*(Optional; PDF 1.5)* A dictionary containing attestations regarding the content of a PDF document, as it relates to the legality of digital signatures (see Section 8.7.4, "Legal Content Attestations").

Example 3.13 shows a sample catalog object.

Example 3.13

```
1 0 obj
   << /Type /Catalog
      /Pages 2 0 R
      /PageMode /UseOutlines
      /Outlines 3 0 R
   >>
endobj
```

3.6.2 Page Tree

The pages of a document are accessed through a structure known as the *page tree*, which defines their ordering within the document. The tree structure allows PDF viewer applications to quickly open a document containing thousands of pages using only limited memory. The tree contains nodes of two types—intermediate nodes, called *page tree nodes*, and leaf nodes, called *page objects*—whose form is described in the sections below. Viewer applications should be prepared to handle any form of tree structure built of such nodes. The simplest structure would consist of a single page tree node that references all of the document's page objects directly; however, to optimize the performance of viewer applications, the Acrobat Distiller program constructs trees of a particular form, known as *balanced trees*. Further information on this form of tree can be found in *Data Structures and Algorithms*, by Aho, Hopcroft, and Ullman (see the Bibliography).

Page Tree Nodes

Table 3.26 shows the required entries in a page tree node.

TABLE 3.26 Required entries in a page tree node

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Pages** for a page tree node.
Parent	dictionary	*(Required except in root node; must be an indirect reference)* The page tree node that is the immediate parent of this one.
Kids	array	*(Required)* An array of indirect references to the immediate children of this node. The children may be page objects or other page tree nodes.
Count	integer	*(Required)* The number of leaf nodes (page objects) that are descendants of this node within the page tree.

Note: *The structure of the page tree is not necessarily related to the logical structure of the document itself; that is, page tree nodes do not represent chapters, sections, and so forth. (Other data structures are defined for that purpose; see Section 10.6, "Logical Structure.") Applications that consume or produce PDF files are not required to preserve the existing structure of the page tree.*

Example 3.14 illustrates the page tree for a document with three pages. See "Page Objects," below, for the contents of the individual page objects, and Section G.4, "Page Tree Example," for a more extended example showing the page tree for a longer document.

Example 3.14

```
2 0 obj
   << /Type /Pages
      /Kids [ 4 0 R
              10 0 R
              24 0 R
            ]
      /Count 3
   >>
endobj
```

```
4 0 obj
   << /Type /Page
      …Additional entries describing the attributes of this page…
   >>
endobj

10 0 obj
   << /Type /Page
      …Additional entries describing the attributes of this page…
   >>
endobj

24 0 obj
   << /Type /Page
      …Additional entries describing the attributes of this page…
   >>
endobj
```

In addition to the entries shown in Table 3.26, a page tree node may contain further entries defining *inherited attributes* for the page objects that are its descendants (see “Inheritance of Page Attributes” on page 122).

Page Objects

The leaves of the page tree are *page objects*, each of which is a dictionary specifying the attributes of a single page of the document. Table 3.27 shows the contents of this dictionary (see also implementation note 28 in Appendix H). The table also identifies which attributes a page may inherit from its ancestor nodes in the page tree, as described under “Inheritance of Page Attributes” on page 122. Attributes that are not explicitly identified in the table as inheritable cannot be inherited.

TABLE 3.27 Entries in a page object

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Page** for a page object.
Parent	dictionary	*(Required; must be an indirect reference)* The page tree node that is the immediate parent of this page object.

KEY	TYPE	VALUE
LastModified	date	*(Required if* ***PieceInfo*** *is present; optional otherwise; PDF 1.3)* The date and time (see Section 3.8.3, "Dates") when the page's contents were most recently modified. If a page-piece dictionary (**PieceInfo**) is present, the modification date is used to ascertain which of the application data dictionaries that it contains correspond to the current content of the page (see Section 10.4, "Page-Piece Dictionaries").
Resources	dictionary	*(Required; inheritable)* A dictionary containing any resources required by the page (see Section 3.7.2, "Resource Dictionaries"). If the page requires no resources, the value of this entry should be an empty dictionary; omitting the entry entirely indicates that the resources are to be inherited from an ancestor node in the page tree.
MediaBox	rectangle	*(Required; inheritable)* A rectangle (see Section 3.8.4, "Rectangles"), expressed in default user space units, defining the boundaries of the physical medium on which the page is intended to be displayed or printed (see Section 10.10.1, "Page Boundaries").
CropBox	rectangle	*(Optional; inheritable)* A rectangle, expressed in default user space units, defining the visible region of default user space. When the page is displayed or printed, its contents are to be clipped (cropped) to this rectangle and then imposed on the output medium in some implementation-defined manner (see Section 10.10.1, "Page Boundaries"). Default value: the value of **MediaBox**.
BleedBox	rectangle	*(Optional; PDF 1.3)* A rectangle, expressed in default user space units, defining the region to which the contents of the page should be clipped when output in a production environment (see Section 10.10.1, "Page Boundaries"). Default value: the value of **CropBox**.
TrimBox	rectangle	*(Optional; PDF 1.3)* A rectangle, expressed in default user space units, defining the intended dimensions of the finished page after trimming (see Section 10.10.1, "Page Boundaries"). Default value: the value of **CropBox**.
ArtBox	rectangle	*(Optional; PDF 1.3)* A rectangle, expressed in default user space units, defining the extent of the page's meaningful content (including potential white space) as intended by the page's creator (see Section 10.10.1, "Page Boundaries"). Default value: the value of **CropBox**.

KEY	TYPE	VALUE
BoxColorInfo	dictionary	*(Optional; PDF 1.4)* A *box color information dictionary* specifying the colors and other visual characteristics to be used in displaying guidelines on the screen for the various page boundaries (see "Display of Page Boundaries" on page 836). If this entry is absent, the viewer application should use its own current default settings.
Contents	stream or array	*(Optional)* A *content stream* (see Section 3.7.1, "Content Streams") describing the contents of this page. If this entry is absent, the page is empty. The value may be either a single stream or an array of streams. If it is an array, the effect is as if all of the streams in the array were concatenated, in order, to form a single stream. This allows a program generating a PDF file to create image objects and other resources as they occur, even though they interrupt the content stream. The division between streams may occur only at the boundaries between lexical tokens (see Section 3.1, "Lexical Conventions"), but is unrelated to the page's logical content or organization. Applications that consume or produce PDF files are not required to preserve the existing structure of the **Contents** array. (See implementation note 29 in Appendix H.)
Rotate	integer	*(Optional; inheritable)* The number of degrees by which the page should be rotated clockwise when displayed or printed. The value must be a multiple of 90. Default value: 0.
Group	dictionary	*(Optional; PDF 1.4)* A *group attributes dictionary* specifying the attributes of the page's page group for use in the transparent imaging model (see Sections 7.3.6, "Page Group," and 7.5.5, "Transparency Group XObjects").
Thumb	stream	*(Optional)* A stream object defining the page's *thumbnail image* (see Section 8.2.3, "Thumbnail Images").
B	array	*(Optional; PDF 1.1; recommended if the page contains article beads)* An array of indirect references to *article beads* appearing on the page (see Section 8.3.2, "Articles"; see also implementation note 30 in Appendix H). The beads are listed in the array in natural reading order.
Dur	number	*(Optional; PDF 1.1)* The page's *display duration* (also called its *advance timing*): the maximum length of time, in seconds, that the page will be displayed during presentations before the viewer application automatically advances to the next page (see Section 8.3.3, "Presentations"). By default, the viewer does not advance automatically.

KEY	TYPE	VALUE
Trans	dictionary	*(Optional; PDF 1.1)* A *transition dictionary* describing the transition effect to be used when displaying the page during presentations (see Section 8.3.3, "Presentations").
Annots	array	*(Optional)* An array of *annotation dictionaries* representing annotations associated with the page (see Section 8.4, "Annotations").
AA	dictionary	*(Optional; PDF 1.2)* An *additional-actions dictionary* defining actions to be performed when the page is opened or closed (see Section 8.5.2, "Trigger Events"; see also implementation note 31 in Appendix H).
Metadata	stream	*(Optional; PDF 1.4)* A *metadata stream* containing metadata for the page (see Section 10.2.2, "Metadata Streams").
PieceInfo	dictionary	*(Optional; PDF 1.3)* A *page-piece dictionary* associated with the page (see Section 10.4, "Page-Piece Dictionaries").
StructParents	integer	*(Required if the page contains structural content items; PDF 1.3)* The integer key of the page's entry in the *structural parent tree* (see "Finding Structure Elements from Content Items" on page 739).
ID	string	*(Optional; PDF 1.3; indirect reference preferred)* The digital identifier of the page's parent *Web Capture content set* (see Section 10.9.5, "Object Attributes Related to Web Capture").
PZ	number	*(Optional; PDF 1.3)* The page's preferred *zoom (magnification) factor*: the factor by which it should be scaled to achieve the "natural" display magnification (see Section 10.9.5, "Object Attributes Related to Web Capture").
SeparationInfo	dictionary	*(Optional; PDF 1.3)* A *separation dictionary* containing information needed to generate color separations for the page (see Section 10.10.3, "Separation Dictionaries").
Tabs	name	*(Optional; PDF 1.5)* A name specifying the tab order to be used for annotations on the page. The possible values are R (row order), C (column order), and S (structure order). See Section 8.4, "Annotations," for details.
TemplateInstantiated	name	*(Required if this page was created from a named page object; PDF 1.5)* The name of the originating page object (see Section 8.6.5, "Named Pages").
PresSteps	dictionary	*(Optional; PDF 1.5)* A *navigation node dictionary* representing the first node on the page (see "Sub-page Navigation" on page 555).

Example 3.15 shows the definition of a page object with a thumbnail image and two annotations. The media box specifies that the page is to be printed on letter-size paper. In addition, the resource dictionary is specified as a direct object and shows that the page makes use of three fonts, named F3, F5, and F7.

Example 3.15

```
3 0 obj
   << /Type /Page
      /Parent 4 0 R
      /MediaBox [0 0 612 792]
      /Resources << /Font << /F3 7 0 R
                             /F5 9 0 R
                             /F7 11 0 R
                          >>
                   /ProcSet [/PDF]
                >>
      /Contents 12 0 R
      /Thumb 14 0 R
      /Annots [ 23 0 R
                24 0 R
              ]
   >>
endobj
```

Inheritance of Page Attributes

Some of the page attributes shown in Table 3.27 are designated as *inheritable*. If such an attribute is omitted from a page object, its value is inherited from an ancestor node in the page tree. If the attribute is a required one, a value must be supplied in an ancestor node; if it is optional and no inherited value is specified, the default value is used.

An attribute can thus be defined once for a whole set of pages, by specifying it in an intermediate page tree node and arranging the pages that share the attribute as descendants of that node. For example, a document might specify the same media box for all of its pages by including a **MediaBox** entry in the root node of the page tree. If necessary, an individual page object could then override this inherited value with a **MediaBox** entry of its own.

***Note:** In a document conforming to the Linearized PDF organization (see Appendix F), all page attributes must be specified explicitly as entries in the page dictionaries to which they apply; they may not be inherited from an ancestor node.*

Figure 3.6 illustrates the inheritance of attributes. In the page tree shown, pages 1, 2, and 4 are rotated clockwise by 90 degrees, page 3 by 270 degrees, page 6 by 180 degrees, and pages 5 and 7 not at all (0 degrees).

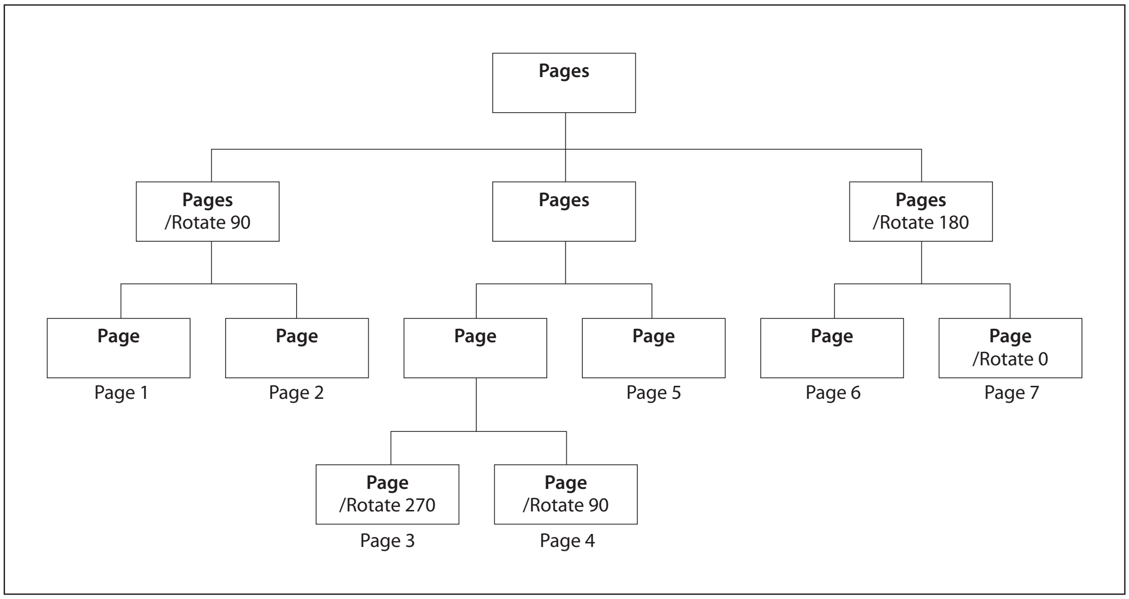


FIGURE 3.6 *Inheritance of attributes*

3.6.3 Name Dictionary

Some categories of objects in a PDF file can be referred to by name rather than by object reference. The correspondence between names and objects is established by the document's *name dictionary (PDF 1.2)*, located via the **Names** entry in the document's catalog (see Section 3.6.1, "Document Catalog"). Each entry in this dictionary designates the root of a name tree (Section 3.8.5, "Name Trees") defining names for a particular category of objects. Table 3.28 shows the contents of the name dictionary.

3.7 Content Streams and Resources

Content streams are the primary means for describing the appearance of pages and other graphical elements. A content stream depends on information contained in an associated resource dictionary; in combination, these two objects form a self-contained entity. This section describes these objects.

TABLE 3.28 Entries in the name dictionary

KEY	TYPE	VALUE
Dests	name tree	*(Optional; PDF 1.2)* A name tree mapping name strings to destinations (see “Named Destinations” on page 542).
AP	name tree	*(Optional; PDF 1.3)* A name tree mapping name strings to annotation appearance streams (see Section 8.4.4, “Appearance Streams”).
JavaScript	name tree	*(Optional; PDF 1.3)* A name tree mapping name strings to document-level JavaScript actions (see “JavaScript Actions” on page 645).
Pages	name tree	*(Optional; PDF 1.3)* A name tree mapping name strings to visible pages for use in interactive forms (see Section 8.6.5, “Named Pages”).
Templates	name tree	*(Optional; PDF 1.3)* A name tree mapping name strings to invisible (template) pages for use in interactive forms (see Section 8.6.5, “Named Pages”).
IDS	name tree	*(Optional; PDF 1.3)* A name tree mapping digital identifiers to Web Capture content sets (see Section 10.9.3, “Content Sets”).
URLS	name tree	*(Optional; PDF 1.3)* A name tree mapping uniform resource locators (URLs) to Web Capture content sets (see Section 10.9.3, “Content Sets”).
EmbeddedFiles	name tree	*(Optional; PDF 1.4)* A name tree mapping name strings to file specifications for embedded file streams (see Section 3.10.3, “Embedded File Streams”).
AlternatePresentations	name tree	*(Optional; PDF 1.4)* A name tree mapping name strings to alternate presentations (see Section 9.4, “Alternate Presentations”).
Renditions	name tree	*(Optional; PDF 1.5)* A name tree mapping name strings (which must have Unicode encoding) to rendition objects (see Section 9.1.2, “Renditions”).

3.7.1 Content Streams

A *content stream* is a PDF stream object whose data consists of a sequence of instructions describing the graphical elements to be painted on a page. The instructions are represented in the form of PDF objects, using the same object syntax as in the rest of the PDF document. However, whereas the document as a whole is a static, random-access data structure, the objects in the content stream are intended to be interpreted and acted upon sequentially.

Each page of a document is represented by one or more content streams. Content streams are also used to package up sequences of instructions as self-contained graphical elements, such as forms (see Section 4.9, "Form XObjects"), patterns (Section 4.6, "Patterns"), certain fonts (Section 5.5.4, "Type 3 Fonts"), and annotation appearances (Section 8.4.4, "Appearance Streams").

A content stream, after decoding with any specified filters, is interpreted according to the PDF syntax rules described in Section 3.1, "Lexical Conventions." It consists of PDF objects denoting operands and operators. The operands needed by an operator precede it in the stream. See Example 3.3 on page 44 for an example of a content stream.

An *operand* is a direct object belonging to any of the basic PDF data types except a stream. Dictionaries are permitted as operands only by certain specific operators. Indirect objects and object references are not permitted at all.

An *operator* is a PDF keyword that specifies some action to be performed, such as painting a graphical shape on the page. An operator keyword is distinguished from a name object by the absence of an initial slash character (/). Operators are meaningful only inside a content stream.

***Note:** This "postfix" notation, in which an operator is preceded by its operands, is superficially the same as in the PostScript language. However, PDF has no concept of an operand stack as PostScript has. In PDF, all of the operands needed by an operator must immediately precede that operator. Operators do not return results, and there may not be operands left over when an operator finishes execution.*

Most operators have to do with painting graphical elements on the page or with specifying parameters that affect subsequent painting operations. The individual operators are described in the chapters devoted to their functions:

- Chapter 4 describes operators that paint general graphics, such as filled areas, strokes, and sampled images, and that specify device-independent graphical parameters, such as color.
- Chapter 5 describes operators that paint text using character glyphs defined in fonts.
- Chapter 6 describes operators that specify device-dependent rendering parameters.
- Chapter 10 describes the marked-content operators that associate higher-level logical information with objects in the content stream. These operators do not affect the rendered appearance of the content; rather, they specify information useful to applications that use PDF for document interchange.

Ordinarily, when a viewer application encounters an operator in a content stream that it does not recognize, an error will occur. (See implementation note 32 in Appendix H.) A pair of compatibility operators, **BX** and **EX** *(PDF 1.1)*, modify this behavior (see Table 3.29). These operators must occur in pairs and may be nested. They bracket a *compatibility section*, a portion of a content stream within which unrecognized operators are to be ignored without error. This mechanism enables a PDF document to use operators defined in newer versions of PDF without sacrificing compatibility with older viewers; it should be used only in cases where ignoring such newer operators is the appropriate thing to do. The **BX** and **EX** operators are not themselves part of any graphics object (see Section 4.1, "Graphics Objects") or of the graphics state (Section 4.3, "Graphics State").

TABLE 3.29 Compatibility operators

OPERANDS	OPERATOR	DESCRIPTION
—	**BX**	*(PDF 1.1)* Begin a compatibility section. Unrecognized operators (along with their operands) will be ignored without error until the balancing **EX** operator is encountered.
—	**EX**	*(PDF 1.1)* End a compatibility section begun by a balancing **BX** operator.

3.7.2 Resource Dictionaries

As stated above, the operands supplied to operators in a content stream may only be direct objects; indirect objects and object references are not permitted. In some cases, an operator needs to refer to a PDF object that is defined outside the content stream, such as a font dictionary or a stream containing image data. This can be accomplished by defining such objects as *named resources* and referring to them by name from within the content stream.

Note: *Named resources are meaningful only in the context of a content stream. The scope of a resource name is local to a particular content stream, and is unrelated to externally known identifiers for objects such as fonts. References from one object to another outside of content streams should be made by means of indirect object references rather than named resources.*

A content stream's named resources are defined by a *resource dictionary*, which enumerates the named resources needed by the operators in the content stream and the names by which they can be referred to. For example, if a text operator appearing within the content stream needed a certain font, the content stream's resource dictionary might associate the name F42 with the corresponding font dictionary. The text operator could then use this name to refer to the font.

A resource dictionary is associated with a content stream in one of the following ways:

- For a content stream that is the value of a page's **Contents** entry (or is an element of an array that is the value of that entry), the resource dictionary is designated by the page dictionary's **Resources** entry. (Since a page's **Resources** attribute is inheritable, as described under "Inheritance of Page Attributes" on page 122, it may actually reside in some ancestor node of the page object.)
- For other content streams, the resource dictionary is specified by the **Resources** entry in the stream dictionary of the content stream itself. This applies to content streams that define form XObjects, patterns, Type 3 fonts, and annotation appearances.
- A form XObject or a Type 3 font's glyph description may omit the **Resources** entry, in which case resources will be looked up in the **Resources** entry of the page on which the form or font is used. *This practice is not recommended.*

In the context of a given content stream, the term *current resource dictionary* refers to the resource dictionary associated with the stream in one of the ways described above.

Each key in a resource dictionary is the name of a resource type, as shown in Table 3.30. The corresponding values are as follows:

- For resource type **ProcSet**, the value is an array of procedure set names
- For all other resource types, the value is a subdictionary. Each key in the subdictionary is the name of a specific resource, and the corresponding value is a PDF object associated with the name.

TABLE 3.30 Entries in a resource dictionary

KEY	TYPE	VALUE
ExtGState	dictionary	*(Optional)* A dictionary mapping resource names to graphics state parameter dictionaries (see Section 4.3.4, "Graphics State Parameter Dictionaries").
ColorSpace	dictionary	*(Optional)* A dictionary mapping each resource name to either the name of a device-dependent color space or an array describing a color space (see Section 4.5, "Color Spaces").
Pattern	dictionary	*(Optional)* A dictionary mapping resource names to pattern objects (see Section 4.6, "Patterns").
Shading	dictionary	*(Optional; PDF 1.3)* A dictionary mapping resource names to shading dictionaries (see "Shading Dictionaries" on page 266).
XObject	dictionary	*(Optional)* A dictionary mapping resource names to external objects (see Section 4.7, "External Objects").
Font	dictionary	*(Optional)* A dictionary mapping resource names to font dictionaries (see Chapter 5).
ProcSet	array	*(Optional)* An array of predefined procedure set names (see Section 10.1, "Procedure Sets").
Properties	dictionary	*(Optional; PDF 1.2)* A dictionary mapping resource names to property list dictionaries for marked content (see Section 10.5.1, "Property Lists").

Example 3.16 shows a resource dictionary containing procedure sets, fonts, and external objects. The procedure sets are specified by an array, as described in Section 10.1, "Procedure Sets." The fonts are specified with a subdictionary associat-

ing the names F5, F6, F7, and F8 with objects 6, 8, 10, and 12, respectively. Likewise, the **XObject** subdictionary associates the names Im1 and Im2 with objects 13 and 15, respectively.

Example 3.16

```
<< /ProcSet [/PDF /ImageB]
   /Font << /F5  6 0 R
            /F6  8 0 R
            /F7  10 0 R
            /F8  12 0 R
         >>
   /XObject << /Im1  13 0 R
               /Im2  15 0 R
            >>
>>
```

3.8 Common Data Structures

As mentioned at the beginning of this chapter, there are some general-purpose data structures that are built from the basic object types described in Section 3.2, “Objects,” and are used in many places throughout PDF. This section describes data structures for text strings, dates, rectangles, name trees, and number trees. The subsequent two sections describe more complex data structures for functions and file specifications.

All of these data structures are meaningful only as part of the document hierarchy; they cannot appear within content streams. In particular, the special conventions for interpreting the values of string objects apply only to strings outside content streams. An entirely different convention is used within content streams for using strings to select sequences of glyphs to be painted on the page (see Chapter 5). Table 3.31 summarizes the basic and higher-level data types that are used throughout this book to describe the values of dictionary entries and other PDF data values.

TABLE 3.31 PDF data types

TYPE	DESCRIPTION	SECTION	PAGE
array	Array object	3.2.5	34
boolean	Boolean value	3.2.1	28
date	Date (string)	3.8.3	132
dictionary	Dictionary object	3.2.6	35
file specification	File specification (string or dictionary)	3.10	150
function	Function (dictionary or stream)	3.9	138
integer	Integer number	3.2.2	28
name	Name object	3.2.4	32
name tree	Name tree (dictionary)	3.8.5	133
null	Null object	3.2.8	39
number	Number (integer or real)	3.2.2	28
number tree	Number tree (dictionary)	3.8.6	137
rectangle	Rectangle (array)	3.8.4	133
stream	Stream object	3.2.7	36
string	String object	3.2.3	29
text string	Text string	3.8.1	130
text stream	Text stream	3.8.2	131

3.8.1 Text Strings

Certain strings contain information that is intended to be human-readable, such as text annotations, bookmark names, article names, document information, and so forth. Such strings are referred to as *text strings*. Text strings are encoded in either **PDFDocEncoding** or Unicode character encoding. **PDFDocEncoding** is a superset of the ISO Latin 1 encoding and is documented in Appendix D. Unicode is described in the *Unicode Standard* by the Unicode Consortium (see the Bibliography).

For text strings encoded in Unicode, the first two bytes must be 254 followed by 255. These two bytes represent the Unicode byte order marker, U+FEFF, indicating that the string is encoded in the UTF-16BE (big-endian) encoding scheme specified in the Unicode standard. (This mechanism precludes beginning a string using **PDFDocEncoding** with the two characters thorn ydieresis, which is unlikely to be a meaningful beginning of a word or phrase).

Note: *Applications that process PDF files containing Unicode text strings should be prepared to handle supplementary characters; that is, characters requiring more than two bytes to represent.*

Anywhere in a Unicode text string, an escape sequence may appear to indicate the language in which subsequent text is written; this is useful when the language cannot be determined from the character codes used in the text itself. The escape sequence consists of the following elements, in order:

1. The Unicode value U+001B (that is, the byte sequence 0 followed by 27)
2. A 2-character ISO 639 language code—for example, en for English or ja for Japanese
3. *(Optional)* A 2-character ISO 3166 country code—for example, US for the United States or JP for Japan
4. The Unicode value U+001B

The complete list of codes defined by ISO 639 and ISO 3166 can be obtained from the International Organization for Standardization (see the Bibliography).

3.8.2 Text Streams

A *text stream (PDF 1.5)* is a PDF stream object (Section 3.2.7) whose unencoded bytes meet the same requirements as a text string (Section 3.8.1, above) with respect to encoding, byte order and lead bytes.

3.8.3 Dates

PDF defines a standard date format, which closely follows that of the international standard ASN.1 (Abstract Syntax Notation One), defined in ISO/IEC 8824 (see the Bibliography). A date is a string of the form

(D:*YYYYMMDDHHmmSSOHH*'*mm*')

where

YYYY is the year

MM is the month

DD is the day (01–31)

HH is the hour (00–23)

mm is the minute (00–59)

SS is the second (00–59)

O is the relationship of local time to Universal Time (UT), denoted by one of the characters +, –, or Z (see below)

HH followed by ' is the absolute value of the offset from UT in hours (00–23)

mm followed by ' is the absolute value of the offset from UT in minutes (00–59)

The apostrophe character (') after *HH* and *mm* is part of the syntax. All fields after the year are optional. (The prefix D:, although also optional, is strongly recommended.) The default values for *MM* and *DD* are both 01; all other numerical fields default to zero values. A plus sign (+) as the value of the *O* field signifies that local time is later than UT, a minus sign (–) that local time is earlier than UT, and the letter Z that local time is equal to UT. If no UT information is specified, the relationship of the specified time to UT is considered to be unknown. Whether or not the time zone is known, the rest of the date should be specified in local time.

For example, December 23, 1998, at 7:52 PM, U.S. Pacific Standard Time, is represented by the string

```
D:199812231952−08'00'
```

3.8.4 Rectangles

Rectangles are used to describe locations on a page and bounding boxes for a variety of objects, such as fonts. A rectangle is written as an array of four numbers giving the coordinates of a pair of diagonally opposite corners. Typically, the array takes the form

$[ll_x \; ll_y \; ur_x \; ur_y]$

specifying the lower-left *x*, lower-left *y*, upper-right *x*, and upper-right *y* coordinates of the rectangle, in that order.

Note: *Although rectangles are conventionally specified by their lower-left and upper-right corners, it is acceptable to specify any two diagonally opposite corners. Applications that process PDF should be prepared to normalize such rectangles in situations where specific corners are required.*

3.8.5 Name Trees

A *name tree* serves a similar purpose to a dictionary—associating keys and values—but by different means. A name tree differs from a dictionary in the following important ways:

- Unlike the keys in a dictionary, which are name objects, those in a name tree are strings.
- The keys are ordered.
- The values associated with the keys may be objects of any type. It is recommended, though not required, that stream, dictionary, array, and string objects be specified via indirect object references, and other PDF objects (nulls, numbers, booleans, and names) be specified as direct objects.
- The data structure can represent an arbitrarily large collection of key-value pairs, which can be looked up efficiently without requiring the entire data structure to be read from the PDF file. (In contrast, a dictionary is subject to an implementation limit on the number of entries it can contain.)

A name tree is constructed of *nodes*, each of which is a dictionary object. Table 3.32 shows the entries in a node dictionary. The nodes are of three kinds, depending on the specific entries they contain. The tree always has exactly one *root node*, which contains a single entry: either **Kids** or **Names** but not both. If the root node has a **Names** entry, it is the only node in the tree. If it has a **Kids** entry, then each of the remaining nodes is either an *intermediate node*, containing a **Limits** entry and a **Kids** entry, or a *leaf node*, containing a **Limits** entry and a **Names** entry.

TABLE 3.32 Entries in a name tree node dictionary

KEY	TYPE	VALUE
Kids	array	*(Root and intermediate nodes only; required in intermediate nodes; present in the root node if and only if* ***Names*** *is not present)* An array of indirect references to the immediate children of this node. The children may be intermediate or leaf nodes.
Names	array	*(Root and leaf nodes only; required in leaf nodes; present in the root node if and only if* ***Kids*** *is not present)* An array of the form [key_1 $value_1$ key_2 $value_2$ … key_n $value_n$] where each key_i is a string and the corresponding $value_i$ is the object associated with that key. The keys are sorted in lexical order, as described below.
Limits	array	*(Intermediate and leaf nodes only; required)* An array of two strings, specifying the (lexically) least and greatest keys included in the **Names** array of a leaf node or in the **Names** arrays of any leaf nodes that are descendants of an intermediate node.

The **Kids** entries in the root and intermediate nodes define the tree's structure by identifying the immediate children of each node. The **Names** entries in the leaf (or root) nodes contain the tree's keys and their associated values, arranged in key-value pairs and sorted lexically in ascending order by key. Shorter keys appear before longer ones beginning with the same byte sequence. The encoding of the keys is immaterial as long as it is self-consistent; keys are compared for equality on a simple byte-by-byte basis.

The keys contained within the various nodes' **Names** entries do not overlap; each **Names** entry contains a single contiguous range of all the keys in the tree. In a leaf node, the **Limits** entry specifies the least and greatest keys contained within the node's **Names** entry; in an intermediate node, it specifies the least and greatest keys contained within the **Names** entries of any of that node's descendants. The value associated with a given key can thus be found by walking the tree in order, searching for the leaf node whose **Names** entry contains that key.

Example 3.17 is an abbreviated outline, showing object numbers and nodes, of a name tree that maps the names of all the chemical elements, from actinium to zirconium, to their atomic numbers. Example 3.18 shows the representation of this tree in a PDF file.

Example 3.17 Example of a name tree

```
1: Root node
    2: Intermediate node: Actinium to Gold
        5: Leaf node: Actinium = 25, ..., Astatine = 31
            25: Integer: 89
            ...
            31: Integer: 85
        ...
        11: Leaf node: Gadolinium = 56, ..., Gold = 59
            56: Integer: 64
            ...
            59: Integer: 79
    3: Intermediate node: Hafnium to Protactinium
        12: Leaf node: Hafnium = 60, ..., Hydrogen = 65
            60: Integer: 72
            ...
            65: Integer: 1
        ...
        19: Leaf node: Palladium = 92, ..., Protactinium = 100
            92: Integer: 46
            ...
            100: Integer: 91
    4: Intermediate node: Radium to Zirconium
        20: Leaf node: Radium = 101, ..., Ruthenium = 107
            101: Integer: 89
            ...
            107: Integer: 85
        ...
        24: Leaf node: Xenon = 129, ..., Zirconium = 133
            129: Integer: 54
            ...
            133: Integer: 40
```

Example 3.18

```
1 0 obj
    << /Kids [ 2 0 R                                   % Root node
               3 0 R
               4 0 R
             ]
    >>
endobj

2 0 obj
    << /Limits [(Actinium) (Gold)]                     % Intermediate node
       /Kids [ 5 0 R
               6 0 R
               7 0 R
               8 0 R
               9 0 R
               10 0 R
               11 0 R
             ]
    >>
endobj

3 0 obj
    << /Limits [(Hafnium) (Protactinium)]              % Intermediate node
       /Kids [ 12 0 R
               13 0 R
               14 0 R
               15 0 R
               16 0 R
               17 0 R
               18 0 R
               19 0 R
             ]
    >>
endobj

4 0 obj
    << /Limits [(Radium) (Zirconium)]                  % Intermediate node
       /Kids [ 20 0 R
               21 0 R
               22 0 R
               23 0 R
               24 0 R
             ]
    >>
endobj
```

```
5  0  obj
   << /Limits  [(Actinium)  (Astatine)]                % Leaf node
      /Names  [  (Actinium)  25 0 R
                 (Aluminum)  26 0 R
                 (Americium)  27 0 R
                 (Antimony)  28 0 R
                 (Argon)  29 0 R
                 (Arsenic)  30 0 R
                 (Astatine)  31 0 R
              ]
   >>
endobj

…

24  0  obj
   << /Limits  [(Xenon)  (Zirconium)]                  % Leaf node
      /Names  [  (Xenon)  129 0 R
                 (Ytterbium)  130 0 R
                 (Yttrium)  131 0 R
                 (Zinc)  132 0 R
                 (Zirconium)  133 0 R
              ]
   >>
endobj

25  0  obj
   89                                                  % Atomic number (Actinium)
endobj

…

133  0  obj
   40                                                  % Atomic number (Zirconium)
endobj
```

3.8.6 Number Trees

A *number tree* is similar to a name tree (see Section 3.8.5, “Name Trees”), except that its keys are integers instead of strings, sorted in ascending numerical order. The entries in the leaf (or root) nodes containing the key-value pairs are named **Nums** instead of **Names** as in a name tree. Table 3.33 shows the entries in a number tree’s node dictionaries.

TABLE 3.33 Entries in a number tree node dictionary

KEY	TYPE	VALUE
Kids	array	*(Root and intermediate nodes only; required in intermediate nodes; present in the root node if and only if **Nums** is not present)* An array of indirect references to the immediate children of this node. The children may be intermediate or leaf nodes.
Nums	array	*(Root and leaf nodes only; required in leaf nodes; present in the root node if and only if **Kids** is not present)* An array of the form $[key_1\ value_1\ key_2\ value_2\ \ldots\ key_n\ value_n]$ where each key_i is an integer and the corresponding $value_i$ is the object associated with that key. The keys are sorted in numerical order, analogously to the arrangement of keys in a name tree as described in Section 3.8.5, “Name Trees.”
Limits	array	*(Intermediate and leaf nodes only; required)* An array of two integers, specifying the (numerically) least and greatest keys included in the **Nums** array of a leaf node or in the **Nums** arrays of any leaf nodes that are descendants of an intermediate node.

3.9 Functions

PDF is not a programming language, and a PDF file is not a program; however, PDF does provide several types of *function object (PDF 1.2)* that represent parameterized classes of functions, including mathematical formulas and sampled representations with arbitrary resolution. Functions are used in various ways in PDF, including device-dependent rasterization information for high-quality printing (halftone spot functions and transfer functions), color transform functions for certain color spaces, and specification of colors as a function of position for smooth shadings.

Functions in PDF represent static, self-contained numerical transformations. A function to add two numbers has two input values and one output value:

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

Similarly, a function that computes the arithmetic and geometric mean of two numbers 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:

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

In PDF functions, all the input values and all the output values are numbers, and functions have no side effects.

Each function definition includes a *domain*, the set of legal values for the input. Some types of functions also define a *range*, the set of legal values for the output. Input values passed to the function are clipped to the domain, and output 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 input value 6, that value is replaced with the nearest value in the defined domain, 1, before the function is evaluated; the resulting output value 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 input values –6 and 4 are passed to the function (and are within its domain), then the output value produced by the function, –14, is replaced with 0, the nearest value in the defined range.

A function object may be a dictionary or a stream, depending on the type of function; the term *function dictionary* will be used generically in this section to refer to either a dictionary object or the dictionary portion of a stream object. A function dictionary specifies the function's representation, the set of attributes that parameterize that representation, and the additional data needed by that representation. Four types of function are available, as indicated by the dictionary's **FunctionType** entry:

- *(PDF 1.2)* A *sampled function* (type 0) uses a table of *sample values* to define the function. Various techniques are used to interpolate values between the sample values.
- *(PDF 1.3)* An *exponential interpolation function* (type 2) defines a set of coefficients for an exponential function.

- *(PDF 1.3)* A *stitching function* (type 3) is a combination of other functions, partitioned across a domain.
- *(PDF 1.3)* A *PostScript calculator function* (type 4) uses operators from the PostScript language to describe an arithmetic expression.

All function dictionaries share the entries listed in Table 3.34.

TABLE 3.34 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 4 PostScript calculator 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 $\textbf{Domain}_{2i} \leq x_i \leq \textbf{Domain}_{2i+1}$. Input values outside the declared domain are clipped to the nearest boundary value.
Range	array	*(Required for type 0 and type 4 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 $\textbf{Range}_{2j} \leq y_j \leq \textbf{Range}_{2j+1}$. Output values outside the declared range are clipped to the nearest boundary value. If this entry is absent, no clipping is done.

In addition, each type of function dictionary must include entries 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 and type 4 functions), the **Range** entry is required. The dimensionality of the function implied by the **Domain** and **Range** entries must be consistent with that implied by other attributes of the function.

3.9.1 Type 0 (Sampled) Functions

Type 0 functions use a sequence of *sample values* (contained in a stream) 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 functions with high dimensionality multiplies rapidly unless the sampling resolution is very low. Also, the process of multilinear interpolation becomes computationally intensive if the number of inputs m is greater than 2. The multidimensional spline interpolation is even more computationally intensive.

In addition to the entries in Table 3.34, a type 0 function dictionary includes those shown in Table 3.35.

The **Domain**, **Encode**, and **Size** entries 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:

$$\begin{aligned} y &= \text{Interpolate}(x, x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}) \\ &= y_{\text{min}} + \left((x - x_{\text{min}}) \times \frac{y_{\text{max}} - y_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \right) \end{aligned}$$

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

TABLE 3.35 Additional entries specific to a type 0 function dictionary

KEY	TYPE	VALUE
Size	array	*(Required)* An array of *m* positive integers specifying the number of samples in each input dimension of the sample table.
BitsPerSample	integer	*(Required)* The number of bits used to represent each sample. (If the function has multiple output values, each one occupies **BitsPerSample** bits.) Valid values are 1, 2, 4, 8, 12, 16, 24, and 32.
Order	integer	*(Optional)* The order of interpolation between samples. Valid values are 1 and 3, specifying linear and cubic spline interpolation, respectively. (See implementation note 33 in Appendix H.) Default value: 1.
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 ($\textbf{Size}_0 - 1$) 0 ($\textbf{Size}_1 - 1$) …].
Decode	array	*(Optional)* An array of $2 \times n$ numbers specifying the linear mapping of sample values into the range appropriate for the function's output values. Default value: same as the value of **Range**.
other stream attributes	(various)	*(Optional)* Other attributes of the stream that provides the sample values, as appropriate (see Table 3.4 on page 38).

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

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

That value is encoded:

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

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

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

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:

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

Finally, each decoded value is clipped to the range:

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

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 stream data must be long enough to contain the entire sample array, as indicated by **Size**, **Range**, and **BitsPerSample**; see "Stream Extent" on page 37.

Example 3.19 illustrates a sampled function with 4-bit samples in an array containing 21 columns and 31 rows (651 values). The function takes two arguments, x and y, in the domain [−1.0 1.0], and returns one value, z, in that same range. 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 [−1.0 1.0] 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 (−1.0, −1.0) in the high-order 4 bits and the sample for the point (−0.9, −1.0) in the low-order 4 bits.

Example 3.19

```
14 0 obj
    << /FunctionType 0
        /Domain [−1.0 1.0  −1.0 1.0]
        /Size [21 31]
        /Encode [0 20  0 30]
        /BitsPerSample 4
        /Range [−1.0 1.0]
        /Decode [−1.0 1.0]
        /Length …
        /Filter …
    >>
stream
…651 sample values…
endstream
endobj
```

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 [−1.0 1.0] and **BitsPerSample** is 4, the usual value of **Decode** would be [−1.0 1.0] and the sample values would be integers in the interval [0 15] (as shown in Figure 3.7). But if these values were used, the midpoint of the range, 0.0, 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 [−1.0 +1.1429] (1.1429 being approximately equal to 16 ÷ 14) and the sample values supplied were in the interval [0 14], then the desired effective range of [−1.0 1.0] would be achieved, and the range value 0.0 would be represented by the sample value 7.

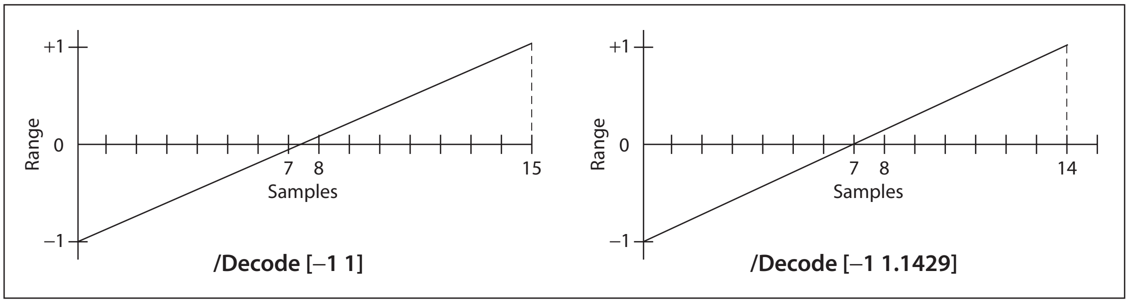


FIGURE 3.7 *Mapping with the **Decode** array*

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.

3.9.2 Type 2 (Exponential Interpolation) Functions

Type 2 functions *(PDF 1.3)* 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 3.34 on page 140, a type 2 function dictionary includes those; listed in Table 3.36. (See implementation note 34 in Appendix H.)

TABLE 3.36 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 $x = 0.0$ (hence the "0" in the name). Default value: [0.0].
C1	array	*(Optional)* An array of *n* numbers defining the function result when $x = 1.0$ (hence the "1" in the name). Default value: [1.0].
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 $0 \le j < n$.

Values of **Domain** must constrain x in such a way that if **N** is not an integer, all values of x must be nonnegative, and if **N** is negative, no value of x may be zero. Typically, **Domain** will be declared as [0.0 1.0], and **N** will be a positive number. The **Range** attribute is optional and can be used to clip the output to a desired range. Note that when **N** is 1, the function performs a linear interpolation between **C0** and **C1**; this can also be expressed as a sampled function (type 0).

3.9.3 Type 3 (Stitching) Functions

Type 3 functions *(PDF 1.3)* 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, [$\textbf{Domain}_0$ $\textbf{Domain}_1$].

In addition to the entries in Table 3.34 on page 140, a type 3 function dictionary includes those listed in Table 3.37. (See implementation note 35 in Appendix H.)

TABLE 3.37 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 domain defined 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, $m = 1$), and $\textbf{Domain}_0$ must be strictly less than $\textbf{Domain}_1$ unless $k = 1$. The 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 relationships (with exceptions as noted below):

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

The **Bounds** array describes a series of half-open intervals, closed on the left and open on the right (except the last, which is closed on the right as well). The value of the **Functions** entry is an array of k functions. The first function applies to x values in the first subdomain, $\textbf{Domain}_0 \leq x < \textbf{Bounds}_0$; the second function applies to x values in the second subdomain, $\textbf{Bounds}_0 \leq x < \textbf{Bounds}_1$; and so on. The last function applies to x values in the last subdomain, which includes the upper bound: $\textbf{Bounds}_{k-2} \leq x \leq \textbf{Domain}_1$. 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, $\textbf{Domain}_0 \leq x \leq \textbf{Domain}_1$.

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

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

for $0 \leq i < k$. In this equation, $\textbf{Bounds}_{-1}$ means $\textbf{Domain}_0$, and $\textbf{Bounds}_{k-1}$ means $\textbf{Domain}_1$. If the last bound, $\textbf{Bounds}_{k-2}$, is equal to $\textbf{Domain}_1$, then x' is defined to be $\textbf{Encode}_{2i}$.

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. (Shading patterns are discussed in Section 4.6.3, "Shading Patterns.") The same effect could be achieved by creating a separate shading dictionary 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, type 3 functions provide a general mechanism for inverting the domains of 1-input functions. For example, consider a function f with a **Domain** of [0.0 1.0], and a stitching function g with a **Domain** of [0.0 1.0], a **Functions** array containing f, and an **Encode** array of [1.0 0.0]. In effect, $g(x) = f(1 - x)$.

3.9.4 Type 4 (PostScript Calculator) Functions

A type 4 function *(PDF 1.3)*, also called a *PostScript calculator function*, is represented as a stream containing code written in a small subset of the PostScript language. While any function can be sampled (in a type 0 PDF function) and others can be described with exponential functions (type 2 in PDF), type 4 functions offer greater flexibility and potentially greater accuracy. For example, a tint transformation function for a hexachrome (six-component) **DeviceN** color space with an alternate color space of **DeviceCMYK** (see “DeviceN Color Spaces” on page 238) requires a 6-in, 4-out function. If such a function were sampled with m values for each input variable, the number of samples, $4 \times m^6$, could be prohibitively large. In practice, such functions are often written as short, simple PostScript functions. (See implementation note 36 in Appendix H.)

Type 4 functions also make it possible to include a wide variety of halftone spot functions without the loss of accuracy that comes from sampling, and without adding to the list of predefined spot functions (see Section 6.4.2, “Spot Functions”). All of the predefined spot functions can be written as type 4 functions.

The language that can be used in a type 4 function contains expressions involving integers, real numbers, and boolean values only. There are no composite data structures such as strings or arrays, no procedures, and no variables or names. Table 3.38 lists the operators that can be used in this type of function. (For more information on these operators, see Appendix B of the *PostScript Language Reference*, Third Edition.) Although the semantics are those of the corresponding PostScript operators, a PostScript interpreter is not required.

TABLE 3.38 Operators in type 4 functions

OPERATOR TYPE	OPERATORS				
Arithmetic operators	**abs**	**cvi**	**floor**	**mod**	**sin**
	add	**cvr**	**idiv**	**mul**	**sqrt**
	atan	**div**	**ln**	**neg**	**sub**
	ceiling	**exp**	**log**	**round**	**truncate**
	cos				
Relational, boolean,	**and**	**false**	**le**	**not**	**true**
and bitwise operators	**bitshift**	**ge**	**lt**	**or**	**xor**
	eq	**gt**	**ne**		
Conditional operators	**if**	**ifelse**			
Stack operators	**copy**	**exch**	**pop**		
	dup	**index**	**roll**		

The operand syntax for type 4 functions follows PDF conventions rather than PostScript conventions. The entire code stream defining the function is enclosed in braces { }. Braces also delimit expressions that are executed conditionally by the **if** and **ifelse** operators:

> *boolean* {*expression*} if
> *boolean* {*expression*$_1$} {*expression*$_2$} ifelse

Note that this is a purely syntactic construct; unlike in PostScript, no "procedure objects" are involved.

A type 4 function dictionary includes the entries in Table 3.34 on page 140, as well as other stream attributes as appropriate (see Table 3.4 on page 38). Example 3.20 shows a type 4 function equivalent to the predefined spot function **DoubleDot** (see Section 6.4.2, "Spot Functions").

Example 3.20

```
10 0 obj
   << /FunctionType 4
      /Domain [−1.0 1.0 −1.0 1.0]
      /Range [−1.0 1.0]
      /Length 71
   >>
stream
   { 360 mul sin
     2 div
     exch 360 mul sin
     2 div
     add
   }
endstream
endobj
```

The **Domain** and **Range** entries are both required. The input variables constitute the initial operand stack; the items remaining on the operand stack after execution of the function are the output variables. It is an error for the number of remaining operands to differ from the number of output variables specified by **Range**, or for any of them to be objects other than numbers.

Implementations of type 4 functions must provide a stack with room for at least 100 entries. No implementation is required to provide a larger stack, and it is an error to overflow the stack.

Although any integers or real numbers that may appear in the stream fall under the same implementation limits (defined in Appendix C) as in other contexts, the *intermediate* results in type 4 function computations do not. An implementation may use a representation that exceeds those limits. Operations on real numbers, for example, might use single-precision or double-precision floating-point numbers. (See implementation note 37 in Appendix H.)

Errors in Type 4 Functions

The code that reads a type 4 function (analogous to the PostScript *scanner*) must detect and report syntax errors. It may also be able to detect some errors that will occur when the function is used, although this is not always possible. Any errors detected by the scanner are considered to be errors in the PDF file itself and are handled like other errors in the file.

The code that executes a type 4 function (analogous to the PostScript *interpreter*) must detect and report errors. PDF does not define a representation for the errors; those details are provided by the application that processes the PDF file. The following types of errors can occur (among others):

- Stack overflow
- Stack underflow
- A type error (for example, applying **not** to a real number)
- A range error (for example, applying **sqrt** to a negative number)
- An undefined result (for example, dividing by 0)

3.10 File Specifications

A PDF file can refer to the contents of another file by using a *file specification (PDF 1.1)*, which can take either of two forms. A *simple* file specification gives just the name of the target file in a standard format, independent of the naming conventions of any particular file system; a *full* file specification includes information related to one or more specific file systems. A simple file specification may take the form of either a string or a dictionary; a full file specification can only be represented as a dictionary.

Although the file designated by a file specification is normally external to the PDF file referring to it, PDF 1.3 permits a copy of the external file to be embedded within the PDF file itself, allowing its contents to be stored or transmitted along with the PDF file. However, embedding a file does not change the presumption that it is external to the PDF file. Consequently, in order for the PDF file to be processed correctly, it may be necessary to copy the embedded files it contains back into a local file system.

3.10.1 File Specification Strings

The standard format for representing a simple file specification in string form divides the string into component substrings separated by the slash character (/). The slash is a generic component separator that is mapped to the appropriate platform-specific separator when generating a platform-dependent file name. Any of the components may be empty. If a component contains one or more literal slashes, each must be preceded by a backslash (\), which in turn must be preceded by another backslash to indicate that it is part of the string and not an escape character. For example, the string

```
(in\\/out)
```

represents the file name

```
in/out
```

The backslashes are removed in processing the string; they are needed only to distinguish the component values from the component separators. The component substrings are stored as bytes and are passed to the operating system without interpretation or conversion of any sort.

Absolute and Relative File Specifications

A simple file specification that begins with a slash is an *absolute* file specification. The last component is the file name; the preceding components specify its context. In some file specifications, the file name may be empty; for example, URL (uniform resource locator) specifications can specify directories instead of files. A file specification that does not begin with a slash is a *relative* file specification giving the location of the file relative to that of the PDF file containing it.

In the case of a URL-based file system, the rules of Internet RFC 1808, *Relative Uniform Resource Locators* (see the Bibliography), are used to compute an absolute URL from a relative file specification and the specification of the PDF file. Prior to this process, the relative file specification is converted to a relative URL by using the escape mechanism of RFC 1738, *Uniform Resource Locators*, to represent any bytes that would be either "unsafe" according to RFC 1738 or not representable in 7-bit U.S. ASCII. In addition, such URL-based relative file specifications are limited to paths as defined in RFC 1808; the scheme, network location/login, fragment identifier, query information, and parameter sections are not allowed.

In the case of other file systems, a relative file specification is converted to an absolute file specification by removing the file name component from the specification of the containing PDF file and appending the relative file specification in its place. For example, the relative file specification

```
ArtFiles/Figure1.pdf
```

appearing in a PDF file whose specification is

```
/HardDisk/PDFDocuments/AnnualReport/Summary.pdf
```

yields the absolute specification

```
/HardDisk/PDFDocuments/AnnualReport/ArtFiles/Figure1.pdf
```

The special component .. (two periods) can be used in a relative file specification to move up a level in the file system hierarchy. When the component immediately preceding .. is not another .., the two cancel each other; both are eliminated from the file specification and the process is repeated. Thus in the example above, the relative file specification

```
../../ArtFiles/Figure1.pdf
```

would yield the absolute specification

```
/HardDisk/ArtFiles/Figure1.pdf
```

Conversion to Platform-Dependent File Names

The conversion of a file specification into a platform-dependent file name depends on the specific file naming conventions of each platform. For example:

- For DOS, the initial component is either a physical or logical drive identifier or a network resource name as returned by the Microsoft Windows function WNetGetConnection, and is followed by a colon (:). A network resource name is constructed from the first two components; the first component is the server name and the second is the share name (volume name). All components are then separated by backslashes. It is possible to specify an absolute DOS path without a drive by making the first component empty. (Empty components are ignored by other platforms.)
- For Mac OS, all components are separated by colons (:).
- For UNIX, all components are separated by slashes (/). An initial slash, if present, is preserved.

Strings used to specify a file name are interpreted in the standard encoding for the platform on which the document is being viewed. Table 3.39 shows examples of file specifications on the most common platforms.

TABLE 3.39 Examples of file specifications

SYSTEM	SYSTEM-DEPENDENT PATHS	WRITTEN FORM
DOS	\pdfdocs\spec.pdf (no drive)	(//pdfdocs/spec.pdf)
	r:\pdfdocs\spec.pdf	(/r/pdfdocs/spec.pdf)
	pclib/eng:\pdfdocs\spec.pdf	(/pclib/eng/pdfdocs/spec.pdf)
Mac OS	Mac HD:PDFDocs:spec.pdf	(/Mac HD/PDFDocs/spec.pdf)
UNIX	/user/fred/pdfdocs/spec.pdf	(/user/fred/pdfdocs/spec.pdf)
	pdfdocs/spec.pdf (relative)	(pdfdocs/spec.pdf)

When creating documents that are to be viewed on multiple platforms, care must be taken to ensure file name compatibility. Only a subset of the U.S. ASCII character set should be used in file specifications: the uppercase alphabetic characters (A–Z), the numeric characters (0–9), and the underscore (_). The period (.) has special meaning in DOS and Windows file names, and as the first character in a Mac OS pathname. In file specifications, the period should be used only to separate a base file name from a file extension.

Some file systems are case-insensitive, so names within a directory should remain distinguishable if lowercase letters are changed to uppercase or vice versa. On DOS and Windows 3.1 systems and on some CD-ROM file systems, file names are limited to 8 characters plus a 3-character extension. File system software typically converts long names to short names by retaining the first 6 or 7 characters of the file name and the first 3 characters after the last period, if any. Since characters beyond the sixth or seventh are often converted to other values unrelated to the original value, file names must be distinguishable from the first 6 characters.

Multiple-Byte Strings in File Specifications

In PDF 1.2 or higher, a file specification may contain multiple-byte character codes, represented in hexadecimal form between angle brackets (< and >). Since the slash character `<2F>` is used as a component delimiter and the backslash `<5C>` is used as an escape character, any occurrence of either of these bytes in a multiple-byte character must be preceded by the ASCII code for the backslash character. For example, a file name containing the 2-byte character code `<89 5C>` must write it as `<89 5C 5C>`. When the viewer application encounters this sequence of bytes in a file name, it replaces the sequence with the original 2-byte code.

3.10.2 File Specification Dictionaries

The dictionary form of file specification provides more flexibility than the string form, allowing different files to be specified for different file systems or platforms, or for file systems other than the standard ones (DOS/Windows, Mac OS, and UNIX). Table 3.40 shows the entries in a file specification dictionary. Viewer applications running on a particular platform should use the appropriate platform-specific entry (**DOS**, **Mac**, or **Unix**) if available. If the required platform-specific entry is not present and there is no file system entry (**FS**), the generic **F** entry should be used as a simple file specification.

TABLE 3.40 Entries in a file specification dictionary

KEY	TYPE	VALUE
Type	name	*(Required if an* ***EF*** *or* ***RF*** *entry is present; recommended always)* The type of PDF object that this dictionary describes; must be **Filespec** for a file specification dictionary (see implementation note 38 in Appendix H).

KEY	TYPE	VALUE
FS	name	*(Optional)* The name of the file system to be used to interpret this file specification. If this entry is present, all other entries in the dictionary are interpreted by the designated file system. PDF defines only one standard file system, **URL** (see Section 3.10.4, "URL Specifications"); a viewer application or plug-in extension can register a different one (see Appendix E). Note that this entry is independent of the **F**, **DOS**, **Mac**, and **Unix** entries.
F	string	*(Required if the* ***DOS****,* ***Mac****, and* ***Unix*** *entries are all absent)* A file specification string of the form described in Section 3.10.1, "File Specification Strings," or (if the file system is **URL**) a uniform resource locator, as described in Section 3.10.4, "URL Specifications."
DOS	string	*(Optional)* A file specification string (see Section 3.10.1, "File Specification Strings") representing a DOS file name.
Mac	string	*(Optional)* A file specification string (see Section 3.10.1, "File Specification Strings") representing a Mac OS file name.
Unix	string	*(Optional)* A file specification string (see Section 3.10.1, "File Specification Strings") representing a UNIX file name.
ID	array	*(Optional)* An array of two strings constituting a file identifier (see Section 10.3, "File Identifiers") that is also included in the referenced file. The use of this entry improves a viewer application's chances of finding the intended file and allows it to warn the user if the file has changed since the link was made.
V	boolean	*(Optional; PDF 1.2)* A flag indicating whether the file referenced by the file specification is *volatile* (changes frequently with time). If the value is **true**, viewer applications should never cache a copy of the file. For example, a movie annotation referencing a URL to a live video camera could set this flag to **true**, notifying the application that it should reacquire the movie each time it is played. Default value: **false**.
EF	dictionary	*(Required if* ***RF*** *is present; PDF 1.3)* A dictionary containing a subset of the keys **F**, **DOS**, **Mac**, and **Unix**, corresponding to the entries by those names in the file specification dictionary. The value of each such key is an embedded file stream (see Section 3.10.3, "Embedded File Streams") containing the corresponding file. If this entry is present, the **Type** entry is required and the file specification dictionary must be indirectly referenced.

KEY	TYPE	VALUE
RF	dictionary	*(Optional; PDF 1.3)* A dictionary with the same structure as the **EF** dictionary, which must also be present. Each key in the **RF** dictionary must also be present in the **EF** dictionary. Each value is a related files array (see “Related Files Arrays” on page 158) identifying files that are related to the corresponding file in the **EF** dictionary. If this entry is present, the **Type** entry is required and the file specification dictionary must be indirectly referenced.

3.10.3 Embedded File Streams

File specifications ordinarily refer to files external to the PDF file in which they occur. To preserve the integrity of the PDF file, this requires that all external files it refers to must accompany it when it is archived or transmitted. *Embedded file streams (PDF 1.3)* address this problem by allowing the contents of the referenced files to be embedded directly within the body of the PDF file itself. For example, if the file contains OPI (Open Prepress Interface) dictionaries that refer to externally stored high-resolution images (see Section 10.10.6, “Open Prepress Interface (OPI)”), the image data can be incorporated into the PDF file with embedded file streams. This makes the PDF file a self-contained unit that can be stored or transmitted as a single entity. (The embedded files are included purely for convenience, and need not be directly processed by any PDF consumer application.)

An embedded file stream can be included in a PDF document in the following ways:

- Any file specification dictionary in the document may have an **EF** entry that specifies an embedded file stream. This method still requires a file specification to be provided, associating a location in the file system with the stream data.
- A document as whole can specify embedded file streams through the **EmbeddedFiles** entry (*PDF 1.4*) in the PDF document’s name dictionary (see Section 3.6.3, “Name Dictionary”). The associated name tree maps name strings to file specifications that in turn refer to embedded file streams through their **EF** entries. (See implementation note 38 in Appendix H.)

Note: *The relationship between the name string provided in the name dictionary and the associated embedded file stream is an artificial one for data management purposes. This allows embedded files to be easily identified by the author of the document, in much the same way that the JavaScript name tree associates name strings with document-level JavaScript actions (see “JavaScript Actions” on page 645).*

The stream dictionary describing an embedded file contains the standard entries for any stream, such as **Length** and **Filter** (see Table 3.4 on page 38), as well as the additional entries shown in Table 3.41.

TABLE 3.41 Additional entries in an embedded file stream dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **EmbeddedFile** for an embedded file stream.
Subtype	name	*(Optional)* The subtype of the embedded file. The value of this entry must be a first-class name, as defined in Appendix E. Names without a registered prefix must conform to the MIME media type names defined in Internet RFC 2046, *Multipurpose Internet Mail Extensions (MIME), Part Two: Media Types* (see the Bibliography), with the provision that characters not allowed in names must use the 2-character hexadecimal code format described in Section 3.2.4, "Name Objects."
Params	dictionary	*(Optional)* An *embedded file parameter dictionary* containing additional, file-specific information (see Table 3.42).

TABLE 3.42 Entries in an embedded file parameter dictionary

KEY	TYPE	VALUE
Size	integer	*(Optional)* The size of the embedded file, in bytes.
CreationDate	date	*(Optional)* The date and time when the embedded file was created.
ModDate	date	*(Optional)* The date and time when the embedded file was last modified.
Mac	dictionary	*(Optional)* A subdictionary containing additional information specific to Mac OS files (see Table 3.43).
CheckSum	string	*(Optional)* A 16-byte string that is the checksum of the bytes of the uncompressed embedded file. The checksum is calculated by applying the standard MD5 message-digest algorithm (described in Internet RFC 1321, *The MD5 Message-Digest Algorithm*; see the Bibliography) to the bytes of the embedded file stream.

For Mac OS files, the **Mac** entry in the embedded file parameter dictionary holds a further subdictionary containing Mac OS–specific file information. Table 3.43 shows the contents of this subdictionary.

TABLE 3.43 Entries in a Mac OS file information dictionary

KEY	TYPE	VALUE
Subtype	string	*(Optional)* The embedded file's file type.
Creator	string	*(Optional)* The embedded file's creator signature.
ResFork	stream	*(Optional)* The binary contents of the embedded file's resource fork.

Related Files Arrays

In some circumstances, a PDF file can refer to a group of related files, such as the set of five files that make up a DCS 1.0 color-separated image. The file specification explicitly names only one of the files; the rest are identified by some systematic variation of that file name (such as by altering the extension). When such a file is to be embedded in a PDF file, the related files must be embedded as well. This is accomplished by including a *related files array (PDF 1.3)* as the value of the **RF** entry in the file specification dictionary. The array has $2 \times n$ elements, which are paired in the form

$$\begin{array}{l} [\ string_1\ \ stream_1 \\ \quad string_2\ \ stream_2 \\ \quad \ldots \\ \quad string_n\ \ stream_n \\ ] \end{array}$$

The first element of each pair is a string giving the name of one of the related files; the second element is an embedded file stream holding the file's contents.

In Example 3.21, objects 21, 31, and 41 are embedded file streams containing the DOS file SUNSET.EPS, the Mac OS file Sunset.eps, and the UNIX file Sunset.eps, respectively. The file specification dictionary's **RF** entry specifies an array, object 30, identifying a set of embedded files related to the Mac OS file, forming a DCS 1.0 set. The example shows only the first two embedded file streams in the set; an actual PDF file would of course include all of them.

Example 3.21

```
10 0 obj                                % File specification dictionary
    << /Type /Filespec
        /DOS (SUNSET.EPS)
        /Mac (Sunset.eps)               % Name of Mac OS file
        /Unix (Sunset.eps)
        /EF << /DOS  21 0 R
                /Mac  31 0 R            % Embedded Mac OS file
                /Unix 41 0 R
            >>
        /RF << /Mac 30 0 R >>           % Related files array for Mac OS file
    >>
endobj

30 0 obj                                % Related files array for Mac OS file
    [ (Sunset.eps) 31 0 R               % Includes file Sunset.eps itself
      (Sunset.C) 32 0 R
      (Sunset.M) 33 0 R
      (Sunset.Y) 34 0 R
      (Sunset.K) 35 0 R
    ]
endobj

31 0 obj                                % Embedded file stream for Mac OS file
    << /Type /EmbeddedFile              %     Sunset.eps
        /Length …
        /Filter …
    >>
stream
…Data for Sunset.eps…
endstream
endobj

32 0 obj                                % Embedded file stream for related file
    << /Type /EmbeddedFile              %     Sunset.C
        /Length …
        /Filter …
    >>
stream
…Data for Sunset.C…
endstream
endobj
```

3.10.4 URL Specifications

When the **FS** entry in a file specification dictionary has the value **URL**, the value of the **F** entry in that dictionary is not a file specification string, but a uniform resource locator (URL) of the form defined in Internet RFC 1738, *Uniform Resource Locators* (see the Bibliography). Example 3.22 shows a URL specification.

Example 3.22

```
<< /FS /URL
   /F (ftp://www.beatles.com/Movies/AbbeyRoad.mov)
>>
```

The URL must adhere to the character-encoding requirements specified in RFC 1738. Because 7-bit U.S. ASCII is a strict subset of **PDFDocEncoding**, this value may also be considered to be in that encoding.

3.10.5 Maintenance of File Specifications

The techniques described in this section can be used to maintain the integrity of the file specifications within a PDF file during operations such as the following:

- Updating the relevant file specification when a referenced file is renamed
- Determining the complete collection of files that must be copied to a mirror site
- When creating new links to external files, discovering existing file specifications that refer to the same files and sharing them
- Finding the file specifications associated with embedded files to be packed or unpacked

It is not possible, in general, to find all file specification strings in a PDF file, because there is no way to determine whether a given string is a file specification string. It is possible, however, to find all file specification *dictionaries*, provided that they meet the following conditions:

- They are indirect objects.
- They contain a **Type** entry whose value is the name **Filespec**.

An application can then locate all of the file specification dictionaries by traversing the PDF file's cross-reference table (see Section 3.4.3, "Cross-Reference Table") and finding all dictionaries with **Type** keys whose value is **Filespec**. For this reason, it is highly recommended that all file specifications be expressed in dictionary form and meet the conditions stated above. Note that any file specification dictionary specifying embedded files (that is, one that contains an **EF** entry) *must* satisfy these conditions (see Table 3.40 on page 154).

Note: *It may not be possible to locate file specification dictionaries that are direct objects, since they are neither self-typed nor necessarily reachable via any standard path of object references.*

Files may be embedded in a PDF file either directly, using the **EF** entry in a file specification dictionary, or indirectly, using related files arrays specified in the **RF** entry. If a file is embedded indirectly, its name is given by the string that precedes the embedded file stream in the related files array; if it is embedded directly, its name is obtained from the value of the corresponding entry in the file specification dictionary. In Example 3.21 on page 159, for instance, the **EF** dictionary has a **DOS** entry identifying object number 21 as an embedded file stream; the name of the embedded DOS file, SUNSET.EPS, is given by the **DOS** entry in the file specification dictionary.

A given external file may be referenced from more than one file specification. Therefore, when embedding a file with a given name, it is necessary to check for other occurrences of the same name as the value associated with the corresponding key in other file specification dictionaries. This requires finding all embeddable file specifications and, for each matching key, checking for both of the following conditions:

- The string value associated with the key matches the name of the file being embedded.
- A value has not already been embedded for the file specification. (If there is already a corresponding key in the **EF** dictionary, then a file has already been embedded for that use of the file name.)

Note that there is no requirement that the files associated with a given file name be unique. The same file name, such as readme.txt, may be associated with different embedded files in distinct file specifications.

CHAPTER 4

Graphics

THE GRAPHICS OPERATORS used in PDF content streams describe the appearance of pages that are to be reproduced on a raster output device. The facilities described in this chapter are intended for both printer and display applications.

The graphics operators form six main groups:

- *Graphics state operators* manipulate the data structure called the *graphics state*, the global framework within which the other graphics operators execute. The graphics state includes the *current transformation matrix* (CTM), which maps user space coordinates used within a PDF content stream into output device coordinates. It also includes the *current color*, the *current clipping path*, and many other parameters that are implicit operands of the painting operators.
- *Path construction operators* specify *paths*, which define shapes, line trajectories, and regions of various sorts. They include operators for beginning a new path, adding line segments and curves to it, and closing it.
- *Path-painting operators* fill a path with a color, paint a stroke along it, or use it as a clipping boundary.
- *Other painting operators* paint certain self-describing graphics objects. These include sampled images, geometrically defined shadings, and entire content streams that in turn contain sequences of graphics operators.
- *Text operators* select and show *character glyphs* from *fonts* (descriptions of typefaces for representing text characters). Because PDF treats glyphs as general graphical shapes, many of the text operators could be grouped with the graphics state or painting operators. However, the data structures and mechanisms for dealing with glyph and font descriptions are sufficiently specialized that Chapter 5 focuses on them.

- *Marked-content operators* associate higher-level logical information with objects in the content stream. This information does not affect the rendered appearance of the content (although it may determine if the content should be presented at all; see Section 4.10, "Optional Content"); it is useful to applications that use PDF for document interchange. Marked content is described in Section 10.5, "Marked Content."

This chapter presents general information about device-independent graphics in PDF: how a PDF content stream describes the abstract appearance of a page. *Rendering*—the device-dependent part of graphics—is covered in Chapter 6. The Bibliography lists a number of books that give details of these computer graphics concepts and their implementation.

4.1 Graphics Objects

As discussed in Section 3.7.1, "Content Streams," the data in a content stream is interpreted as a sequence of *operators* and their *operands*, expressed as basic data objects according to standard PDF syntax. A content stream can describe the appearance of a page, or it can be treated as a graphical element in certain other contexts.

The operands and operators are written sequentially using postfix notation. Although this notation resembles the sequential execution model of the PostScript language, a PDF content stream is not a program to be interpreted; rather, it is a static description of a sequence of *graphics objects*. There are specific rules, described below, for writing the operands and operators that describe a graphics object.

PDF provides five types of graphics object:

- A *path object* is an arbitrary shape made up of straight lines, rectangles, and cubic Bézier curves. A path may intersect itself and may have disconnected sections and holes. A path object ends with one or more painting operators that specify whether the path is stroked, filled, used as a clipping boundary, or some combination of these operations.
- A *text object* consists of one or more character strings that identify sequences of glyphs to be painted. Like a path, text can be stroked, filled, or used as a clipping boundary.

- An *external object (XObject)* is an object defined outside the content stream and referenced as a named resource (see Section 3.7.2, “Resource Dictionaries”). The interpretation of an XObject depends on its type. An *image XObject* defines a rectangular array of color samples to be painted; a *form XObject* is an entire content stream to be treated as a single graphics object. Specialized types of form XObject are used to import content from one PDF file into another *(reference XObjects)* and to group graphical elements together as a unit for various purposes *(group XObjects)*. In particular, the latter are used to define *transparency groups* for use in the transparent imaging model (*transparency group XObjects*, discussed in detail in Chapter 7). There is also a *PostScript XObject*, whose use is discouraged.
- An *inline image object* is a means of expressing the data for a small image directly within the content stream, using a special syntax.
- A *shading object* describes a geometric shape whose color is an arbitrary function of position within the shape. (A shading can also be treated as a color when painting other graphics objects; it is not considered to be a separate graphics object in that case.)

PDF 1.3 and earlier versions use an *opaque imaging model* in which each graphics object is painted in sequence, completely obscuring any previous marks it may overlay on the page. PDF 1.4 introduces a new *transparent imaging model* in which objects can be less than fully opaque, allowing previously painted marks to show through. Each object is painted on the page with a specified *opacity*, which may be constant at every point within the object’s shape or may vary from point to point. The previously existing contents of the page form a *backdrop* with which the new object is *composited*, producing results that combine the colors of the object and backdrop according to their respective opacity characteristics. The objects at any given point on the page can be thought of as forming a *transparency stack*, where the stacking order is defined to be the order in which the objects are specified, bottommost object first. All objects in the stack can potentially contribute to the result, depending on their colors, shapes, and opacities.

PDF’s graphics parameters are so arranged that objects are painted by default with full opacity, reducing the behavior of the transparent imaging model to that of the opaque model. Accordingly, the material in this chapter applies to both the opaque and transparent models except where explicitly stated otherwise; the transparent model is described in its full generality in Chapter 7.

Although the painting behavior described above is often attributed to individual operators making up an object, it is always the object as a whole that is painted. Figure 4.1 shows the ordering rules for the operations that define graphics objects. Some operations are permitted only in certain types of graphics object or in the intervals between graphics objects (called the *page description level* in the figure). Every content stream begins at the page description level, where changes can be made to the graphics state, such as colors and text attributes, as discussed in the following sections.

In the figure, arrows indicate the operators that mark the beginning or end of each type of graphics object. Some operators are identified individually, others by general category. Table 4.1 summarizes these categories for all PDF operators. For

TABLE 4.1 Operator categories

CATEGORY	OPERATORS	TABLE	PAGE
General graphics state	**w**, **J**, **j**, **M**, **d**, **ri**, **i**, **gs**	4.7	189
Special graphics state	**q**, **Q**, **cm**	4.7	189
Path construction	**m**, **l**, **c**, **v**, **y**, **h**, **re**	4.9	196
Path painting	**S**, **s**, **f**, **F**, **f***, **B**, **B***, **b**, **b***, **n**	4.10	200
Clipping paths	**W**, **W***	4.11	205
Text objects	**BT**, **ET**	5.4	367
Text state	**Tc**, **Tw**, **Tz**, **TL**, **Tf**, **Tr**, **Ts**	5.2	360
Text positioning	**Td**, **TD**, **Tm**, **T***	5.5	368
Text showing	**Tj**, **TJ**, **'**, **"**	5.6	369
Type 3 fonts	**d0**, **d1**	5.10	385
Color	**CS**, **cs**, **SC**, **SCN**, **sc**, **scn**, **G**, **g**, **RG**, **rg**, **K**, **k**	4.21	250
Shading patterns	**sh**	4.24	266
Inline images	**BI**, **ID**, **EI**	4.39	316
XObjects	**Do**	4.34	295
Marked content	**MP**, **DP**, **BMC**, **BDC**, **EMC**	10.7	721
Compatibility	**BX**, **EX**	3.29	126

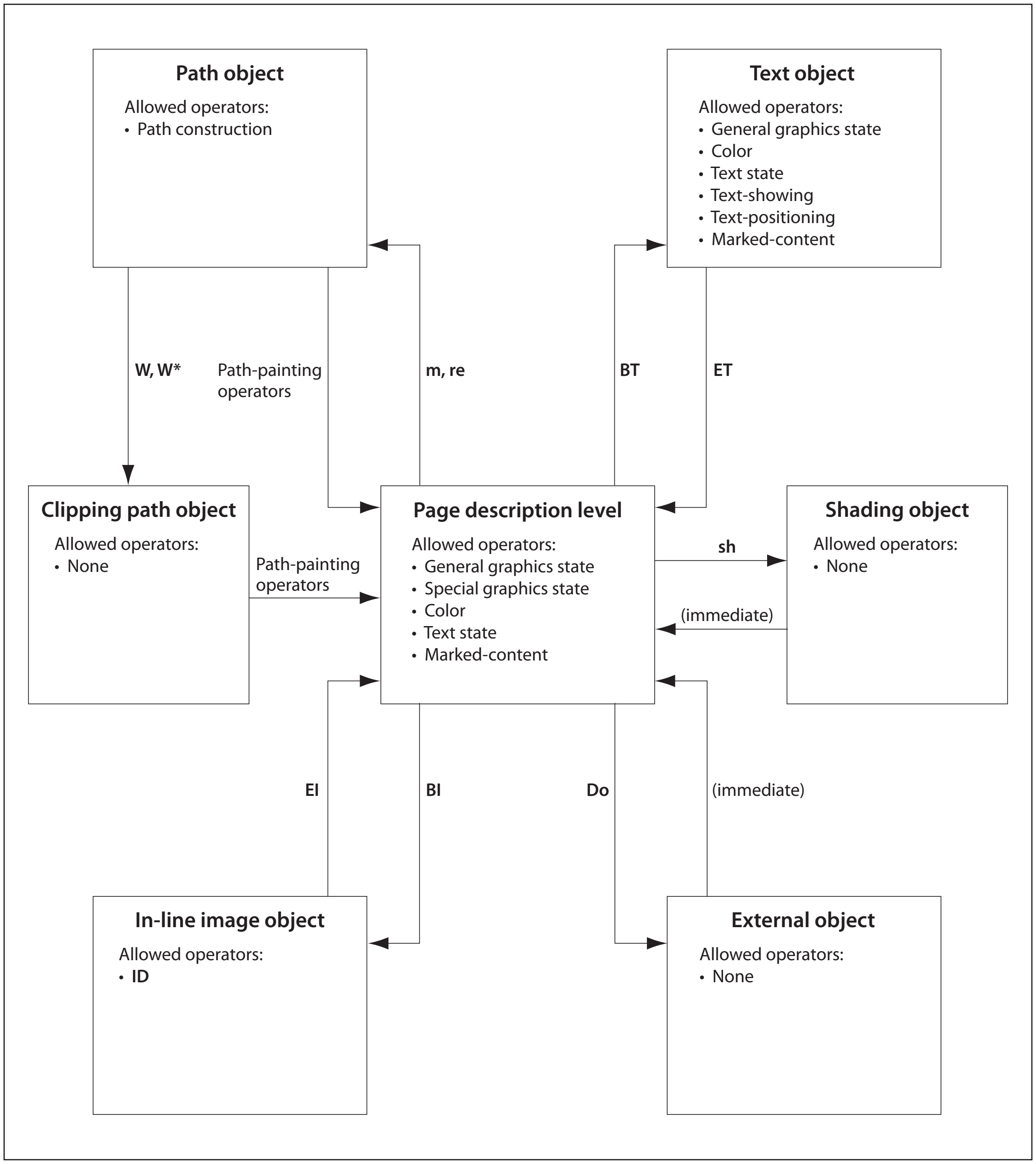


FIGURE 4.1 *Graphics objects*

example, the path construction operators **m** and **re** signal the beginning of a path object. Inside the path object, additional path construction operators are permitted, as are the clipping path operators **W** and **W***, but not general graphics state operators such as **w** or **J**. A path-painting operator, such as **S** or **f**, ends the path object and returns to the page description level.

Note: *A content stream whose operations violate these rules for describing graphics objects can produce unpredictable behavior, even though it may display and print correctly. Applications that attempt to extract graphics objects for editing or other purposes depend on the objects' being well formed. The rules for graphics objects are also important for the proper interpretation of marked content (see Section 10.5, "Marked Content").*

A graphics object also implicitly includes all graphics state parameters that affect its behavior. For instance, a path object depends on the value of the current color parameter at the moment the path object is defined. The effect is as if this parameter were specified as part of the definition of the path object. However, the operators that are invoked at the page description level to set graphics state parameters are *not* considered to belong to any particular graphics object. Graphics state parameters need to be specified only when they change. A graphics object may depend on parameters that were defined much earlier.

Similarly, the individual character strings within a text object implicitly include the graphics state parameters on which they depend. Most of these parameters may be set either inside or outside the text object. The effect is as if they were separately specified for each text string.

The important point is that there is no semantic significance to the exact arrangement of graphics state operators. An application that reads and writes a PDF content stream is not required to preserve this arrangement, but is free to change it to any other arrangement that achieves the same values of the relevant graphics state parameters for each graphics object. An application should not infer any higher-level logical semantics from the arrangement of tokens constituting a graphics object. A separate mechanism, *marked content* (see Section 10.5, "Marked Content"), allows such higher-level information to be explicitly associated with the graphics objects.

4.2 Coordinate Systems

Coordinate systems define the canvas on which all painting occurs. They determine the position, orientation, and size of the text, graphics, and images that appear on a page. This section describes each of the coordinate systems used in PDF, how they are related, and how transformations among them are specified.

4.2.1 Coordinate Spaces

Paths and positions 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 two-dimensional *coordinate space*. A coordinate space is determined by the following properties with respect to the current page:

- The location of the origin
- The orientation of the *x* and *y* axes
- The lengths of the units along each axis

PDF defines several coordinate spaces in which the coordinates specifying graphics objects are interpreted. The following sections describe these spaces and the relationships among them.

Transformations among coordinate spaces are defined by *transformation matrices*, which can specify any linear mapping of two-dimensional coordinates, including translation, scaling, rotation, reflection, and skewing. Transformation matrices are discussed in Sections 4.2.2, “Common Transformations,” and 4.2.3, “Transformation Matrices.”

Device Space

The contents of a page ultimately appear on a raster output device such as a display or a printer. Such 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 called its *device space*. The origin of the device space on different devices can fall in different places on the output page; on displays, the origin can vary depending on the window system. Because the paper or other output medium moves through different printers and imagesetters in different directions, the axes of their device spaces may be oriented differently; for instance, vertical *(y)* coordinates may increase from the top of the page to the bottom on some devices and from bottom to top on others. Finally, different devices have different resolutions; some even have resolutions that differ in the horizontal and vertical directions.

If coordinates in a PDF file were specified in device space, the file would be device-dependent and would appear differently on different devices. For example, images specified in the typical device spaces of a 72-pixel-per-inch display and a 600-dot-per-inch printer would differ in size by more than a factor of 8; an 8-inch line segment on the display would appear less than 1 inch long on the printer. Figure 4.2 shows how the same graphics object, specified in device space, can appear drastically different when rendered on different output devices.

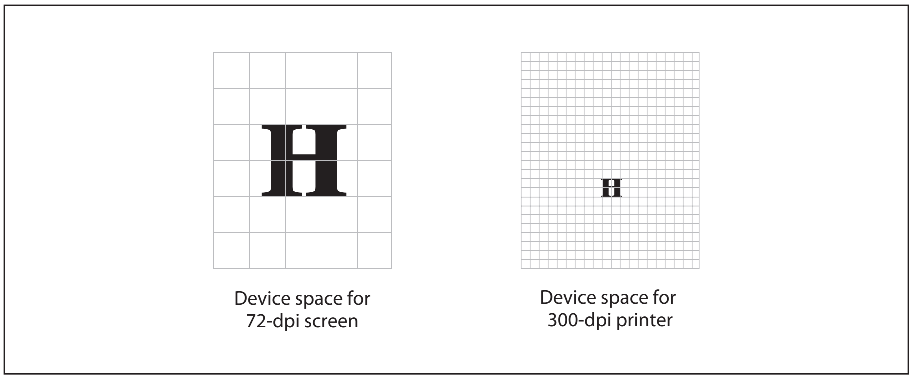

H
H
Device space for 72-dpi screen
Device space for 300-dpi printer

FIGURE 4.2 *Device space*

User Space

To avoid the device-dependent effects of specifying objects in device space, PDF defines a device-independent coordinate system that always bears the same relationship to the current page, regardless of the output device on which printing or displaying will occur. This device-independent coordinate system is called *user space.*

The user space coordinate system is initialized to a default state for each page of a document. The **CropBox** entry in the page dictionary specifies the rectangle of user space corresponding to the visible area of the intended output medium (display window or printed page). The positive *x* axis extends horizontally to the right and the positive *y* axis vertically upward, as in standard mathematical practice (subject to alteration by the **Rotate** entry in the page dictionary). The length of a unit along both the *x* and *y* axes is 1/72 inch. This coordinate system is called *default user space*.

Note: *In PostScript, the origin of default user space always corresponds to the lower-left corner of the output medium. While this convention is common in PDF documents as well, it is not required; the page dictionary's* ***CropBox*** *entry can specify any rectangle of default user space to be made visible on the medium.*

Note: *The unit size in default user space (1/72 inch) is approximately the same as a* point, *a unit widely used in the printing industry. It is not exactly the same, however; there is no universal definition of a point.*

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 region defined by the **CropBox** entry in the page dictionary. The region of default user space that is viewed or printed can be different for each page, and is described in Section 10.10.1, "Page Boundaries."

Note: *Because coordinates in user space (as in any other coordinate space) may be specified as either integers or real numbers, the unit size in default user space does not constrain positions 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 transformation from user space to device space is defined by the *current transformation matrix* (CTM), an element of the PDF graphics state (see Section 4.3, "Graphics State"). A PDF viewer application can adjust the CTM for the native resolution of a particular output device, maintaining the device-independence of the PDF page description itself. Figure 4.3 shows how this allows an object specified in user space to appear the same regardless of the device on which it is rendered.

The default user space provides a consistent, dependable starting place for PDF page descriptions regardless of the output device used. If necessary, a PDF content stream may then modify user space to be more suitable to its needs by applying the *coordinate transformation operator*, **cm** (see Section 4.3.3, "Graphics State Operators"). Thus what may appear to be absolute coordinates in a content stream 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 content stream originally composed to occupy an entire page can be incorporated without change as an element of another page by shrinking the coordinate system in which it is drawn.

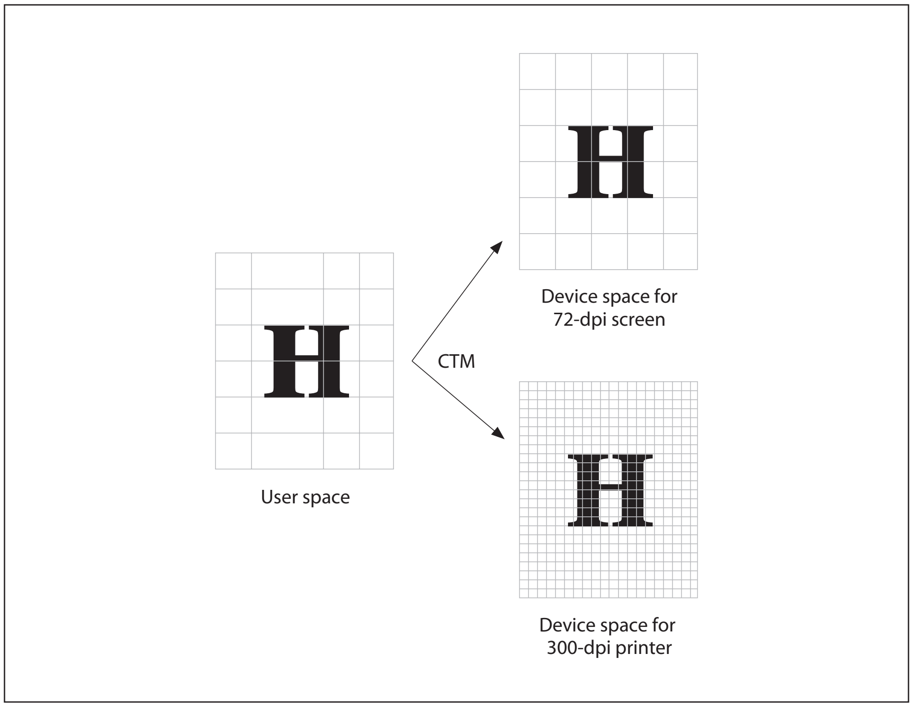

Device space for 72-dpi screen
CTM
User space
Device space for 300-dpi printer

FIGURE 4.3 *User space*

Other Coordinate Spaces

In addition to device space and user space, PDF uses a variety of other coordinate spaces for specialized purposes:

- The coordinates of text are specified in *text space*. The transformation from text space to user space is defined by a *text matrix* in combination with several text-related parameters in the graphics state (see Section 5.3.1, "Text-Positioning Operators").

- Character glyphs in a font are defined in *glyph space* (see Section 5.1.3, "Glyph Positioning and Metrics"). The transformation from glyph space to text space is defined by the *font matrix*. For most types of font, this matrix is predefined to map 1000 units of glyph space to 1 unit of text space; for Type 3 fonts, the font matrix is given explicitly in the font dictionary (see Section 5.5.4, "Type 3 Fonts").
- All sampled images are defined in *image space*. The transformation from image space to user space is predefined and cannot be changed. All images are 1 unit wide by 1 unit high in user space, regardless of the number of samples in the image. To be painted, an image must be mapped to the desired region of the page by temporarily altering the current transformation matrix (CTM).

  ***Note:** In PostScript, unlike PDF, the relationship between image space and user space can be specified explicitly. The fixed transformation prescribed in PDF corresponds to the convention that is recommended for use in PostScript.*
- A form XObject (discussed in Section 4.9, "Form XObjects") is a self-contained content stream that can be treated as a graphical element within another content stream. The space in which it is defined is called *form space*. The transformation from form space to user space is specified by a *form matrix* contained in the form XObject.
- PDF 1.2 defines a type of color known as a *pattern*, discussed in Section 4.6, "Patterns." A pattern is defined either by a content stream that is invoked repeatedly to tile an area or by a shading whose color is a function of position. The space in which a pattern is defined is called *pattern space*. The transformation from pattern space to user space is specified by a *pattern matrix* contained in the pattern.

Relationships among Coordinate Spaces

Figure 4.4 shows the relationships among the coordinate spaces described above. Each arrow in the figure represents a transformation from one coordinate space to another. PDF allows modifications to many of these transformations.

Because PDF coordinate spaces are defined relative to one another, changes made to one transformation can affect the appearance of objects defined in several coordinate spaces. For example, a change in the CTM, which defines the transformation from user space to device space, will affect forms, text, images, and patterns, since they are all "upstream" from user space.

4.2.2 Common Transformations

A *transformation matrix* specifies the relationship between two coordinate spaces. By modifying a transformation matrix, objects can be scaled, rotated, translated, or transformed in other ways.

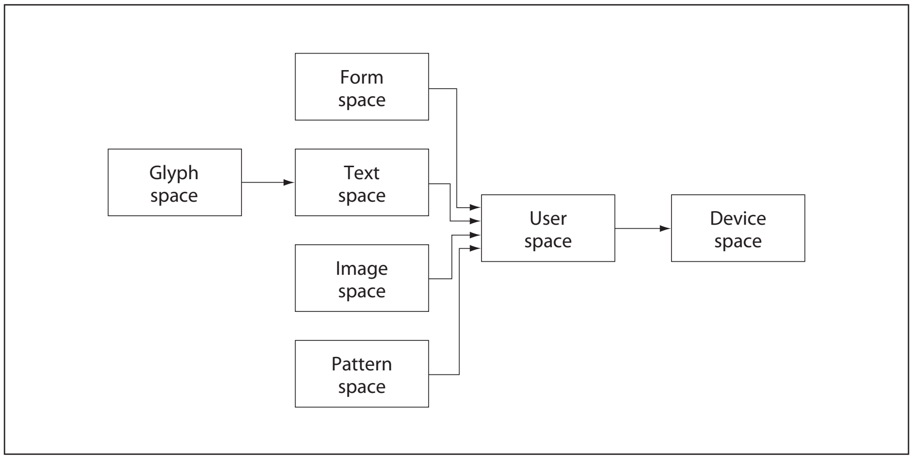


FIGURE 4.4 *Relationships among coordinate systems*

A transformation matrix in PDF is specified by six numbers, usually in the form of an array containing six elements. In its most general form, this array is denoted [*a b c d e f*]; it can represent any linear transformation from one coordinate system to another. This section lists the arrays that specify the most common transformations; Section 4.2.3, "Transformation Matrices," discusses more mathematical details of transformations, including information on specifying transformations that are combinations of those listed here.

- Translations are specified as [1 0 0 1 t_x t_y], where t_x and t_y are the distances to translate the origin of the coordinate system in the horizontal and vertical dimensions, respectively.
- Scaling is obtained by [s_x 0 0 s_y 0 0]. This scales the coordinates so that 1 unit in the horizontal and vertical dimensions of the new coordinate system is the same size as s_x and s_y units, respectively, in the previous coordinate system.
- Rotations are produced by [cos θ sin θ −sin θ cos θ 0 0], which has the effect of rotating the coordinate system axes by an angle θ counterclockwise.
- Skew is specified by [1 tan α tan β 1 0 0], which skews the *x* axis by an angle α and the *y* axis by an angle β.

Figure 4.5 shows examples of each transformation. The directions of translation, rotation, and skew shown in the figure correspond to positive values of the array elements.

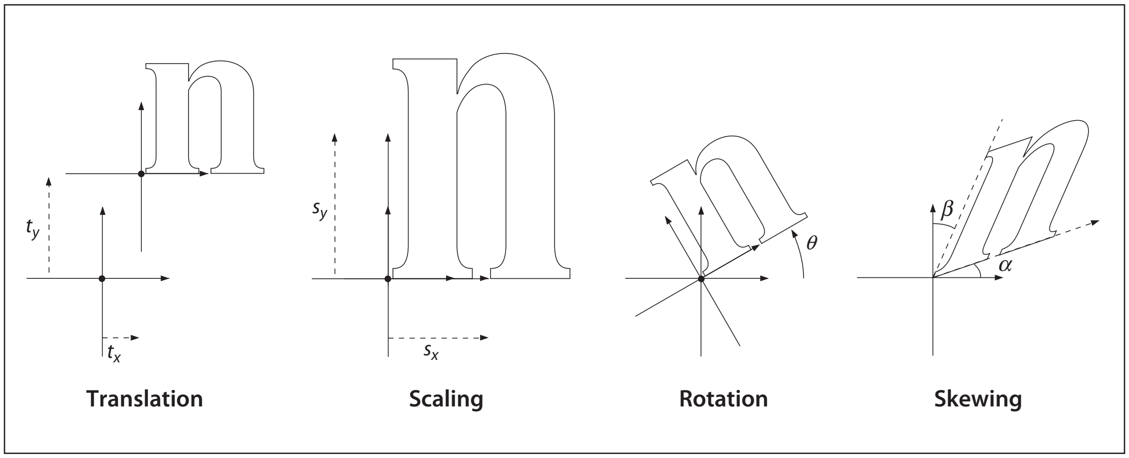


FIGURE 4.5 *Effects of coordinate transformations*

If several transformations are combined, the order in which they are applied is significant. For example, first scaling and then translating the *x* axis is not the same as first translating and then scaling it. In general, to obtain the expected results, transformations should be done in the following order:

1. Translate
2. Rotate
3. Scale or skew

Figure 4.6 shows the effect of the order in which transformations are applied. The figure shows two sequences of transformations applied to a coordinate system. After each successive transformation, an outline of the letter n is drawn.

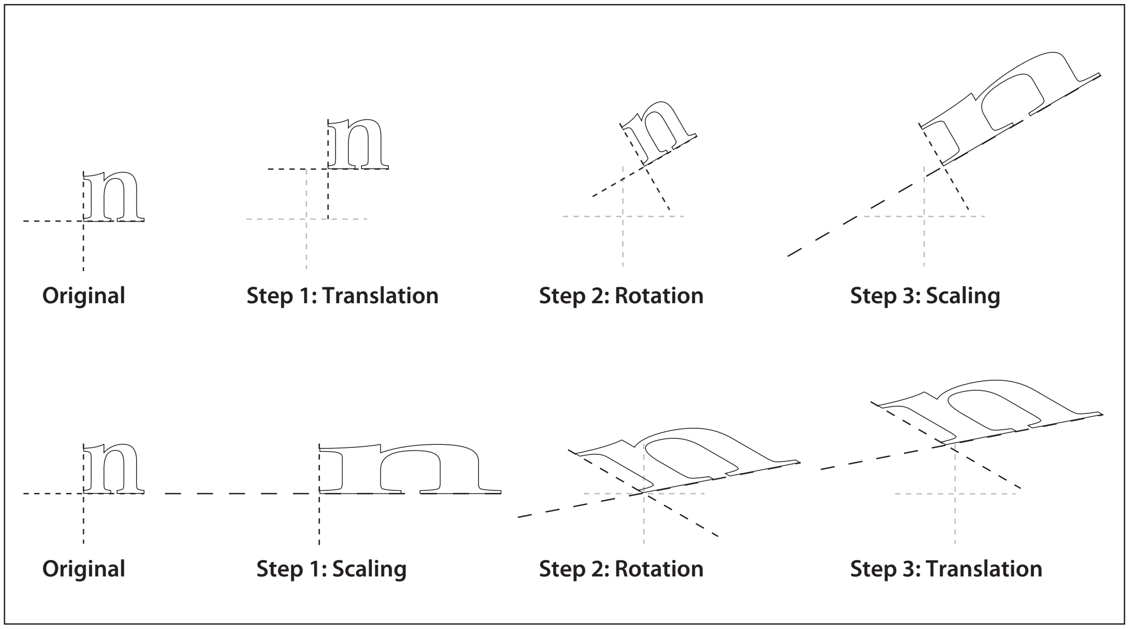


FIGURE 4.6 *Effect of transformation order*

The transformations shown in the figure are as follows:

- A translation of 10 units in the *x* direction and 20 units in the *y* direction
- A rotation of 30 degrees
- A scaling by a factor of 3 in the *x* direction

In the figure, the axes are shown with a dash pattern having a 2-unit dash and a 2-unit gap. In addition, the original (untransformed) axes are shown in a lighter color for reference. Notice that the scale-rotate-translate ordering results in a distortion of the coordinate system, leaving the x and y axes no longer perpendicular, while the recommended translate-rotate-scale ordering does not.

4.2.3 Transformation Matrices

This section discusses the mathematics of transformation matrices. It is not necessary to read this section in order to use the transformations described previously; the information is presented for the benefit of readers who want to gain a deeper understanding of the theoretical basis of coordinate transformations.

To understand the mathematics of coordinate transformations in PDF, it is vital to remember two points:

- *Transformations alter coordinate systems, not graphics objects*. All objects painted before a transformation is applied are unaffected by the transformation. Objects painted after the transformation is applied will be interpreted in the transformed coordinate system.
- *Transformation matrices specify the transformation from the new (transformed) coordinate system to the original (untransformed) coordinate system*. All coordinates used after the transformation are expressed in the transformed coordinate system. PDF applies the transformation matrix to find the equivalent coordinates in the untransformed coordinate system.

Note: *Many computer graphics textbooks consider transformations of graphics objects rather than of coordinate systems. Although either approach is correct and self-consistent, some details of the calculations differ depending on which point of view is taken.*

PDF represents coordinates in a two-dimensional space. The point (x, y) in such a space can be expressed in vector form as $[x \ y \ 1]$. The constant third element of this vector (1) is needed so that the vector can be used with 3-by-3 matrices in the calculations described below.

The transformation between two coordinate systems is represented by a 3-by-3 transformation matrix written as

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Because a transformation matrix has only six elements that can be changed, it is usually specified in PDF as the six-element array [*a b c d e f*].

Coordinate transformations are expressed as matrix multiplications:

$$[x' \; y' \; 1] = [x \; y \; 1] \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Because PDF transformation matrices specify the conversion from the transformed coordinate system to the original (untransformed) coordinate system, x' and y' in this equation are the coordinates in the untransformed coordinate system, while x and y are the coordinates in the transformed system. Carrying out the multiplication, we have

$$\begin{aligned} x' &= a \times x + c \times y + e \\ y' &= b \times x + d \times y + f \end{aligned}$$

If a series of transformations is carried out, the matrices representing each of the individual transformations can be multiplied together to produce a single equivalent matrix representing the composite transformation.

Matrix multiplication is not commutative—the order in which matrices are multiplied is significant. Consider a sequence of two transformations: a scaling transformation applied to the user space coordinate system, followed by a conversion from the resulting scaled user space to device space. Let M_S be the matrix specifying the scaling and M_C the current transformation matrix, which transforms user space to device space. Recalling that coordinates are always specified in the transformed space, the correct order of transformations must first convert the scaled coordinates to default user space and then the default user space coordinates to device space. This can be expressed as

$$X_D = X_U \times M_C = (X_S \times M_S) \times M_C = X_S \times (M_S \times M_C)$$

where

X_D denotes the coordinates in device space

X_U denotes the coordinates in default user space

X_S denotes the coordinates in scaled user space

This shows that when a new transformation is concatenated with an existing one, the matrix representing it must be multiplied *before* (*premultiplied* with) the existing transformation matrix.

This result is true in general for PDF: when a sequence of transformations is carried out, the matrix representing the combined transformation *(M')* is calculated by premultiplying the matrix representing the additional transformation *(M_T)* with the one representing all previously existing transformations *(M)*:

$$M' = M_T \times M$$

Note: *When rendering graphics objects, it is sometimes necessary for a viewer application to perform the inverse of a transformation—that is, to find the user space coordinates that correspond to a given pair of device space coordinates. Not all transformations are invertible, however. For example, if a matrix contains* a, b, c, *and* d *elements that are all zero, 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 unintended operations, such as scaling by 0. Use of a noninvertible matrix when painting graphics objects can result in unpredictable behavior.*

4.3 Graphics State

A PDF viewer application 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 **f** (fill) operator implicitly uses the *current color* parameter, and the **S** (stroke) operator additionally uses the *current line width* parameter from the graphics state.

The graphics state is initialized at the beginning of each page, using the default values specified in Tables 4.2 and 4.3. Table 4.2 lists those graphics state parameters that are device-independent and are appropriate to specify in page descriptions. The parameters listed in Table 4.3 control details of the rendering (scan conversion) process and are device-dependent; a page description that is intended to be device-independent should not modify these parameters.

TABLE 4.2 Device-independent graphics state parameters

PARAMETER	TYPE	VALUE
CTM	array	The *current transformation matrix*, which maps positions from user coordinates to device coordinates (see Section 4.2, "Coordinate Systems"). This matrix is modified by each application of the coordinate transformation operator, **cm**. Initial value: a matrix that transforms default user coordinates to device coordinates.
clipping path	(internal)	The *current clipping path*, which defines the boundary against which all output is to be cropped (see Section 4.4.3, "Clipping Path Operators"). Initial value: the boundary of the entire imageable portion of the output page.
color space	name or array	The *current color space* in which color values are to be interpreted (see Section 4.5, "Color Spaces"). There are two separate color space parameters: one for stroking and one for all other painting operations. Initial value: **DeviceGray**.
color	(various)	The *current color* to be used during painting operations (see Section 4.5, "Color Spaces"). The type and interpretation of this parameter depend on the current color space; for most color spaces, a color value consists of one to four numbers. There are two separate color parameters: one for stroking and one for all other painting operations. Initial value: black.
text state	(various)	A set of nine graphics state parameters that pertain only to the painting of text. These include parameters that select the font, scale the glyphs to an appropriate size, and accomplish other effects. The text state parameters are described in Section 5.2, "Text State Parameters and Operators."

PARAMETER	TYPE	VALUE
line width	number	The thickness, in user space units, of paths to be stroked (see “Line Width” on page 185). Initial value: 1.0.
line cap	integer	A code specifying the shape of the endpoints for any open path that is stroked (see “Line Cap Style” on page 186). Initial value: 0, for square butt caps.
line join	integer	A code specifying the shape of joints between connected segments of a stroked path (see “Line Join Style” on page 186). Initial value: 0, for mitered joins.
miter limit	number	The maximum length of mitered line joins for stroked paths (see “Miter Limit” on page 187). This parameter limits the length of “spikes” produced when line segments join at sharp angles. Initial value: 10.0, for a miter cutoff below approximately 11.5 degrees.
dash pattern	array and number	A description of the dash pattern to be used when paths are stroked (see “Line Dash Pattern” on page 187). Initial value: a solid line.
rendering intent	name	The *rendering intent* to be used when converting CIE-based colors to device colors (see “Rendering Intents” on page 230). Default value: **RelativeColorimetric**.
stroke adjustment	boolean	*(PDF 1.2)* A flag specifying whether to compensate for possible rasterization effects when stroking a path with a line width that is small relative to the pixel resolution of the output device (see Section 6.5.4, “Automatic Stroke Adjustment”). Note that this is considered a device-independent parameter, even though the details of its effects are device-dependent. Initial value: **false**.
blend mode	name or array	*(PDF 1.4)* The *current blend mode* to be used in the transparent imaging model (see Sections 7.2.4, “Blend Mode,” and 7.5.2, “Specifying Blending Color Space and Blend Mode”). This parameter is implicitly reset to its initial value at the beginning of execution of a transparency group XObject (see Section 7.5.5, “Transparency Group XObjects”). Initial value: **Normal**.

PARAMETER	TYPE	VALUE
soft mask	dictionary or name	*(PDF 1.4)* A *soft-mask dictionary* (see “Soft-Mask Dictionaries” on page 510) specifying the mask shape or mask opacity values to be used in the transparent imaging model (see “Source Shape and Opacity” on page 485 and “Mask Shape and Opacity” on page 508), or the name **None** if no such mask is specified. This parameter is implicitly reset to its initial value at the beginning of execution of a transparency group XObject (see Section 7.5.5, “Transparency Group XObjects”). Initial value: **None**.
alpha constant	number	*(PDF 1.4)* The constant shape or constant opacity value to be used in the transparent imaging model (see “Source Shape and Opacity” on page 485 and “Constant Shape and Opacity” on page 509). There are two separate alpha constant parameters: one for stroking and one for all other painting operations. This parameter is implicitly reset to its initial value at the beginning of execution of a transparency group XObject (see Section 7.5.5, “Transparency Group XObjects”). Initial value: 1.0.
alpha source	boolean	*(PDF 1.4)* A flag specifying whether the current soft mask and alpha constant parameters are to be interpreted as shape values (**true**) or opacity values (**false**). This flag also governs the interpretation of the **SMask** entry, if any, in an image dictionary (see Section 4.8.4, “Image Dictionaries”). Initial value: **false**.

TABLE 4.3 Device-dependent graphics state parameters

PARAMETER	TYPE	VALUE
overprint	boolean	*(PDF 1.2)* A flag specifying (on output devices that support the overprint control feature) whether painting in one set of colorants should cause the corresponding areas of other colorants to be erased (**false**) or left unchanged (**true**); see Section 4.5.6, “Overprint Control.” In PDF 1.3, there are two separate overprint parameters: one for stroking and one for all other painting operations. Initial value: **false**.
overprint mode	number	*(PDF 1.3)* A code specifying whether a color component value of 0 in a **DeviceCMYK** color space should erase that component (0) or leave it unchanged (1) when overprinting (see Section 4.5.6, “Overprint Control”). Initial value: 0.

PARAMETER	TYPE	VALUE
black generation	function or name	*(PDF 1.2)* A function that calculates the level of the black color component to use when converting *RGB* colors to *CMYK* (see Section 6.2.3, "Conversion from DeviceRGB to DeviceCMYK"). Initial value: installation-dependent.
undercolor removal	function or name	*(PDF 1.2)* A function that calculates the reduction in the levels of the cyan, magenta, and yellow color components to compensate for the amount of black added by black generation (see Section 6.2.3, "Conversion from DeviceRGB to DeviceCMYK"). Initial value: installation-dependent.
transfer	function, array, or name	*(PDF 1.2)* A function that adjusts device gray or color component levels to compensate for nonlinear response in a particular output device (see Section 6.3, "Transfer Functions"). Initial value: installation-dependent.
halftone	dictionary, stream, or name	*(PDF 1.2)* A halftone screen for gray and color rendering, specified as a halftone dictionary or stream (see Section 6.4, "Halftones"). Initial value: installation-dependent.
flatness	number	The precision with which curves are to be rendered on the output device (see Section 6.5.1, "Flatness Tolerance"). The value of this parameter 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	*(PDF 1.3)* The precision with which color gradients are to be rendered on the output device (see Section 6.5.2, "Smoothness Tolerance"). The value of this parameter gives the maximum error tolerance, 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.

Some graphics state parameters are set with specific PDF operators, some are set by including a particular entry in a *graphics state parameter dictionary*, and some can be specified either way. The current line width, for example, can be set either with the **w** operator or (in PDF 1.3) with the **LW** entry in a graphics state parameter dictionary, whereas the current color is set only with specific operators and the current halftone is set only with a graphics state parameter dictionary. It is expected that all future graphics state parameters will be specified with new entries in the graphics state parameter dictionary rather than with new operators.

In general, the operators that set graphics state parameters simply store them unchanged for later use by the painting operators. However, some parameters have special properties or behavior:

- Most parameters must be of the correct type or have values that fall within a certain range.
- Parameters that are numeric values, such as the current color, line width, and miter limit, are forced into valid range, if necessary. However, they are *not* adjusted to reflect capabilities of the raster output device, such as resolution or number of distinguishable colors. Painting operators perform such adjustments, but the adjusted values are not stored back into the graphics state.
- Paths are internal objects that are not directly represented in PDF.

Note: *As indicated in Tables 4.2 and 4.3, some of the parameters—color space, color, and overprint—have two values, one used for stroking (of paths and text objects) and one for all other painting operations. The two parameter values can be set independently, allowing for operations such as combined filling and stroking of the same path with different colors. Except where noted, a term such as* current color *should be interpreted to refer to whichever color parameter applies to the operation being performed. When necessary, the individual color parameters are distinguished explicitly as the* stroking color *and the* nonstroking color.

4.3.1 Graphics State Stack

A well-structured PDF document typically contains many graphical elements that are essentially independent of each other and sometimes nested to multiple levels. The *graphics state stack* allows these elements to make local changes to the graphics state without disturbing the graphics state of the surrounding environment. The stack is a LIFO (last in, first out) data structure in which the contents of the graphics state can be saved and later restored using the following operators:

- The **q** operator pushes a copy of the entire graphics state onto the stack.
- The **Q** operator restores the entire graphics state to its former value by popping it from the stack.

These operators can be used to encapsulate a graphical element so that it can modify parameters of the graphics state and later restore them to their previous values. Occurrences of the **q** and **Q** operators must be balanced within a given content stream (or within the sequence of streams specified in a page dictionary's **Contents** array).

4.3.2 Details of Graphics State Parameters

This section gives details of several of the device-independent graphics state parameters listed in Table 4.2 on page 180.

Line Width

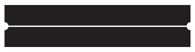
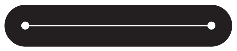
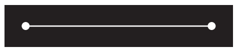
The *line width* parameter specifies the thickness of the line used to stroke a path. It is a nonnegative number expressed in user space units; stroking a path entails painting all points whose perpendicular distance from the path in user space is less than or equal to half the line width. The effect 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 horizontal and vertical dimensions, the thickness of stroked lines in device space will vary according to their orientation. The actual line width achieved 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 can be used to ensure uniform line width; see Section 6.5.4, "Automatic Stroke Adjustment."

A line width of 0 denotes 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.

Line Cap Style

The *line cap style* specifies the shape to be used at the ends of open subpaths (and dashes, if any) when they are stroked. Table 4.4 shows the possible values.

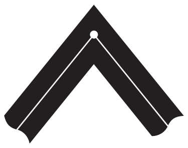
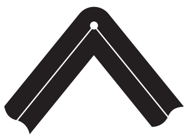
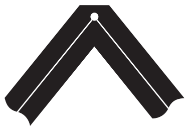
TABLE 4.4 Line cap styles

STYLE	APPEARANCE	DESCRIPTION
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.

Line Join Style

The *line join style* specifies the shape to be used at the corners of paths that are stroked. Table 4.5 shows the possible values. 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.

TABLE 4.5 Line join styles

STYLE	APPEARANCE	DESCRIPTION
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 "Miter Limit," above), a bevel join is used instead.
1		*Round join*. An arc of a circle with a diameter equal to the line width is drawn around the point where the two segments meet, connecting the outer edges of the strokes for the two segments. This pieslice-shaped figure is filled in, producing a rounded corner.
2		*Bevel join*. The two segments are finished with butt caps (see "Line Cap Style" on page 186) and the resulting notch beyond the ends of the segments is filled with a triangle.

***Note:** The definition of "round join" was changed in PDF 1.5. In rare cases, the implementation of the previous specification could produce unexpected results.*

Miter Limit

When two line segments meet at a sharp angle and mitered joins have been specified as the line join style, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The *miter limit* imposes a maximum on the ratio of the miter length to the line width (see Figure 4.7). When the limit is exceeded, the join is converted from a miter to a bevel.

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

$$\frac{miterLength}{lineWidth} = \frac{1}{\sin\left(\frac{\varphi}{2}\right)}$$

For example, a miter limit of 1.414 converts miters to bevels for φ less than 90 degrees, a limit of 2.0 converts them for φ less than 60 degrees, and a limit of 10.0 converts them for φ less than approximately 11.5 degrees.

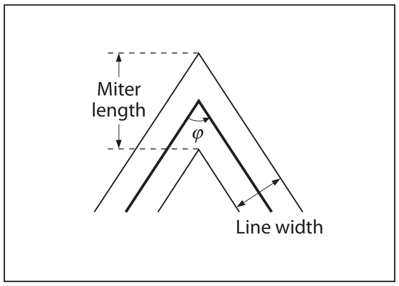


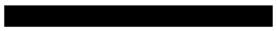
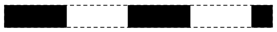
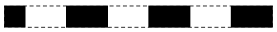
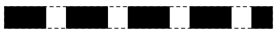
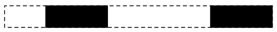
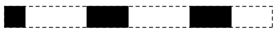
FIGURE 4.7 *Miter length*

Line Dash Pattern

The *line dash pattern* controls the pattern of dashes and gaps used to stroke paths. It is specified by a *dash array* and a *dash phase*. The dash array's elements are numbers that specify the lengths of alternating dashes and gaps; the dash phase specifies the distance into the dash pattern at which to start the dash. The elements of both the dash array and the dash phase are expressed in user space units.

Before beginning to stroke a path, the dash array is cycled through, adding up the lengths of dashes and gaps. When the accumulated length equals the value specified by the dash phase, stroking of the path begins, using the dash array cyclically from that point onward. Table 4.6 shows examples of line dash patterns. As can be seen from the table, an empty dash array and zero phase can be used to restore the dash pattern to a solid line.

TABLE 4.6 Examples of line dash patterns

DASH ARRAY AND PHASE	APPEARANCE	DESCRIPTION
[] 0		No dash; solid, unbroken lines
[3] 0		3 units on, 3 units off, …
[2] 1		1 on, 2 off, 2 on, 2 off, …
[2 1] 0		2 on, 1 off, 2 on, 1 off, …
[3 5] 6		2 off, 3 on, 5 off, 3 on, 5 off, …
[2 3] 11		1 on, 3 off, 2 on, 3 off, 2 on, …

Dashed lines wrap around curves and corners just as solid stroked lines do. The ends of each dash are treated with the current line cap style, and corners within dashes are treated with the current line join style. A stroking operation takes no measures to coordinate the dash pattern with features of the path; it simply dispenses dashes and gaps along the path in the pattern defined by the dash array.

When a path consisting of several subpaths is stroked, each subpath is treated independently—that is, the dash pattern is restarted and the dash phase is reapplied to it at the beginning of each subpath.

4.3.3 Graphics State Operators

Table 4.7 shows the operators that set the values of parameters in the graphics state. (See also the color operators listed in Table 4.21 on page 250 and the text state operators in Table 5.2 on page 360.)

TABLE 4.7 Graphics state operators

OPERANDS	OPERATOR	DESCRIPTION
—	**q**	Save the current graphics state on the graphics state stack (see "Graphics State Stack" on page 184).
—	**Q**	Restore the graphics state by removing the most recently saved state from the stack and making it the current state (see "Graphics State Stack" on page 184).
a b c d e f	**cm**	Modify the current transformation matrix (CTM) by concatenating the specified matrix (see Section 4.2.1, "Coordinate Spaces"). Although the operands specify a matrix, they are written as six separate numbers, not as an array.
lineWidth	**w**	Set the line width in the graphics state (see "Line Width" on page 185).
lineCap	**J**	Set the line cap style in the graphics state (see "Line Cap Style" on page 186).
lineJoin	**j**	Set the line join style in the graphics state (see "Line Join Style" on page 186).
miterLimit	**M**	Set the miter limit in the graphics state (see "Miter Limit" on page 187).
dashArray dashPhase	**d**	Set the line dash pattern in the graphics state (see "Line Dash Pattern" on page 187).
intent	**ri**	*(PDF 1.1)* Set the color rendering intent in the graphics state (see "Rendering Intents" on page 230).
flatness	**i**	Set the flatness tolerance in the graphics state (see Section 6.5.1, "Flatness Tolerance"). *flatness* is a number in the range 0 to 100; a value of 0 specifies the output device's default flatness tolerance.
dictName	**gs**	*(PDF 1.2)* Set the specified parameters in the graphics state. *dictName* is the name of a graphics state parameter dictionary in the **ExtGState** subdictionary of the current resource dictionary (see the next section).

4.3.4 Graphics State Parameter Dictionaries

While some parameters in the graphics state can be set with individual operators, as shown in Table 4.7, others cannot. The latter can only be set with the generic graphics state operator **gs** *(PDF 1.2)*. The operand supplied to this operator is the name of a *graphics state parameter dictionary* whose contents specify the values of one or more graphics state parameters. This name is looked up in the **ExtGState** subdictionary of the current resource dictionary. (The name **ExtGState**, for "extended graphics state," is a vestige of earlier versions of PDF.)

Note: *The graphics state parameter dictionary is also used by type 2 patterns, which do not have a content stream in which the graphics state operators could be invoked (see Section 4.6.3, "Shading Patterns").*

Each entry in the parameter dictionary specifies the value of an individual graphics state parameter, as shown in Table 4.8. It is not necessary for all entries to be present for every invocation of the **gs** operator; the parameter dictionary supplied may include any desired combination of parameter entries. The results of **gs** are cumulative; parameter values established in previous invocations will persist until explicitly overridden. Note that some parameters appear in both Tables 4.7 and 4.8; these parameters can be set either with individual graphics state operators or with **gs**. It is expected that any future extensions to the graphics state will be implemented by adding new entries to the graphics state parameter dictionary, rather than by introducing new graphics state operators.

TABLE 4.8 Entries in a graphics state parameter dictionary

KEY	TYPE	DESCRIPTION
Type	name	*(Optional)* The type of PDF object that this dictionary describes; must be **ExtGState** for a graphics state parameter dictionary.
LW	number	*(Optional; PDF 1.3)* The line width (see "Line Width" on page 185).
LC	integer	*(Optional; PDF 1.3)* The line cap style (see "Line Cap Style" on page 186).
LJ	integer	*(Optional; PDF 1.3)* The line join style (see "Line Join Style" on page 186).
ML	number	*(Optional; PDF 1.3)* The miter limit (see "Miter Limit" on page 187).
D	array	*(Optional; PDF 1.3)* The line dash pattern, expressed as an array of the form [*dashArray dashPhase*], where *dashArray* is itself an array and *dashPhase* is an integer (see "Line Dash Pattern" on page 187).
RI	name	*(Optional; PDF 1.3)* The name of the rendering intent (see "Rendering Intents" on page 230).

KEY	TYPE	DESCRIPTION
OP	boolean	*(Optional)* A flag specifying whether to apply overprint (see Section 4.5.6, "Overprint Control"). In PDF 1.2 and earlier, there is a single overprint parameter that applies to all painting operations. Beginning with PDF 1.3, there are two separate overprint parameters: one for stroking and one for all other painting operations. Specifying an **OP** entry sets both parameters unless there is also an **op** entry in the same graphics state parameter dictionary, in which case the **OP** entry sets only the overprint parameter for stroking.
op	boolean	*(Optional; PDF 1.3)* A flag specifying whether to apply overprint (see Section 4.5.6, "Overprint Control") for painting operations other than stroking. If this entry is absent, the **OP** entry, if any, sets this parameter.
OPM	integer	*(Optional; PDF 1.3)* The overprint mode (see Section 4.5.6, "Overprint Control").
Font	array	*(Optional; PDF 1.3)* An array of the form [*font size*], where *font* is an indirect reference to a font dictionary and *size* is a number expressed in text space units. These two objects correspond to the operands of the **Tf** operator (see Section 5.2, "Text State Parameters and Operators"); however, the first operand is an indirect object reference instead of a resource name.
BG	function	*(Optional)* The black-generation function, which maps the interval [0.0 1.0] to the interval [0.0 1.0] (see Section 6.2.3, "Conversion from DeviceRGB to DeviceCMYK").
BG2	function or name	*(Optional; PDF 1.3)* Same as **BG** except that the value may also be the name Default, denoting the black-generation function that was in effect at the start of the page. If both **BG** and **BG2** are present in the same graphics state parameter dictionary, **BG2** takes precedence.
UCR	function	*(Optional)* The undercolor-removal function, which maps the interval [0.0 1.0] to the interval [−1.0 1.0] (see Section 6.2.3, "Conversion from DeviceRGB to DeviceCMYK").
UCR2	function or name	*(Optional; PDF 1.3)* Same as **UCR** except that the value may also be the name Default, denoting the undercolor-removal function that was in effect at the start of the page. If both **UCR** and **UCR2** are present in the same graphics state parameter dictionary, **UCR2** takes precedence.

KEY	TYPE	DESCRIPTION
TR	function, array, or name	*(Optional)* The transfer function, which maps the interval [0.0 1.0] to the interval [0.0 1.0] (see Section 6.3, "Transfer Functions"). The value is either a single function (which applies to all process colorants) or an array of four functions (which apply to the process colorants individually). The name **Identity** may be used to represent the identity function.
TR2	function, array, or name	*(Optional; PDF 1.3)* Same as **TR** except that the value may also be the name Default, denoting the transfer function that was in effect at the start of the page. If both **TR** and **TR2** are present in the same graphics state parameter dictionary, **TR2** takes precedence.
HT	dictionary, stream, or name	*(Optional)* The halftone dictionary or stream (see Section 6.4, "Halftones") or the name Default, denoting the halftone that was in effect at the start of the page.
FL	number	*(Optional; PDF 1.3)* The flatness tolerance (see Section 6.5.1, "Flatness Tolerance").
SM	number	*(Optional; PDF 1.3)* The smoothness tolerance (see Section 6.5.2, "Smoothness Tolerance").
SA	boolean	*(Optional)* A flag specifying whether to apply automatic stroke adjustment (see Section 6.5.4, "Automatic Stroke Adjustment").
BM	name or array	*(Optional; PDF 1.4)* The current blend mode to be used in the transparent imaging model (see Sections 7.2.4, "Blend Mode," and 7.5.2, "Specifying Blending Color Space and Blend Mode").
SMask	dictionary or name	*(Optional; PDF 1.4)* The current soft mask, specifying the mask shape or mask opacity values to be used in the transparent imaging model (see "Source Shape and Opacity" on page 485 and "Mask Shape and Opacity" on page 508). ***Note:** Although the current soft mask is sometimes referred to as a "soft clip," altering it with the **gs** operator completely replaces the old value with the new one, rather than intersecting the two as is done with the current clipping path parameter (see Section 4.4.3, "Clipping Path Operators").*
CA	number	*(Optional; PDF 1.4)* The current stroking alpha constant, specifying the constant shape or constant opacity value to be used for stroking operations in the transparent imaging model (see "Source Shape and Opacity" on page 485 and "Constant Shape and Opacity" on page 509).
ca	number	*(Optional; PDF 1.4)* Same as **CA**, but for nonstroking operations.

KEY	TYPE	DESCRIPTION
AIS	boolean	*(Optional; PDF 1.4)* The alpha source flag ("alpha is shape"), specifying whether the current soft mask and alpha constant are to be interpreted as shape values (**true**) or opacity values (**false**).
TK	boolean	*(Optional; PDF 1.4)* The text knockout flag, which determines the behavior of overlapping glyphs within a text object in the transparent imaging model (see Section 5.2.7, "Text Knockout").

Example 4.1 shows two graphics state parameter dictionaries. In the first, automatic stroke adjustment is turned on, and the dictionary includes a transfer function that inverts its value, $f(x) = 1 - x$. In the second, overprint is turned off, and the dictionary includes a parabolic transfer function, $f(x) = (2x - 1)^2$, with a sample of 21 values. The domain of the transfer function, [0.0 1.0], is mapped to [0 20], and the range of the sample values, [0 255], is mapped to the range of the transfer function, [0.0 1.0].

Example 4.1

```
10 0 obj                                    % Page object
   << /Type /Page
      /Parent 5 0 R
      /Resources 20 0 R
      /Contents 40 0 R
   >>
endobj

20 0 obj                                    % Resource dictionary for page
   << /ProcSet [/PDF /Text]
      /Font << /F1 25 0 R >>
      /ExtGState << /GS1 30 0 R
                    /GS2 35 0 R
                 >>
   >>
endobj

30 0 obj                                    % First graphics state parameter dictionary
   << /Type /ExtGState
      /SA true
      /TR 31 0 R
   >>
endobj
```

```
31  0  obj                                  % First transfer function
    <<  /FunctionType  0
        /Domain  [0.0  1.0]
        /Range  [0.0  1.0]
        /Size  2
        /BitsPerSample  8
        /Length  7
        /Filter  /ASCIIHexDecode
    >>
stream
01  00  >
endstream
endobj

35  0  obj                                  % Second graphics state parameter dictionary
    <<  /Type  /ExtGState
        /OP  false
        /TR  36 0 R
    >>
endobj

36  0  obj                                  % Second transfer function
    <<  /FunctionType  0
        /Domain  [0.0  1.0]
        /Range  [0.0  1.0]
        /Size  21
        /BitsPerSample  8
        /Length  63
        /Filter  /ASCIIHexDecode
    >>
stream
FF  CE  A3  7C  5B  3F  28  16  0A  02  00  02  0A  16  28  3F  5B  7C  A3  CE  FF  >
endstream
endobj
```

4.4 Path Construction and Painting

Paths define shapes, trajectories, and regions of all sorts. They are used to draw lines, define the shapes of filled areas, and specify boundaries for clipping other graphics. The graphics state includes a *current clipping path* that defines the clipping boundary for the current page. At the beginning of each page, the clipping path is initialized to include the entire page.

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 significant. 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, **h**, 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*.

As discussed in Section 4.1, "Graphics Objects," a path object is defined by a sequence of operators to construct the path, followed by one or more operators to paint the path or to use it as a clipping boundary. PDF path operators fall into three categories:

- *Path construction operators* (Section 4.4.1) define the geometry of a path. A path is constructed by sequentially applying one or more of these operators.
- *Path-painting operators* (Section 4.4.2) end a path object, usually causing the object to be painted on the current page in any of a variety of ways.
- *Clipping path operators* (Section 4.4.3), invoked immediately prior to a path-painting operator, cause the path object also to be used for clipping of subsequent graphics objects.

4.4.1 Path Construction Operators

A page description begins with an empty path and builds up its definition by invoking one or more path construction operators to add segments to it. The path construction operators may be invoked in any sequence, but the first one invoked must be **m** or **re** to begin a new subpath. The path definition concludes with the application of a path-painting operator such as **S**, **f**, or **b** (see Section 4.4.2, "Path-Painting Operators"); this may optionally be preceded by one of the clipping path operators **W** or **W*** (Section 4.4.3, "Clipping Path Operators"). Note that the path construction operators in themselves do not place any marks on the page; only the painting operators do that. A path definition is not complete until a path-painting operator has been applied to it.

The path currently under construction is called the *current path*. In PDF (unlike PostScript), the current path is *not* part of the graphics state and is *not* saved and restored along with the other graphics state parameters. PDF paths are strictly internal objects with no explicit representation. Once a path has been painted, it is no longer defined; there is then no current path until a new one is begun with the **m** or **re** operator.

The trailing endpoint of the segment most recently added to the current path 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 an error.

Table 4.9 shows the path construction operators. All operands are numbers denoting coordinates in user space.

TABLE 4.9 Path construction operators

OPERANDS	OPERATOR	DESCRIPTION
x y	**m**	Begin a new subpath by moving the current point to coordinates (x, y), omitting any connecting line segment. If the previous path construction operator in the current path was also **m**, the new **m** overrides it; no vestige of the previous **m** operation remains in the path.
x y	**l** (lowercase **L**)	Append a straight line segment from the current point to the point (x, y). The new current point is (x, y).
x_1 y_1 x_2 y_2 x_3 y_3	**c**	Append a cubic Bézier curve to the current path. The curve extends from the current point to the point (x_3, y_3), using (x_1, y_1) and (x_2, y_2) as the Bézier control points (see "Cubic Bézier Curves," below). The new current point is (x_3, y_3).
x_2 y_2 x_3 y_3	**v**	Append a cubic Bézier curve to the current path. The curve extends from the current point to the point (x_3, y_3), using the current point and (x_2, y_2) as the Bézier control points (see "Cubic Bézier Curves," below). The new current point is (x_3, y_3).
x_1 y_1 x_3 y_3	**y**	Append a cubic Bézier curve to the current path. The curve extends from the current point to the point (x_3, y_3), using (x_1, y_1) and (x_3, y_3) as the Bézier control points (see "Cubic Bézier Curves," below). The new current point is (x_3, y_3).

OPERANDS	OPERATOR	DESCRIPTION
—	**h**	Close the current subpath by appending a straight line segment from the current point to the starting point of the subpath. This operator 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 **h** operation. If the current subpath is already closed, **h** does nothing.
x y width height	**re**	Append a rectangle to the current path as a complete subpath, with lower-left corner (x, y) and dimensions *width* and *height* in user space. The operation *x y width height* re is equivalent to *x y* m (*x* + *width*) *y* l (*x* + *width*) (*y* + *height*) l *x* (*y* + *height*) l h

Cubic Bézier Curves

Curved path segments are specified as *cubic Bézier curves*. Such curves are defined by four points: the two endpoints (the current point P_0 and the final point P_3) and two *control points* P_1 and P_2. Given the coordinates of the four points, the curve is generated by varying the parameter t from 0.0 to 1.0 in the following equation:

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

When $t = 0.0$, the value of the function $R(t)$ coincides with the current point P_0; when $t = 1.0$, $R(t)$ coincides with the final point P_3. Intermediate values of t generate intermediate points along the curve. The curve does not, in general, pass through the two control points P_1 and P_2.

Cubic Bézier curves have two desirable properties:

- The curve can be very quickly split into smaller pieces for rapid rendering.
- The curve is contained within the convex hull of the four points defining the curve, most easily visualized as the polygon obtained by stretching a rubber

band around the outside of the four points. This property allows rapid testing of whether the curve lies completely outside the visible region, and hence does not have to be rendered.

The Bibliography lists several books that describe cubic Bézier curves in more depth.

The most general PDF operator for constructing curved path segments is the **c** operator, which specifies the coordinates of points P_1, P_2, and P_3 explicitly, as shown in Figure 4.8. (The starting point, P_0, is defined implicitly by the current point.)

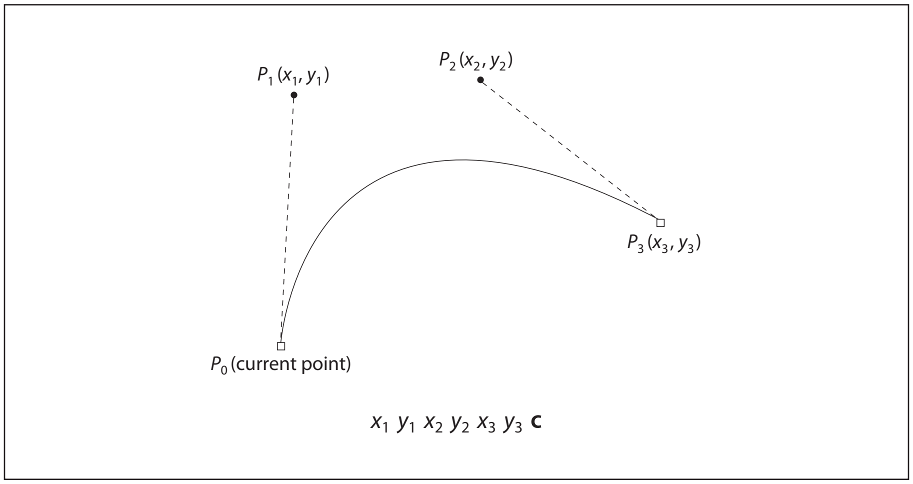


FIGURE 4.8 *Cubic Bézier curve generated by the* **c** *operator*

Two more operators, **v** and **y**, each specify one of the two control points implicitly (see Figure 4.9). In both of these cases, one control point and the final point of the curve are supplied as operands; the other control point is implied, as follows:

- For the **v** operator, the first control point coincides with initial point of the curve.
- For the **y** operator, the second control point coincides with final point of the curve.

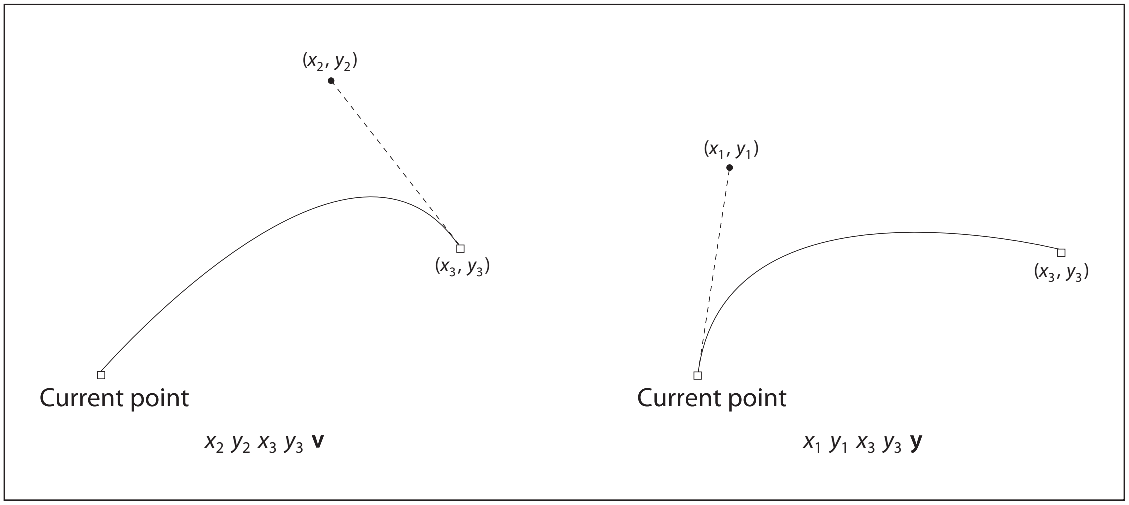


FIGURE 4.9 *Cubic Bézier curves generated by the **v** and **y** operators*

4.4.2 Path-Painting Operators

The path-painting operators end a path object, causing it to be painted on the current page in the manner that the operator specifies. The principal path-painting operators are **S** (for *stroking*) and **f** (for *filling*). Variants of these operators combine stroking and filling in a single operation or apply different rules for determining the area to be filled. Table 4.10 lists all the path-painting operators.

Stroking

The **S** operator paints a line along the current path. The stroked line follows each straight or curved segment in the path, centered on the segment with sides parallel to it. Each of the path's subpaths is treated separately.

The results of the **S** operator depend on the current settings of various parameters in the graphics state. See Section 4.3, "Graphics State," for further information on these parameters.

TABLE 4.10 Path-painting operators

OPERANDS	OPERATOR	DESCRIPTION
—	**S**	Stroke the path.
—	**s**	Close and stroke the path. This operator has the same effect as the sequence h S.
—	**f**	Fill the path, using the nonzero winding number rule to determine the region to fill (see "Nonzero Winding Number Rule" on page 202). Any subpaths that are open are implicitly closed before being filled.
—	**F**	Equivalent to **f**; included only for compatibility. Although applications that read PDF files must be able to accept this operator, those that generate PDF files should use **f** instead.
—	**f***	Fill the path, using the even-odd rule to determine the region to fill (see "Even-Odd Rule" on page 203).
—	**B**	Fill and then stroke the path, using the nonzero winding number rule to determine the region to fill. This produces the same result as constructing two identical path objects, painting the first with **f** and the second with **S**. Note, however, that the filling and stroking portions of the operation consult different values of several graphics state parameters, such as the current color. See also "Special Path-Painting Considerations" on page 528.
—	**B***	Fill and then stroke the path, using the even-odd rule to determine the region to fill. This operator produces the same result as **B**, except that the path is filled as if with **f*** instead of **f**. See also "Special Path-Painting Considerations" on page 528.
—	**b**	Close, fill, and then stroke the path, using the nonzero winding number rule to determine the region to fill. This operator has the same effect as the sequence h B. See also "Special Path-Painting Considerations" on page 528.
—	**b***	Close, fill, and then stroke the path, using the even-odd rule to determine the region to fill. This operator has the same effect as the sequence h B*. See also "Special Path-Painting Considerations" on page 528.
—	**n**	End the path object without filling or stroking it. This operator is a "path-painting no-op," used primarily for the side effect of changing the current clipping path (see Section 4.4.3, "Clipping Path Operators").

- The width of the stroked line is determined by the current line width parameter ("Line Width" on page 185).
- The color or pattern of the line is determined by the current color and color space for stroking operations.
- The line can be painted either solid or with a dash pattern, as specified by the current line dash pattern ("Line Dash Pattern" on page 187).
- If a subpath is open, the unconnected ends are treated according to the current line cap style, which may be butt, rounded, or square ("Line Cap Style" on page 186).
- Wherever two consecutive segments are connected, the joint between them is treated according to the current line join style, which may be mitered, rounded, or beveled ("Line Join Style" on page 186). Mitered joins are also subject to the current miter limit ("Miter Limit" on page 187).

  Note: *Points at which unconnected segments happen to meet or intersect receive no special treatment. In particular, "closing" a subpath with an explicit* **l** *operator rather than with* **h** *may result in a messy corner, because line caps will be applied instead of a line join.*
- The stroke adjustment parameter *(PDF 1.2)* specifies that coordinates and line widths be adjusted automatically to produce strokes of uniform thickness despite rasterization effects (Section 6.5.4, "Automatic Stroke Adjustment").

If a subpath is degenerate (consists of a single-point closed path or of two or more points at the same coordinates), the **S** operator 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, **S** produces no output, because the orientation of the caps would be indeterminate. (Note that this rule applies only to zero-length subpaths of the path being stroked, and not to zero-length dashes in a dash pattern. In the latter case, the line caps are always painted, since their orientation is determined by the direction of the underlying path.) A single-point open subpath (specified by a trailing **m** operator) produces no output.

Filling

The **f** operator uses the current nonstroking color to paint the entire region enclosed by the current path. If the path consists of several disconnected subpaths, **f** paints the insides of all subpaths, considered together. Any subpaths that are open are implicitly closed before being filled.

If a subpath is degenerate (consists of a single-point closed path or of two or more points at the same coordinates), **f** paints the single device pixel lying under that point; the result is device-dependent and not generally useful. A single-point open subpath (specified by a trailing **m** operator) produces no output.

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 **f** operator uses. Similarly, the **W** operator uses this rule to determine the inside of the current clipping path. The even-odd rule is occasionally useful for special effects or for compatibility with other graphics systems; the **f*** and **W*** 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 shown in Figure 4.10. 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.

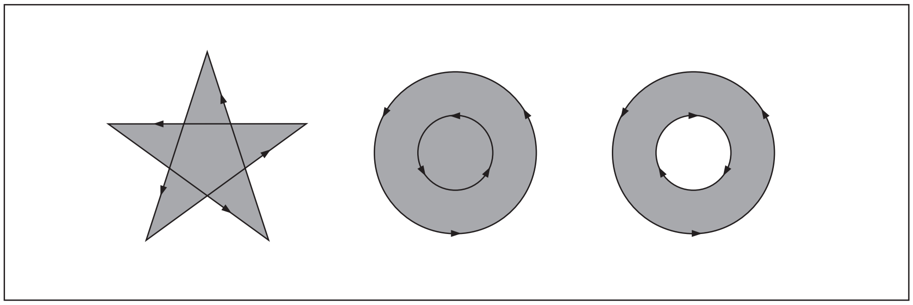


FIGURE 4.10 *Nonzero winding number rule*

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 4.11 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.

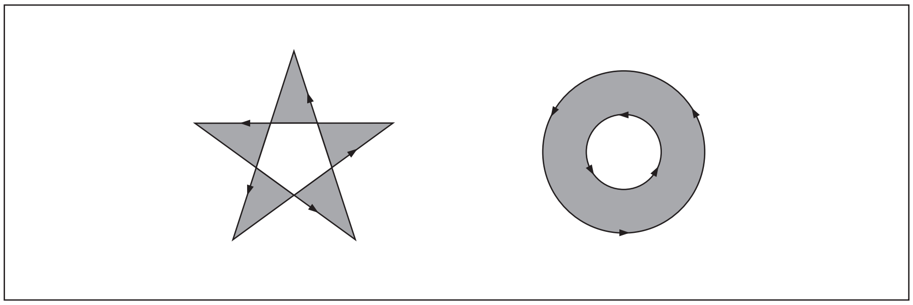


FIGURE 4.11 *Even-odd rule*

4.4.3 Clipping Path Operators

The graphics state contains a *current clipping path* that limits the regions of the page affected by 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 under "Filling," above.)

Note: *In the context of the transparent imaging model* (PDF 1.4)*, the current clipping path constrains an object's shape (see Section 7.1, "Overview of Transparency"). The effective shape is the intersection of the object's intrinsic shape with the clipping path; the source shape value is 0.0 outside this intersection. Similarly, the shape of a transparency group (defined as the union of the shapes of its constituent objects) is influenced both by the clipping path in effect when each of the objects is painted and by the one in effect at the time the group's results are painted onto its backdrop.*

The initial clipping path includes the entire page. A clipping path operator (**W** or **W***, shown in Table 4.11) may appear after the last path construction operator and before the path-painting operator that terminates a path object. Although the clipping path operator appears before the painting operator, it does not alter the clipping path at the point where it appears. Rather, it modifies the effect of the succeeding painting operator. *After* the path has been painted, the clipping path in the graphics state is set to the intersection of the current clipping path and the newly constructed path.

TABLE 4.11 Clipping path operators

OPERANDS	OPERATOR	DESCRIPTION
—	**W**	Modify the current clipping path by intersecting it with the current path, using the nonzero winding number rule to determine which regions lie inside the clipping path.
—	**W***	Modify the current clipping path by intersecting it with the current path, using the even-odd rule to determine which regions lie inside the clipping path.

***Note:** In addition to path objects, text objects can also be used for clipping; see Section 5.2.5, "Text Rendering Mode."*

The **n** operator (see Table 4.10 on page 200) is a "no-op" path-painting operator; it causes no marks to be placed on the page, but can be used with a clipping path operator to establish a new clipping path. That is, after a path has been constructed, the sequence W n will intersect that path with the current clipping path to establish a new clipping path.

There is no way to enlarge the current clipping path or to set a new clipping path without reference to the current one. However, since the clipping path is part of the graphics state, its effect can be localized to specific graphics objects by enclosing the modification of the clipping path and the painting of those objects between a pair of **q** and **Q** operators (see Section 4.3.1, "Graphics State Stack"). Execution of the **Q** operator causes the clipping path to revert to the value that was saved by the **q** operator, before the clipping path was modified.

4.5 Color Spaces

PDF includes powerful facilities for specifying the colors of graphics objects to be painted on the current page. The color facilities are divided into two parts:

- *Color specification*. A PDF file 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 viewer application reproduces colors on the raster output device by a multiple-step process that includes some combination of color conversion, gamma correction, halftoning, and scan conversion. Some aspects of this process use information that is specified in PDF. However, unlike the facilities for color specification, the color rendering facilities are device-dependent and ordinarily should not be included in a page description.

Figures 4.12 and 4.13 on pages 207 and 208 illustrate the division between PDF's (device-independent) color specification and (device-dependent) color rendering facilities. This section describes the color specification features, covering everything that most PDF documents need in order to specify colors. The facilities for controlling color rendering are described in Chapter 6; a PDF document should use these facilities only to configure or calibrate an output device or to achieve special device-dependent effects.

4.5.1 Color Values

As described in Section 4.4.2, "Path-Painting Operators," marks placed on the page by operators such as **f** and **S** 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 numeric values to specify red, green, and blue components.

Color values are interpreted according to the *current color space*, another parameter of the graphics state. A PDF content stream first selects a color space by invoking the **CS** operator (for the stroking color) or the **cs** operator (for the nonstroking color). It then selects color values within that color space with the **SC** operator (stroking) or the **sc** operator (nonstroking). There are also convenience operators—**G**, **g**, **RG**, **rg**, **K**, and **k**—that select both a color space and a color value within it in a single step. Table 4.21 on page 250 lists all the color-setting operators.

Sampled images (see Section 4.8, "Images") specify the color values of individual samples with respect to a color space designated by the image object itself. While these values are independent of the current color space and color parameters in the graphics state, all later stages of color processing treat them in exactly the same way as color values specified with the **SC** or **sc** operator.

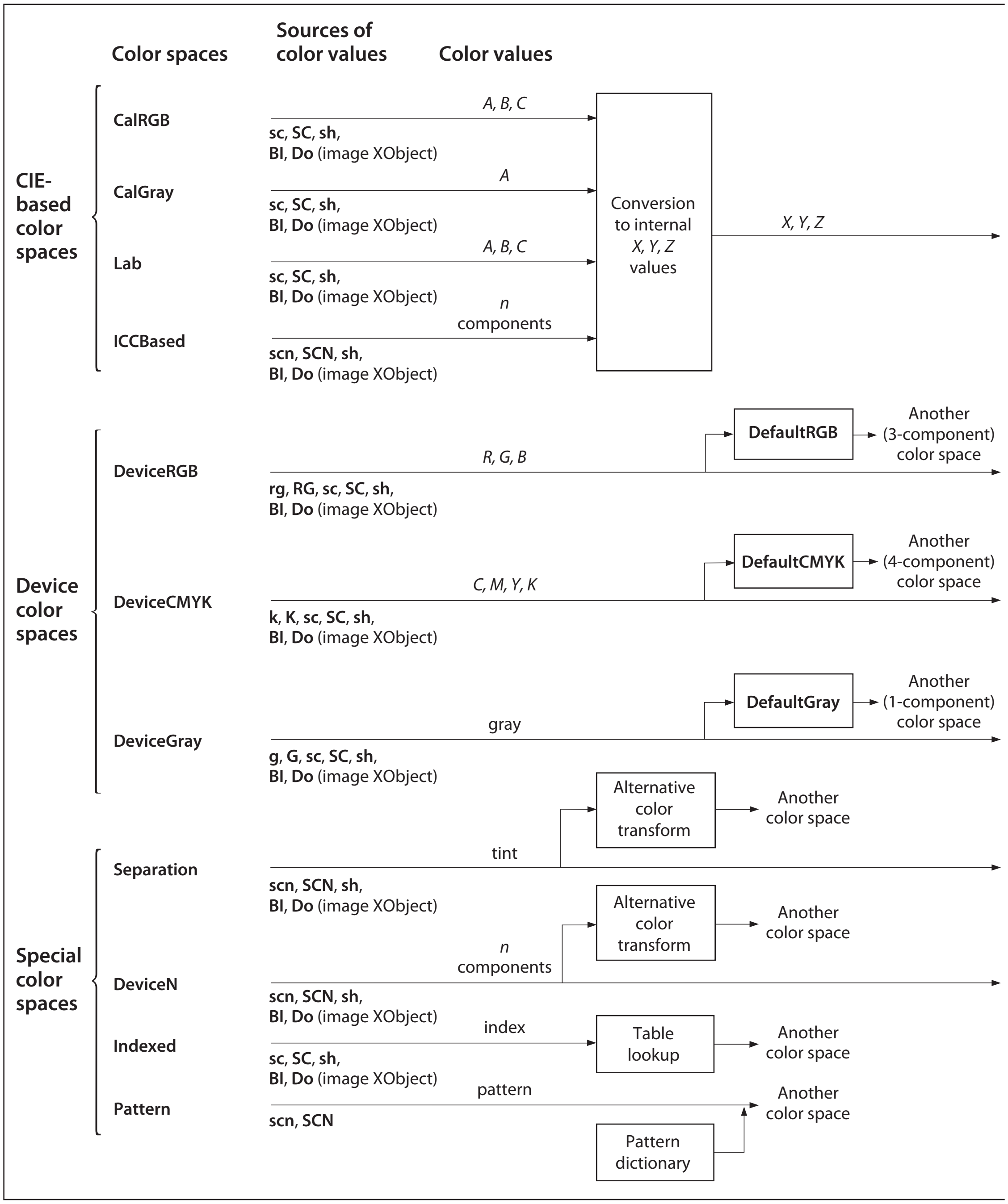

Color spaces
Sources of color values
Color values
CIE-based color spaces
CalRGB
sc, SC, sh, BI, Do (image XObject)
A, B, C
CalGray
sc, SC, sh, BI, Do (image XObject)
A
Lab
sc, SC, sh, BI, Do (image XObject)
A, B, C
ICCBased
scn, SCN, sh, BI, Do (image XObject)
n components
Conversion to internal X, Y, Z values
X, Y, Z
Device color spaces
DeviceRGB
rg, RG, sc, SC, sh, BI, Do (image XObject)
R, G, B
DefaultRGB
Another (3-component) color space
DeviceCMYK
k, K, sc, SC, sh, BI, Do (image XObject)
C, M, Y, K
DefaultCMYK
Another (4-component) color space
DeviceGray
g, G, sc, SC, sh, BI, Do (image XObject)
gray
DefaultGray
Another (1-component) color space
Special color spaces
Separation
scn, SCN, sh, BI, Do (image XObject)
tint
Alternative color transform
Another color space
DeviceN
scn, SCN, sh, BI, Do (image XObject)
n components
Alternative color transform
Another color space
Indexed
sc, SC, sh, BI, Do (image XObject)
index
Table lookup
Another color space
Pattern
scn, SCN
pattern
Pattern dictionary
Another color space

FIGURE 4.12 *Color specification*

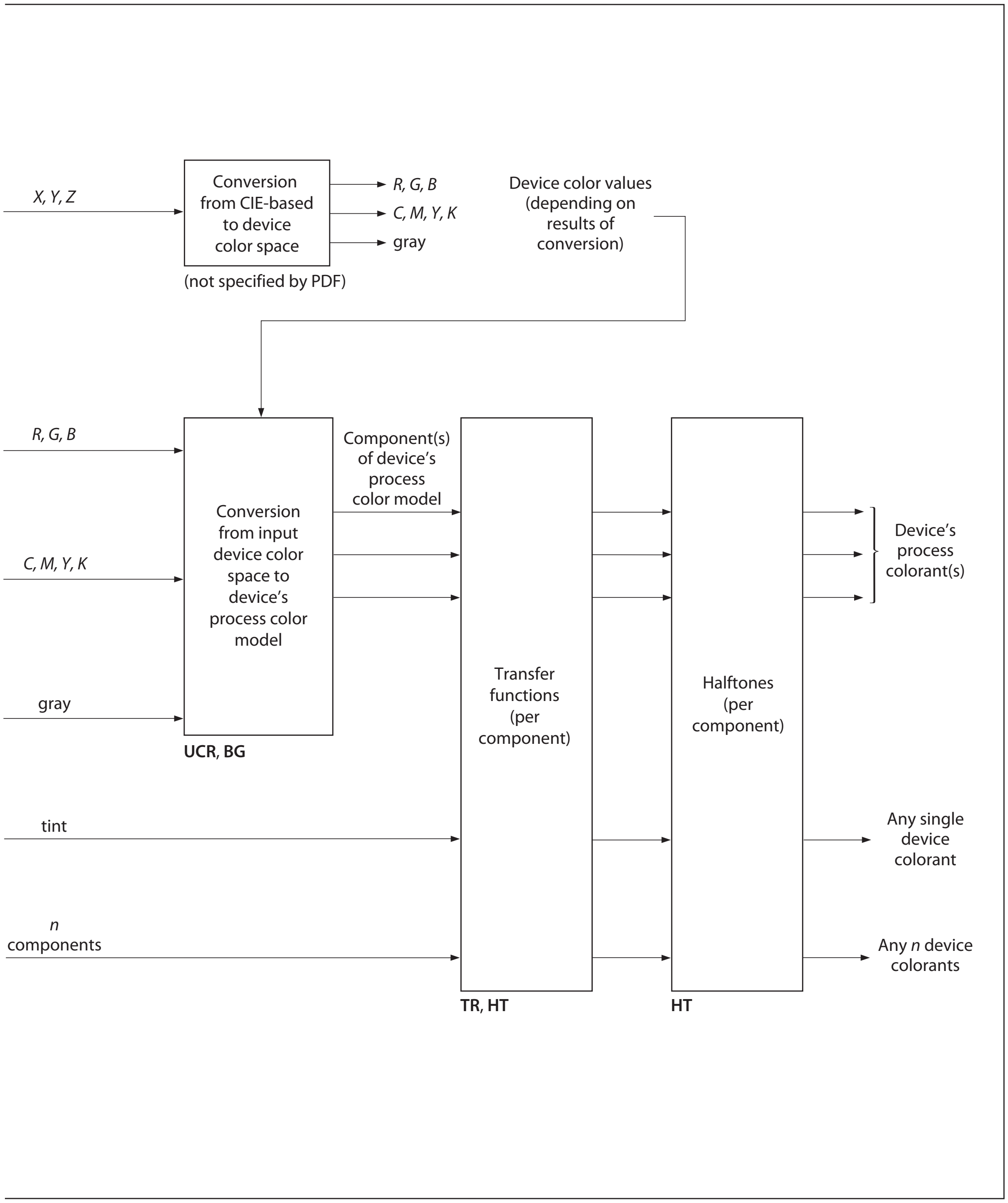

X, Y, Z
Conversion from CIE-based to device color space
(not specified by PDF)
R, G, B
C, M, Y, K
gray
Device color values (depending on results of conversion)
R, G, B
C, M, Y, K
gray
Conversion from input device color space to device's process color model
UCR, BG
Component(s) of device's process color model
tint
n components
Transfer functions (per component)
TR, HT
Halftones (per component)
HT
Device's process colorant(s)
Any single device colorant
Any *n* device colorants

FIGURE 4.13 *Color rendering*

4.5.2 Color Space Families

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 broad 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 grayscale, *RGB* (red-green-blue), and *CMYK* (cyan-magenta-yellow-black), corresponding to the color space families **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**. Since each of these families consists of just a single color space with no parameters, they are often loosely referred to as the **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces.
- *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 **CalGray**, **CalRGB**, **Lab**, and **ICC-Based**. 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.

Table 4.12 summarizes the color space families supported by PDF. (See implementation note 39 in Appendix H.)

TABLE 4.12 Color space families

DEVICE	CIE-BASED	SPECIAL
DeviceGray *(PDF 1.1)*	**CalGray** *(PDF 1.1)*	**Indexed** *(PDF 1.1)*
DeviceRGB *(PDF 1.1)*	**CalRGB** *(PDF 1.1)*	**Pattern** *(PDF 1.2)*
DeviceCMYK *(PDF 1.1)*	**Lab** *(PDF 1.1)*	**Separation** *(PDF 1.2)*
	ICCBased *(PDF 1.3)*	**DeviceN** *(PDF 1.3)*

A color space is defined by an array object whose first element is a name object identifying the color space family. The remaining array elements, if any, are parameters that further characterize the color space; their number and types vary according to the particular family. For families that do not require parameters, the color space can be specified simply by the family name itself instead of an array.

There are two principal ways in which a color space can be specified:

- Within a content stream, the **CS** or **cs** operator establishes the current color space parameter in the graphics state. The operand is always a name object, which either identifies one of the color spaces that need no additional parameters (**DeviceGray**, **DeviceRGB**, **DeviceCMYK**, or some cases of **Pattern**) or is used as a key in the **ColorSpace** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"). In the latter case, the value of the dictionary entry is in turn a color space array or name. A color space array is never permitted inline within a content stream.
- Outside a content stream, certain objects, such as image XObjects, specify a color space as an explicit parameter, often associated with the key **ColorSpace**. In this case, the color space array or name is always defined directly as a PDF object, not by an entry in the **ColorSpace** resource subdictionary. This convention also applies when color spaces are defined in terms of other color spaces.

The following operators set the current color space and current color parameters in the graphics state:

- **CS** sets the stroking color space; **cs** sets the nonstroking color space.
- **SC** and **SCN** set the stroking color; **sc** and **scn** set the nonstroking color. Depending on the color space, these operators require one or more operands, each specifying one component of the color value.
- **G**, **RG**, and **K** set the stroking color space implicitly and the stroking color as specified by the operands; **g**, **rg**, and **k** do the same for the nonstroking color space and color.

4.5.3 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 PDF document to control colors precisely for a particular device, but the results may not be consistent from one device to another.

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 methods of color specification, called *additive* and *subtractive* color (see Plate 1). The most widely used forms of these two types of color specification are known as *RGB* and *CMYK*, respectively, for the names of the primary colors on which they're based. The corresponding device color spaces are as follows:

- **DeviceGray** controls the intensity of achromatic light, on a scale from black to white.
- **DeviceRGB** controls the intensities of red, green, and blue light, the three additive primary colors used in displays.
- **DeviceCMYK** controls the concentrations of cyan, magenta, yellow, and black inks, the four subtractive process colors used in printing.

Although the notion of explicit color spaces is a PDF 1.1 feature, the operators for specifying colors in the device color spaces—**G**, **g**, **RG**, **rg**, **K**, and **k**—are available in all versions of PDF. Beginning with PDF 1.2, colors specified in device color spaces can optionally be remapped systematically into other color spaces; see "Default Color Spaces" on page 227.

Note: *In the transparent imaging model* (PDF 1.4), *the use of device color spaces is subject to special treatment within a transparency group whose group color space is CIE-based (see Sections 7.3, "Transparency Groups," and 7.5.5, "Transparency Group XObjects"). In particular, the device color space operators should be used only if device color spaces have been remapped to CIE-based spaces by means of the default color space mechanism. Otherwise, the results will be implementation-dependent and unpredictable.*

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 4.2 shows alternative ways to select the **DeviceGray** color space and a specific gray level within that space for stroking operations.

Example 4.2

```
/DeviceGray  CS                    % Set DeviceGray color space
gray  SC                           % Set gray level

gray  G                            % Set both in one operation
```

The **CS** and **SC** operators select the current stroking color space and current stroking color separately; **G** sets them in combination. (The **cs**, **sc**, and **g** operators perform the same functions for nonstroking operations.) Setting either current color space to **DeviceGray** initializes the corresponding current color to 0.0.

DeviceRGB Color Space

Colors in the **DeviceRGB** color space are specified according to the additive *RGB* (red-green-blue) color model, in which color values are defined by three components representing the intensities of the additive primary colorants red, green, and blue. Each component is specified by a number in the range 0.0 to 1.0, where 0.0 denotes the complete absence of a primary component and 1.0 denotes maximum intensity. If all three components 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.

Example 4.3 shows alternative ways to select the **DeviceRGB** color space and a specific color within that space for stroking operations.

Example 4.3

```
/DeviceRGB  CS                     % Set DeviceRGB color space
red  green  blue  SC               % Set color

red  green  blue  RG               % Set both in one operation
```

The **CS** and **SC** operators select the current stroking color space and current stroking color separately; **RG** sets them in combination. (The **cs**, **sc**, and **rg** operators perform the same functions for nonstroking operations.) Setting either current color space to **DeviceRGB** initializes the red, green, and blue components of the corresponding current color to 0.0.

DeviceCMYK Color Space

The **DeviceCMYK** color space allows colors to be specified according to the subtractive *CMYK* (cyan-magenta-yellow-black) model typical of printers and other paper-based output devices. In theory, each of the three standard *process colorants* used in printing (cyan, magenta, and yellow) absorbs one of the additive primary colors (red, green, and blue, respectively). Black, a fourth standard process colorant, absorbs all of the additive primaries in equal amounts. The four components in a **DeviceCMYK** color value represent the concentrations of these process colorants. Each component is specified by a number in the range 0.0 to 1.0, where 0.0 denotes the complete absence of a process colorant (that is, absorbs none of the corresponding additive primary) and 1.0 denotes maximum concentration (absorbs as much as possible of the additive primary). Note that the sense of these numbers is opposite to that of *RGB* color components.

Example 4.4 shows alternative ways to select the **DeviceCMYK** color space and a specific color within that space for stroking operations.

Example 4.4

```
/DeviceCMYK  CS                          % Set DeviceCMYK color space
cyan  magenta  yellow  black  SC         % Set color

cyan  magenta  yellow  black  K          % Set both in one operation
```

The **CS** and **SC** operators select the current stroking color space and current stroking color separately; **K** sets them in combination. (The **cs**, **sc**, and **k** operators perform the same functions for nonstroking operations.) Setting either current color space to **DeviceCMYK** initializes the cyan, magenta, and yellow components of the corresponding current color to 0.0 and the black component to 1.0.

4.5.4 CIE-Based Color Spaces

Calibrated color in PDF is defined in terms of an international standard used in the graphic arts, television, and printing industries. *CIE-based* color spaces enable a page description to specify color values in a way that is related to human visual perception. The goal is for the same color specification to produce consistent results on different output devices, within the limitations of each device; Plate 2 illustrates the kind of variation in color reproduction that can result from the use of uncalibrated color on different devices. PDF 1.1 supports three CIE-based color space families, named **CalGray**, **CalRGB**, and **Lab**; PDF 1.3 adds a fourth, named **ICCBased**.

***Note:** In PDF 1.1, a color space family named **CalCMYK** was partially defined, with the expectation that its definition would be completed in a future version. However, this is no longer being considered. PDF 1.3 and later versions support calibrated four-component color spaces by means of ICC profiles (see "ICCBased Color Spaces" on page 222). PDF consumer applications should ignore **CalCMYK** color space attributes and render colors specified in this family as if they had been specified using **DeviceCMYK**.*

The details 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 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. The **CalRGB** and **Lab** color spaces *(PDF 1.1)* are special cases of three-component CIE-based color spaces, known as *CIE-based ABC* color spaces. These spaces are defined in terms of a two-stage, nonlinear transformation of the CIE 1931 *XYZ* space. The formulation of such 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.

Color values in a CIE-based *ABC* color space have three components, arbitrarily named *A*, *B*, and *C*. The first stage transforms these components by first forcing their values to a specified range, then applying *decoding functions*, and finally multiplying the results by a 3-by-3 matrix, producing three intermediate components arbitrarily named *L*, *M*, and *N*. The second stage transforms these intermediate components in a similar fashion, producing the final *X*, *Y*, and *Z* components of the CIE 1931 *XYZ* space (see Figure 4.14).

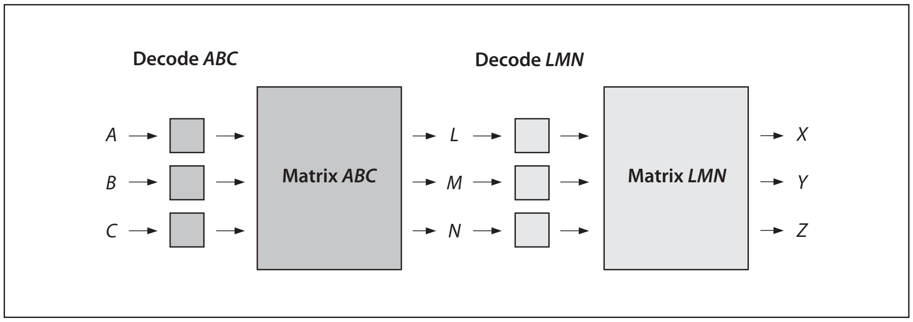


FIGURE 4.14 *Component transformations in a CIE-based* ABC *color space*

Color spaces in the CIE-based families are defined by an array

[*name dictionary*]

where *name* is the name of the family and *dictionary* is a dictionary containing parameters that further characterize the space. The entries in this dictionary have specific interpretations that vary depending on the color space; some entries are required and some are optional. See the sections on specific color space families, below, for details.

Setting the current stroking or nonstroking color space to any CIE-based color space initializes all components of the corresponding current color to 0.0 (unless the range of valid values for a given component does not include 0.0, in which case the nearest valid value is substituted.)

Note: *The model and terminology used here—*CIE-based ABC *(above) and* CIE-based A *(below)—are derived from the PostScript language, which supports these color space families in their full generality. PDF supports specific useful cases of CIE-based* ABC *and CIE-based* A *spaces; most others can be represented as* ***ICCBased*** *spaces.*

CalGray Color Spaces

A **CalGray** color space *(PDF 1.1)* is a special case of a single-component CIE-based color space, known as a *CIE-based A* color space. This type of space is the one-dimensional (and usually achromatic) analog of CIE-based *ABC* spaces. Color values in a CIE-based *A* space have a single component, arbitrarily named *A*. Figure 4.15 illustrates the transformations of the *A* component to *X*, *Y*, and *Z* components of the CIE 1931 *XYZ* space.

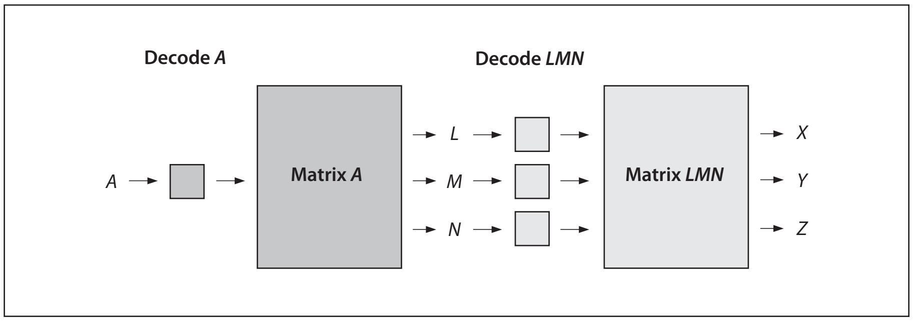


FIGURE 4.15 *Component transformations in a CIE-based* A *color space*

A **CalGray** color space is a CIE-based *A* color space with only one transformation stage instead of two. In this type of space, *A* represents the gray component of a calibrated gray space. This component must be in the range 0.0 to 1.0. The decoding function (denoted by "Decode *A*" in Figure 4.15) is a gamma function whose coefficient is specified by the **Gamma** entry in the color space dictionary (see Table 4.13). The transformation matrix denoted by "Matrix *A*" in the figure is derived from the dictionary's **WhitePoint** entry, as described below. Since there is no second transformation stage, "Decode *LMN*" and "Matrix *LMN*" are implicitly taken to be identity transformations.

TABLE 4.13 Entries in a CalGray color space dictionary

KEY	TYPE	VALUE
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 "CalRGB Color Spaces," below, for further discussion. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.0.
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 "CalRGB Color Spaces," below, for further discussion. All three of these numbers must be nonnegative. Default value: [0.0 0.0 0.0].
Gamma	number	*(Optional)* A number *G* defining the gamma for the gray *(A)* component. *G* must be positive and will generally be greater than or equal to 1. Default value: 1.

The transformation defined by the **Gamma** and **WhitePoint** entries is

$$
\begin{aligned}
X &= L = X_W \times A^G \\
Y &= M = Y_W \times A^G \\
Z &= N = Z_W \times A^G
\end{aligned}
$$

In other words, the *A* component is first decoded by the gamma function, and the result is multiplied by the components of the white point to obtain the *L*, *M*, and *N* components of the intermediate representation. Since there is no second stage, these are also the *X*, *Y*, and *Z* components of the final representation.

The following examples illustrate interesting and useful special cases of **CalGray** spaces. Example 4.5 establishes a space consisting of the *Y* dimension of the CIE 1931 *XYZ* space with the CCIR XA/11–recommended D65 white point.

Example 4.5

```
[ /CalGray
      << /WhitePoint [0.9505 1.0000 1.0890] >>
]
```

Example 4.6 establishes a calibrated gray space with the CCIR XA/11–recommended D65 white point and opto-electronic transfer function.

Example 4.6

```
[ /CalGray
      << /WhitePoint [0.9505 1.0000 1.0890]
         /Gamma 2.222
      >>
]
```

CalRGB Color Spaces

A **CalRGB** color space is a CIE-based *ABC* color space with only one transformation stage instead of two. In this type of space, *A*, *B*, and *C* represent calibrated red, green, and blue color values. These three color components must be in the range 0.0 to 1.0; component values falling outside that range will be adjusted to the nearest valid value without error indication. The decoding functions (denoted by "Decode *ABC*" in Figure 4.14 on page 215) are gamma functions whose coefficients are specified by the **Gamma** entry in the color space dictionary (see Table 4.14). The transformation matrix denoted by "Matrix *ABC*" in Figure 4.14 is defined by the dictionary's **Matrix** entry. Since there is no second transformation stage, "Decode *LMN*" and "Matrix *LMN*" are implicitly taken to be identity transformations.

TABLE 4.14 Entries in a CalRGB color space dictionary

KEY	TYPE	VALUE
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 discussion. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.0.
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 discussion. All three of these numbers must be nonnegative. Default value: [0.0 0.0 0.0].
Gamma	array	*(Optional)* An array of three numbers [G_R G_G G_B] specifying the gamma for the red, green, and blue *(A*, *B*, and *C)* components of the color space. Default value: [1.0 1.0 1.0].
Matrix	array	*(Optional)* An array of nine numbers [X_A Y_A Z_A X_B Y_B Z_B X_C Y_C Z_C] specifying the linear interpretation of the decoded *A*, *B*, and *C* components of the color space with respect to the final *XYZ* representation. Default value: the identity matrix [1 0 0 0 1 0 0 0 1].

The **WhitePoint** and **BlackPoint** entries in the color space dictionary control the overall effect of the CIE-based gamut mapping function described in Section 6.1, "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 input 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 transformation defined by the **Gamma** and **Matrix** entries in the **CalRGB** color space dictionary is

$$\begin{aligned} X &= L = X_A \times A^{G_R} + X_B \times B^{G_G} + X_C \times C^{G_B} \\ Y &= M = Y_A \times A^{G_R} + Y_B \times B^{G_G} + Y_C \times C^{G_B} \\ Z &= N = Z_A \times A^{G_R} + Z_B \times B^{G_G} + Z_C \times C^{G_B} \end{aligned}$$

In other words, the *A*, *B*, and *C* components are first decoded individually by the gamma functions. The results are treated as a three-element vector and multiplied by **Matrix** (a 3-by-3 matrix) to obtain the *L*, *M*, and *N* components of the intermediate representation. Since there is no second stage, these are also the *X*, *Y*, and *Z* components of the final representation.

Example 4.7 shows an example of a **CalRGB** color space for the CCIR XA/11–recommended D65 white point with 1.8 gammas and Sony Trinitron® phosphor chromaticities.

Example 4.7

```
[ /CalRGB
      << /WhitePoint [0.9505 1.0000 1.0890]
         /Gamma [1.8000 1.8000 1.8000]
         /Matrix [ 0.4497 0.2446 0.0252
                   0.3163 0.6720 0.1412
                   0.1845 0.0833 0.9227
                 ]
      >>
]
```

In some cases, the parameters of a **CalRGB** color space may be 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, **Matrix** and **WhitePoint** can be found as follows:

$$z = y_W \times ((x_G - x_B) \times y_R - (x_R - x_B) \times y_G + (x_R - x_G) \times y_B)$$

$$Y_A = \frac{y_R}{R} \times \frac{(x_G - x_B) \times y_W - (x_W - x_B) \times y_G + (x_W - x_G) \times y_B}{z}$$

$$X_A = Y_A \times \frac{x_R}{y_R} \qquad Z_A = Y_A \times \left(\frac{1 - x_R}{y_R} - 1 \right)$$

$$Y_B = -\frac{y_G}{G} \times \frac{(x_R - x_B) \times y_W - (x_W - x_B) \times y_R + (x_W - x_R) \times y_B}{z}$$

$$X_B = Y_B \times \frac{x_G}{y_G} \qquad Z_B = Y_B \times \left(\frac{1 - x_G}{y_G} - 1 \right)$$

$$Y_C = \frac{y_B}{B} \times \frac{(x_R - x_G) \times y_W - (x_W - x_G) \times y_R + (x_W - x_R) \times y_G}{z}$$

$$X_C = Y_C \times \frac{x_B}{y_B} \qquad Z_C = Y_C \times \left(\frac{1 - x_B}{y_B} - 1 \right)$$

$$X_W = X_A \times R + X_B \times G + X_C \times B$$
$$Y_W = Y_A \times R + Y_B \times G + Y_C \times B$$
$$Z_W = Z_A \times R + Z_B \times G + Z_C \times B$$

Lab Color Spaces

A **Lab** color space is a CIE-based *ABC* color space with two transformation stages (see Figure 4.14 on page 215). In a this type of space, *A*, *B*, and *C* represent the *L**, *a**, and *b** components of a CIE 1976 *L*a*b** space. The range of the first *(L*)* component is always 0 to 100; the ranges of the second and third *(a** and *b*)* components are defined by the **Range** entry in the color space dictionary (see Table 4.15).

Plate 3 illustrates the coordinates of a typical **Lab** color space; Plate 4 compares the gamuts (ranges of representable colors) for *L*a*b**, *RGB*, and *CMYK* spaces.

TABLE 4.15 Entries in a Lab color space dictionary

KEY	TYPE	VALUE
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 "CalRGB Color Spaces" on page 217 for further discussion. The numbers X_W and Z_W must be positive, and Y_W must be equal to 1.0.
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 "CalRGB Color Spaces" on page 217 for further discussion. All three of these numbers must be nonnegative. Default value: [0.0 0.0 0.0].
Range	array	*(Optional)* An array of four numbers $[a_{\text{min}}\ a_{\text{max}}\ b_{\text{min}}\ b_{\text{max}}]$ specifying the range of valid values for the a^* and b^* *(B* and *C)* components of the color space—that is, $$a_{\text{min}} \leq a^* \leq a_{\text{max}}$$ and $$b_{\text{min}} \leq b^* \leq b_{\text{max}}$$ Component values falling outside the specified range will be adjusted to the nearest valid value without error indication. Default value: [−100 100 −100 100].

A **Lab** color space does not specify explicit decoding functions or matrix coefficients for either stage of the transformation from $L^*a^*b^*$ space to *XYZ* space (denoted by "Decode *ABC*," "Matrix *ABC*," "Decode *LMN*," and "Matrix *LMN*" in Figure 4.14 on page 215). Instead, these parameters have constant implicit values. The first transformation stage is defined by the equations

$$\begin{aligned}
L &= \frac{L^* + 16}{116} + \frac{a^*}{500} \\
M &= \frac{L^* + 16}{116} \\
N &= \frac{L^* + 16}{116} - \frac{b^*}{200}
\end{aligned}$$

The second transformation stage is given by

$$\begin{aligned} X &= X_W \times g(L) \\ Y &= Y_W \times g(M) \\ Z &= Z_W \times g(N) \end{aligned}$$

where the function $g(x)$ is defined as

$$\begin{aligned} g(x) &= x^3 && \text{if } x \geq \frac{6}{29} \\ g(x) &= \frac{108}{841} \times \left(x - \frac{4}{29}\right) && \text{otherwise} \end{aligned}$$

Example 4.8 defines 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.

Example 4.8

```
[ /Lab
        << /WhitePoint [0.9505 1.0000 1.0890]
           /Range [−128 127 −128 127]
        >>
]
```

ICCBased Color Spaces

ICCBased color spaces *(PDF 1.3)* are based on a cross-platform *color profile* as defined by the International Color Consortium (ICC). Unlike the **CalGray**, **CalRGB**, and **Lab** color spaces, which are characterized by entries in the color space dictionary, an **ICCBased** color space is characterized by a sequence of bytes in a standard format. Details of the profile format can be found in the ICC specification (see the Bibliography).

An **ICCBased** color space is specified as an array:

```
[/ICCBased stream]
```

The stream contains the ICC profile. Besides the usual entries common to all streams (see Table 3.4 on page 38), the profile stream has the additional entries listed in Table 4.16.

TABLE 4.16 Additional entries specific to an ICC profile stream dictionary

KEY	TYPE	VALUE
N	integer	*(Required)* The number of color components in the color space described by the ICC profile data. This number must match the number of components actually in the ICC profile. As of PDF 1.4, **N** must be 1, 3, or 4.
Alternate	array or name	*(Optional)* An alternate color space to be used in case the one specified in the stream data is not supported (for example, by viewer applications designed for earlier versions of PDF). The alternate space may be any valid color space (except a **Pattern** color space) that has the number of components specified by **N**. If this entry is omitted and the viewer application does not understand the ICC profile data, the color space used will be **DeviceGray**, **DeviceRGB**, or **DeviceCMYK**, depending on whether the value of **N** is 1, 3, or 4, respectively. ***Note:*** *Note that there is no conversion of source color values, such as a tint transformation, when using the alternate color space. Color values that are within the range of the* ***ICCBased*** *color space might not be within the range of the alternate color space. In this case, the nearest values within the range of the alternate space will be substituted.*
Range	array	*(Optional)* An array of 2 × **N** numbers [min_0 max_0 min_1 max_1 …] specifying the minimum and maximum valid values of the corresponding color components. These values must match the information in the ICC profile. Default value: [0.0 1.0 0.0 1.0 …].
Metadata	stream	*(Optional; PDF 1.4)* A *metadata stream* containing metadata for the color space (see Section 10.2.2, "Metadata Streams").

The ICC specification is an evolving standard. The **ICCBased** color spaces supported in PDF 1.3 are based on ICC specification version 3.3; those in PDF 1.4 are based on the ICC specification ICC.1:1998-09 and its addendum ICC.1A:1999-04; those in PDF 1.5 are based on the ICC specification ICC.1:2001-12. (Earlier versions of the ICC specification are also supported.) This has the following consequences for producers and consumers of PDF:

- A consumer that supports a given PDF version is required to support ICC profiles conforming to the corresponding version (and earlier versions) of the ICC specification, as described above. It may optionally support later ICC versions.

- For the most predictable and consistent results, a producer of a given PDF version should embed only profiles conforming to the corresponding version of the ICC specification.
- A PDF producer may embed profiles conforming to a later ICC version, with the understanding that the results will vary depending on the capabilities of the consumer. The consumer might process the profile while ignoring newer features, or it might fail altogether to process the profile. In light of this, it is recommended that the producer provide an alternate color space (**Alternate** entry in the **ICCBased** color space dictionary) containing a profile that is appropriate for the PDF version.

As of version 1.4, PDF supports only the profile types shown in Table 4.17; other types may be supported in the future. (In particular, note that *XYZ* and 16-bit *L*a*b** profiles are not supported.) Each of the indicated fields must have one of the values listed for that field in the second column of the table. (Profiles must satisfy *both* the criteria shown in the table.) The terminology is taken from the ICC specifications.

TABLE 4.17 ICC profile types

HEADER FIELD	REQUIRED VALUE
deviceClass	icSigInputClass ('scnr') icSigDisplayClass ('mntr') icSigOutputClass ('prtr') icSigColorSpaceClass ('spac')
colorSpace	icSigGrayData ('GRAY') icSigRgbData ('RGB ') icSigCmykData ('CMYK') icSigLabData ('Lab ')

The terminology used in PDF color spaces and ICC color profiles is similar, but sometimes the same terms are used with different meanings. For example, the default value for each component in an **ICCBased** color space is 0. The range of each color component is a function of the color space specified by the profile and is indicated in the ICC specification. The ranges for several ICC color spaces are shown in Table 4.18.

TABLE 4.18 Ranges for typical ICC color spaces

ICC COLOR SPACE	COMPONENT RANGES
Gray	[0.0 1.0]
RGB	[0.0 1.0]
CMYK	[0.0 1.0]
L*a*b*	*L**: [0 100]; *a** and *b**: [−128 127]

Since the **ICCBased** color space is being used as a source color space, only the "to CIE" profile information (*AToB* in ICC terminology) is used; the "from CIE" (*BToA*) information is ignored when present. An ICC profile may also specify a *rendering intent*, but PDF viewer applications ignore this information; the rendering intent is specified in PDF by a separate parameter (see "Rendering Intents" on page 230).

***Note:** The requirements stated above apply to an **ICCBased** color space that is used to specify the source colors of graphics objects. When such a space is used as the blending color space for a transparency group in the transparent imaging model (see Sections 7.2.3, "Blending Color Space"; 7.3, "Transparency Groups"; and 7.5.5, "Transparency Group XObjects"), it must have both "to CIE"* (AToB) *and "from CIE"* (BToA) *information. This is because the group color space is used as both the destination for objects being painted within the group and the source for the group's results. ICC profiles are also used in specifying* output intents *for matching the color characteristics of a PDF document with those of a target output device or production environment. When used in this context, they are subject to still other constraints on the "to CIE" and "from CIE" information; see Section 10.10.4, "Output Intents," for details.*

The representations of **ICCBased** color spaces are less compact than **CalGray**, **CalRGB**, and **Lab**, but can represent a wider range of color spaces. In those cases where a given color space can be expressed by more than one of the CIE-based color space families, the resulting colors are expected to be rendered similarly, regardless of the method selected for representation.

One particular color space is the so-called "standard *RGB*" or *sRGB*, defined in the International Electrotechnical Commission (IEC) document *Colour Measurement and Management in Multimedia Systems and Equipment* (see the Bibliography). In PDF, the *sRGB* color space can be expressed precisely only as an **ICCBased** space, although it can be approximated by a **CalRGB** space.

Example 4.9 shows an **ICCBased** color space for a typical three-component *RGB* space. The profile's data has been encoded in hexadecimal representation for readability; in actual practice, a lossless decompression filter such as **FlateDecode** should be used.

Example 4.9

```
10 0 obj                                    % Color space
   [/ICCBased 15 0 R]
endobj

15 0 obj                                    % ICC profile stream
   << /N 3
        /Alternate /DeviceRGB
        /Length 1605
        /Filter /ASCIIHexDecode
   >>
stream
00 00 02 0C 61 70 70 6C 02 00 00 00 6D 6E 74 72
52 47 42 20 58 59 5A 20 07 CB 00 02 00 16 00 0E
00 22 00 2C 61 63 73 70 41 50 50 4C 00 00 00 00
61 70 70 6C 00 00 04 01 00 00 00 00 00 00 00 02
00 00 00 00 00 00 F6 D4 00 01 00 00 00 00 D3 2B
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 09 64 65 73 63 00 00 00 F0 00 00 00 71
72 58 59 5A 00 00 01 64 00 00 00 14 67 58 59 5A
00 00 01 78 00 00 00 14 62 58 59 5A 00 00 01 8C
00 00 00 14 72 54 52 43 00 00 01 A0 00 00 00 0E
67 54 52 43 00 00 01 B0 00 00 00 0E 62 54 52 43
00 00 01 C0 00 00 00 0E 77 74 70 74 00 00 01 D0
00 00 00 14 63 70 72 74 00 00 01 E4 00 00 00 27
64 65 73 63 00 00 00 00 00 00 00 17 41 70 70 6C
65 20 31 33 22 20 52 47 42 20 53 74 61 6E 64 61
72 64 00 00 00 00 00 00 00 00 00 00 00 17 41 70
70 6C 65 20 31 33 22 20 52 47 42 20 53 74 61 6E
64 61 72 64 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 58 59 5A 58 59 5A 20 00 00 00 00 00 00 63 0A
```

```
00 00 35 0F 00 00 03 30 58 59 5A 20 00 00 00 00
00 00 53 3D 00 00 AE 37 00 00 15 76 58 59 5A 20
00 00 00 00 00 00 40 89 00 00 1C AF 00 00 BA 82
63 75 72 76 00 00 00 00 00 00 00 01 01 CC 63 75
63 75 72 76 00 00 00 00 00 00 00 01 01 CC 63 75
63 75 72 76 00 00 00 00 00 00 00 01 01 CC 58 59
58 59 5A 20 00 00 00 00 00 00 F3 1B 00 01 00 00
00 01 67 E7 74 65 78 74 00 00 00 00 20 43 6F 70
79 72 69 67 68 74 20 41 70 70 6C 65 20 43 6F 6D
70 75 74 65 72 73 20 31 39 39 34 00 >
endstream
endobj
```

Default Color Spaces

Specifying colors in a device color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**) makes them device-dependent. By setting *default color spaces (PDF 1.1)*, a PDF document can request that such colors be systematically transformed *(remapped)* into device-independent CIE-based color spaces. This capability can be useful in a variety of circumstances, such as the following:

- A document originally intended for one output device is redirected to a different device.
- A document is intended to be compatible with viewer applications designed for earlier versions of PDF, and thus cannot specify CIE-based colors directly.
- Color corrections or rendering intents need to be applied to device colors (see "Rendering Intents" on page 230).

A color space is selected for painting each graphics object. This is either the current color space parameter in the graphics state or a color space given as an entry in an image XObject, inline image, or shading dictionary. Regardless of how the color space is specified, it may be subject to remapping as described below.

When a device color space is selected, the **ColorSpace** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries") is checked for the presence of an entry designating a corresponding default color space (**DefaultGray**, **DefaultRGB**, or **DefaultCMYK**, corresponding to **DeviceGray**, **DeviceRGB**, or **DeviceCMYK**, respectively). If such an entry is present, its value is used as the color space for the operation currently being performed. (If the viewer application does not recognize this color space, no remapping will occur; the original device color space will be used.)

Color values in the original device color space are passed unchanged to the default color space, which must have the same number of components as the original space. The default color space should be chosen to be compatible with the original, taking into account the components' ranges and whether the components are additive or subtractive. If a color value lies outside the range of the default color space, it will be adjusted to the nearest valid value.

Note: *Any color space other than a* ***Lab****,* ***Indexed****, or* ***Pattern*** *color space may be used as a default color space, provided that it is compatible with the original device color space as described above.*

If the selected space is a special color space based on an underlying device color space, the default color space will be used in place of the underlying space. This applies to the following:

- The underlying color space of a **Pattern** color space
- The base color space of an **Indexed** color space
- The alternate color space of a **Separation** or **DeviceN** color space (but only if the alternate color space is actually selected)

See Section 4.5.5, "Special Color Spaces," for details on these color spaces.

Note: *Note that there is no conversion of color values, such as a tint transformation, when using the default color space. Color values that are within the range of the device color space might not be within the range of the default color space (particularly if the default is an* ***ICCBased*** *color space). In this case, the nearest values within the range of the default space will be used. For this reason, a* ***Lab*** *color space is not permitted as the* ***DefaultRGB*** *color space.*

Implicit Conversion of CIE-Based Color Spaces

In workflows in which PDF documents are intended for rendering on a specific target output device (such as a printing press with particular inks and media), it is often useful to specify the source colors for some or all of a document's objects in a CIE-based color space that matches the calibration of the intended device. The resulting document, while tailored to the specific characteristics of the target device, remains device-independent and will produce reasonable results if retargeted to a different output device. However, the expectation is that if the document is printed on the intended target device, source colors that have been specified in a color space matching the calibration of the device will pass through unchanged, without conversion to and from the intermediate CIE 1931 *XYZ* space as depicted in Figure 4.14 on page 215.

In particular, when colors intended for a *CMYK* output device are specified in an **ICCBased** color space using a matching *CMYK* printing profile, converting such colors from four components to three and back is unnecessary and will result in an undesirable loss of fidelity in the black component. In such cases, PDF viewer applications may provide the ability for the user to specify a particular calibration to use for printing, proofing, or previewing. This calibration is then considered to be that of the native color space of the intended output device (typically **DeviceCMYK**), and colors expressed in a CIE-based source color space matching it can be treated as if they were specified directly in the device's native color space. Note that the conditions under which such implicit conversion is done cannot be specified in PDF itself, since nothing in PDF describes the calibration of the output device (although an output intent dictionary, if present, may suggest such a calibration; see Section 10.10.4, "Output Intents"). The conversion is completely hidden by the viewer application and plays no part in the interpretation of PDF color spaces.

When this type of implicit conversion is done, all of the semantics of the device color space should also apply, even though they do not apply to CIE-based spaces in general. In particular:

- The nonzero overprint mode (see Section 4.5.6, "Overprint Control") determines the interpretation of color component values in the space.
- If the space is used as the blending color space for a transparency group in the transparent imaging model (see Sections 7.2.3, "Blending Color Space"; 7.3, "Transparency Groups"; and 7.5.5, "Transparency Group XObjects"), components of the space, such as **Cyan**, can be selected in a **Separation** or **DeviceN** color space used within the group (see "Separation Color Spaces" on page 234 and "DeviceN Color Spaces" on page 238).
- Likewise, any uses of device color spaces for objects within such a transparency group have well-defined conversions to the group color space.

Note:** A source color space can be specified directly (for example, with an* ***ICCBased *color space) or indirectly using the default color space mechanism (for example,* ***DefaultCMYK****; see "Default Color Spaces" on page 227). The implicit conversion of a CIE-based color space to a device space should not depend on whether the CIE-based space is specified directly or indirectly.*

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 (PDF 1.1)* allows a PDF file to set priorities regarding which of these properties to preserve and which to sacrifice. For example, the PDF file might request that colors falling within the output device's gamut (the range of colors it can reproduce) be rendered 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.

Rendering intents are specified with the **ri** operator (see Section 4.3.3, "Graphics State Operators") and with the **Intent** entry in image dictionaries (Section 4.8.4, "Image Dictionaries"). The value is a name identifying the desired rendering intent. Table 4.19 lists the standard rendering intents recognized in the initial release of PDF viewer applications from Adobe Systems; Plate 5 illustrates their effects. These intents have been deliberately chosen to correspond closely to those 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. If the viewer application does not recognize the specified name, it uses the **RelativeColorimetric** intent by default.

See Section 7.6.4, "Rendering Parameters and Transparency," and in particular "Rendering Intent and Color Conversions" on page 533, for further discussion of the role of rendering intents in the transparent imaging model.

TABLE 4.19 Rendering intents

NAME	DESCRIPTION
AbsoluteColorimetric	Colors are represented solely with respect to the light source; no correction is made for the output 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 output 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 unmarked, 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.

4.5.5 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 Spaces

A **Pattern** color space *(PDF 1.2)* enables a PDF content stream to paint an area with a "color" defined as a *pattern*, which may be either a *tiling pattern* (type 1) or a *shading pattern* (type 2). Section 4.6, "Patterns," discusses patterns in detail.

Indexed Color Spaces

An **Indexed** color space allows a PDF content stream to select from a *color map* or *color table* of arbitrary colors in some other space, using small integers as indices. A PDF viewer application 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 defined by a four-element array, as follows:

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

The first element is the color space family name **Indexed**. The remaining elements are parameters that an **Indexed** color space requires; their meanings are discussed below. Setting the current stroking or nonstroking color space to an **Indexed** color space initializes the corresponding 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 PDF 1.3) a **Separation** or **DeviceN** space, but not a **Pattern** space 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 CIE-based *ABC* space such as a **CalRGB** or **Lab** space, the values are to be interpreted as *A*, *B*, and *C* components.

***Note:** Attempting to use a **Separation** or **DeviceN** color space as the base for an **Indexed** color space will generate an error in PDF 1.2.*

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 255, which is what would be required to index a table with 8-bit index values.

The color table is defined by the *lookup* parameter, which can be either a stream or (in PDF 1.2) a string. It provides the mapping between index values and the corresponding colors in the base color space.

The color table data must be $m \times (hival + 1)$ bytes long, where m is the number of color components in the base color space. Each byte is an unsigned integer in the range 0 to 255 that is scaled to the range of the corresponding color component in the base color space; that is, 0 corresponds to the minimum value in the range for that component, and 255 corresponds to the maximum.

***Note:** This is different from the interpretation of an **Indexed** color space's color table in PostScript. In PostScript, the component value is always scaled to the range 0.0 to 1.0, regardless of the range of color values in the base color space.*

The color components for each entry in the table appear consecutively in the string or stream. For example, if the base color space is **DeviceRGB** and the indexed color space contains two colors, the order of bytes in the string or stream is $R_0\ G_0\ B_0\ R_1\ G_1\ B_1$, where letters denote the color component and numeric subscripts denote the table entry.

Example 4.10 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 4.10

```
[ /Indexed
      /DeviceRGB
      255
      <000000 FF0000 00FF00 0000FF B57342 …>
]
```

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 and scaling the results to the range 0.0 to 1.0 yields a color with red, green, and blue components of 0.710, 0.451, and 0.259, respectively.

Although an **Indexed** color space is useful mainly for images, index values can also be used with the color selection operators **SC**, **SCN**, **sc**, and **scn**. For example,

```
123 sc
```

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 adjusted to the nearest value within that range.

Separation Color Spaces

Color output devices produce full color by combining *primary* or *process colorants* in varying amounts. On an additive color device such as a display, the primary colorants consist of red, green, and blue phosphors; on a subtractive device such as a printer, they typically consist of cyan, magenta, yellow, and sometimes black inks. In addition, some devices can apply special colorants, often called *spot colorants*, to produce effects that cannot be achieved with the standard process colorants alone. Examples include metallic and fluorescent colors and special textures.

When printing a page, most devices produce a single *composite* page on which all process colorants (and spot colorants, 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 *(PDF 1.2)* provides a means for specifying the use of additional colorants or for isolating the control of individual color components of a device color space for a subtractive device. 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 components 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 defined as follows:

```
[/Separation name alternateSpace tintTransform]
```

In other words, it 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 **DeviceGray** and **DeviceRGB**.) The **SCN** and **scn** operators respectively set the current stroking and nonstroking color in the graphics state to a tint value; the initial value in either case is 1.0. A sampled image with single-component samples can also be used as a source of tint values.

The *name* parameter in the color space array is a name 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.

The special colorant name **All** refers collectively to all colorants available on an output device, including those for the standard process colorants. 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 targets in the same place on every separation. Such marks would typically be painted as the last step in composing a page, to ensure that they are not overwritten by subsequent painting 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 *alternateSpace* and *tintTransform* parameters (discussed below), although valid values must still be provided.

At the moment the color space is set to a **Separation** space, the viewer application determines whether the device has an available colorant corresponding to the name of the requested space. If so, the application ignores the *alternateSpace* and *tintTransform* parameters; subsequent painting operations within the space will apply the designated colorant directly, according to the tint values supplied.

Note: *The preceding paragraph applies only to subtractive output devices such as printers and imagesetters. For an additive device such as a computer display, a* ***Separation*** *color space never applies a process colorant directly; it always reverts to the alternate color space as described below. This is because the model of applying process colorants independently does not work as intended on an additive device; for instance, painting tints of the* ***Red*** *component on a white background produces a result that varies from white to cyan.*

Note that this exception applies only to colorants for additive devices, not to the specific names ***Red****,* ***Green****, and* ***Blue****. In contrast, a printer might have a (subtractive) ink named, say,* ***Red****, which should work as a* ***Separation*** *color space just the same as any other supported colorant.*

If the colorant name associated with a **Separation** color space does not correspond to a colorant available on the device, the viewer application arranges instead for subsequent painting operations to be performed in an *alternate 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 colorants. This works as follows:

- The *alternateSpace* parameter must be an array or name object that identifies the alternate color space. This can be any device or CIE-based color space, but not another special color space (**Pattern**, **Indexed**, **Separation**, or **DeviceN**).
- The *tintTransform* parameter must be a function (see Section 3.9, "Functions"). During subsequent painting operations, a viewer application will call this function to transform a tint value into color component values in the alternate color space. The function is called with the tint value and must return the corresponding color component values. That is, the number of components and the interpretation of their values depend on the alternate color space.

Note: *Painting in the alternate color space may produce a good approximation of the intended color when only opaque objects are painted. However, it will not correctly represent the interactions between an object and its backdrop when the object is painted with transparency or when overprinting (see Section 4.5.6, "Overprint Control") is enabled.*

Example 4.11 illustrates the specification of a **Separation** color space (object 5) 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 alternate color space; the tint transformation function provided (object 12) maps tint values linearly into shades of a *CMYK* color value approximating the "logo green" color.

Example 4.11

```
5 0 obj                                      % Color space
   [ /Separation
         /LogoGreen
         /DeviceCMYK
         12 0 R
   ]
endobj
```

```
12  0  obj                                          % Tint transformation function
   <<  /FunctionType  4
        /Domain  [0.0  1.0]
        /Range  [0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0]
        /Length  62
   >>
stream
   {  dup  0.84  mul
      exch  0.00  exch  dup  0.44  mul
      exch  0.21  mul
   }
endstream
endobj
```

See Section 7.6.2, "Spot Colors and Transparency," for further discussion of the role of **Separation** color spaces in the transparent imaging model.

DeviceN Color Spaces

DeviceN color spaces *(PDF 1.3)* 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 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 nonlinear function of the image data in a manner that emphasizes the shadows. Plate 6 shows an example using black and magenta color components. In Plate 7, a single-component grayscale image is used to generate a *quadtone* result using four colorants: black and three PANTONE spot colors. See Example 4.17 on page 245 for the code used to generate this image.

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 4.5.6, "Overprint Control"), painting in this color space will leave the black component unchanged.

A **DeviceN** color space is specified as follows:

```
[/DeviceN names alternateSpace tintTransform]
```

or

```
[/DeviceN names alternateSpace tintTransform attributes]
```

It is a four- or five-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.

Color values in the **DeviceN** color space are tint components in the range 0.0 to 1.0. The value 0.0 represents the minimum amount of colorant; 1.0 represents the maximum. The **SCN** and **scn** operators respectively set the current stroking and nonstroking color in the graphics state to a set of tint values; the initial value is 1.0 for each tint. A sampled image can also be treated as a source of tint values.

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 number of components is subject to an implementation limit; see Appendix C.
- The *names* parameter in the color space array is an array of name objects specifying the individual colorants. The colorant names must all be different from one another, except that **None** may be repeated. (The special colorant name **All** is not allowed.) The length of the array determines the number of components, and hence the number of operands required by the **SCN** and **scn** operators when this space is the current color space. Operand values supplied to **SCN** or **scn** are interpreted as color component values in the order in which the colors are given in the *names* array.

- At the moment the color space is set to a **DeviceN** space, the viewer application will select the requested set of colorants only if all of them are available on the device; otherwise, it will select the alternate color space designated by the *alternateSpace* parameter.
- The tint transformation function is called with *n* tint values 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 alternate color space.

In a **DeviceN** color space, one or more of the colorant names in the *names* array may be the name **None**. This indicates that the corresponding color component is never painted on the page, as in a **Separation** color space for the **None** colorant. (However, see implementation note 40 in Appendix H.) When a **DeviceN** color space is painting the named device colorants directly, color components corresponding to **None** colorants are discarded. However, when the **DeviceN** color space reverts to its alternate color space, those components are passed to the tint transformation function, which may use them in any desired manner.

***Note:** A **DeviceN** color space whose component colorant names are all **None** always discards its output, just the same as a **Separation** color space for **None**; it never reverts to the alternate color space. Reversion occurs only if at least one color component (other than **None**) is specified and is not available on the device.*

The optional *attributes* parameter is a dictionary containing additional information about the color space. At the time of publication, only one entry is defined in this dictionary, as shown in Table 4.20.

Example 4.12 shows a **DeviceN** color space consisting of three color components named **Orange**, **Green**, and **None**. In this example, the **DeviceN** color space, object 30, has an attributes dictionary whose **Colorants** entry is an indirect reference to object 45 (which might also be referenced by attributes dictionaries of other **DeviceN** color spaces). *tintTransform1*, whose definition is not shown, maps three color components (tints of the colorants **Orange**, **Green**, and **None**) to four color components in the alternate color space, **DeviceCMYK**. *tintTransform2* maps a single color component (an orange tint) to four components in **DeviceCMYK**. Likewise, *tintTransform3* maps a green tint to **DeviceCMYK**, and *tintTransform4* maps a tint of PANTONE 131 to **DeviceCMYK**.

TABLE 4.20 Entry in a DeviceN color space attributes dictionary

KEY	TYPE	VALUE
Colorants	dictionary	*(Optional)* A dictionary describing the individual colorants used in the **DeviceN** color space. For each entry in this dictionary, the key is a colorant name and the value is an array defining a **Separation** color space for that colorant (see “Separation Color Spaces” on page 234). The key must match the colorant name given in that color space. The dictionary need not list all colorants used in the **DeviceN** color space and may list additional colorants. This dictionary has no effect on the operation of the **DeviceN** color space itself or the appearance that it produces. However, it provides information about the individual colorants that may be useful to some applications. In particular, the alternate color space and tint transformation function of a **Separation** color space describe the appearance of that colorant alone, whereas those of a **DeviceN** color space describe only the appearance of its colorants in combination.

Example 4.12

```
30 0 obj                                        % Color space
  [ /DeviceN
        [/Orange /Green /None]
        /DeviceCMYK
        tintTransform1
        << /Colorants 45 0 R >>
  ]
endobj

45 0 obj                                        % Colorants dictionary
  << /Orange [ /Separation
                  /Orange
                  /DeviceCMYK
                  tintTransform2
              ]
     /Green [ /Separation
                 /Green
                 /DeviceCMYK
                 tintTransform3
             ]
     /PANTONE#20131 [ /Separation
                         /PANTONE#20131
```

```
                                        /DeviceCMYK
                                        tintTransform4
                                    ]
    >>
endobj
```

See Section 7.6.2, “Spot Colors and Transparency,” for further discussion of the role of **DeviceN** color spaces in the transparent imaging model.

Multitone Examples

The following examples illustrate various interesting and useful special cases of the use of **Indexed** and **DeviceN** color spaces in combination to produce multitone colors.

Examples 4.13 and 4.14 illustrate the use of **DeviceN** to create duotone color spaces. In Example 4.13, an **Indexed** color space maps index values in the range 0 to 255 to a duotone **DeviceN** space in cyan and black. In effect, the index values are treated as if they were tints of the duotone space, which are then mapped into tints of the two underlying colorants. Only the beginning of the lookup table string for the **Indexed** color space is shown; the full table would contain 256 two-byte entries, each specifying a tint value for cyan and black, for a total of 512 bytes. If the alternate color space of the **DeviceN** space is selected, the tint transformation function (object 15 in the example) maps the two tint components for cyan and black to the four components for a **DeviceCMYK** color space by supplying zero values for the other two components. Example 4.14 shows the definition of another duotone color space, this time using black and gold colorants (where gold is a spot colorant) and using a **CalRGB** space as the alternate color space. This could be defined in the same way as in the preceding example, with a tint transformation function that converts from the two tint components to colors in the alternate **CalRGB** color space.

Example 4.13

```
10 0 obj                                                % Color space
   [ /Indexed
         [ /DeviceN
               [/Cyan /Black]
               /DeviceCMYK
               15 0 R
         ]
         255
         <6605 6806 6907 6B09 6C0A …>
   ]
endobj

15 0 obj                                                % Tint transformation function
   << /FunctionType 4
       /Domain [0.0 1.0 0.0 1.0]
       /Range [0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0]
       /Length 16
   >>
stream
   {0 0 3 −1 roll}
endstream
endobj
```

Example 4.14

```
30 0 obj                                                % Color space
   [ /Indexed
         [ /DeviceN
               [/Black /Gold]
               [ /CalRGB
                     << /WhitePoint [1.0 1.0 1.0]
                         /Gamma [2.2 2.2 2.2]
                     >>
               ]
               35 0 R                                   % Tint transformation function
         ]
         255
         …Lookup table…
   ]
endobj
```

Given a formula for converting any combination of black and gold tints to calibrated *RGB*, a 2-in, 3-out type 4 (PostScript calculator) function could be used for the tint transformation. Alternatively, a type 0 (sampled) function could be used, but this would require a large number of sample points to represent the function accurately; for example, sampling each input variable for 256 tint values between 0.0 and 1.0 would require $256^2 = 65{,}536$ samples. But since the **DeviceN** color space is being used as the base of an **Indexed** color space, there are actually only 256 possible combinations of black and gold tint values. A more compact way to represent this information is to put the alternate color values directly into the lookup table alongside the **DeviceN** color values, as in Example 4.15.

Example 4.15

```
10 0 obj                                          % Color space
   [ /Indexed
         [ /DeviceN
               [/Black /Gold /None /None /None]
               [   /CalRGB
                     << /WhitePoint [1.0 1.0 1.0]
                          /Gamma [2.2 2.2 2.2]
                     >>
               ]
               20 0 R                             % Tint transformation function
         ]
         255
         … Lookup table …
   ]
endobj
```

In this example, each entry in the lookup table has *five* components: two for the black and gold colorants and three more (specified as **None**) for the equivalent **CalRGB** color components. If the black and gold colorants are available on the output device, the **None** components will be ignored; if black and gold are not available, the tint transformation function will be used to convert a five-component color into a three-component equivalent in the alternate **CalRGB** color space. But since, by construction, the third, fourth, and fifth components *are* the **CalRGB** components, the tint transformation function can merely discard the first two components and return the last three. This can be easily expressed with a type 4 (PostScript calculator) function, as shown in Example 4.16.

Example 4.16

```
20  0  obj                                                       % Tint transformation function
   <<  /FunctionType  4
       /Domain  [0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0]
       /Range  [0.0  1.0   0.0  1.0   0.0  1.0]
       /Length  27
   >>
stream
   {5  3  roll  pop  pop}
endstream
endobj
```

Finally, Example 4.17 uses an extension of the techniques described above to produce the quadtone (four-component) image shown in Plate 7.

Example 4.17

```
5  0  obj                                 % Image XObject
   <<  /Type  /XObject
       /Subtype  /Image
       /Width  288
       /Height  288
       /ColorSpace  10 0 R
       /BitsPerComponent  8
       /Length  105278
       /Filter  /ASCII85Decode
   >>
stream
… Data for grayscale image …
endstream
endobj

10  0  obj                                % Indexed color space for image
   [  /Indexed
         15 0 R                           % Base color space
         255                              % Table has 256 entries
         30 0 R                           % Lookup table
   ]
endobj
```

```
15  0  obj                              % Base color space (DeviceN) for Indexed space
   [  /DeviceN
         [   /Black                     % Four colorants (black plus three spot colors)
             /PANTONE#20216#20CVC
             /PANTONE#20409#20CVC
             /PANTONE#202985#20CVC
             /None                      % Three components for alternate space
             /None
             /None
         ]
         16 0 R                         % Alternate color space
         20 0 R                         % Tint transformation function
   ]
endobj

16  0  obj                              % Alternate color space for DeviceN space
   [  /CalRGB
         <<  /WhitePoint  [1.0  1.0  1.0]  >>
   ]
endobj

20  0  obj                              % Tint transformation function for DeviceN space
   <<   /FunctionType  4
        /Domain  [0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0   0.0  1.0]
        /Range  [0.0  1.0   0.0  1.0   0.0  1.0]
        /Length  44
   >>
stream
   {  7  3  roll                        % Just discard first four values
      pop  pop  pop  pop
   }
endstream
endobj

30  0  obj                              % Lookup table for Indexed color space
   <<   /Length  1975
        /Filter  [/ASCII85Decode  /FlateDecode]
   >>
stream
8;T1BB2"M7*!"psYBt1k\gY1T<D&tO]r*F7Hga*
… Additional data (seven components for each table entry) …
endstream
endobj
```

As in the preceding examples, an **Indexed** color space based on a **DeviceN** space is used to paint the grayscale image shown on the left in the plate with four colorants: black and three PANTONE spot colors. The alternate color space is a simple calibrated *RGB*. Thus the **DeviceN** color space has seven components: the four desired colorants plus the three components of the alternate space. The example shows the image XObject (see Section 4.8.4, "Image Dictionaries") representing the quadtone image, followed by the color space used to interpret the image data. (See implementation note 41 in Appendix H.)

4.5.6 Overprint Control

The graphics state contains an *overprint parameter*, controlled by the **OP** and **op** entries in a graphics state parameter dictionary. 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 color space, they are implied by the process color model of the output device (see Chapter 6). 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 parameter 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 at any position on the page is whatever was painted there last; this is consistent with the normal painting behavior of the opaque imaging model.

If the overprint parameter 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 PDF language.

***Note:** Not all devices support overprinting. Furthermore, many PostScript printers support it only when separations are being produced, and not for composite output. If overprinting is not supported, the value of the overprint parameter is ignored.*

An additional graphics state parameter, the *overprint mode (PDF 1.3)*, affects the interpretation of a tint value of 0.0 for a color component in a **DeviceCMYK** color space when overprinting is enabled. This parameter is controlled by the **OPM** entry in a graphics state parameter dictionary; it has an effect only when the overprint parameter is **true**, as described above.

When colors are specified in a **DeviceCMYK** color space and the native color space of the output device is also **DeviceCMYK**, each of the source color components controls the corresponding device colorant directly. Ordinarily, each source color component value replaces the value previously painted for the corresponding device colorant, no matter what the new value is; this is the default behavior, specified by overprint mode 0.

When the overprint mode is 1 (also called *nonzero overprint mode*), a tint value of 0.0 for a source color component leaves the corresponding component of the previously painted color unchanged. The effect is equivalent to painting in a **DeviceN** color space that includes only those components whose values are nonzero. For example, if the overprint parameter is **true** and the overprint mode is 1, the operation

```
0.2  0.3  0.0  1.0  k
```

is equivalent to

```
0.2  0.3  1.0  scn
```

in the color space shown in Example 4.18.

Example 4.18

```
10 0 obj                                          % Color space
   [ /DeviceN
         [/Cyan /Magenta /Black]
         /DeviceCMYK
         15 0 R
   ]
endobj

15 0 obj                                          % Tint transformation function
   << /FunctionType 4
        /Domain [0.0 1.0  0.0 1.0  0.0 1.0]
        /Range [0.0 1.0  0.0 1.0  0.0 1.0  0.0 1.0]
        /Length 13
   >>
stream
   {0 exch}
endstream
endobj
```

Nonzero overprint mode applies only to painting operations that use the current color in the graphics state when the current color space is **DeviceCMYK** (or is implicitly converted to **DeviceCMYK**; see "Implicit Conversion of CIE-Based Color Spaces" on page 228). It does not apply to the painting of images or to any colors that are the result of a computation, such as those in a shading pattern or conversions from some other color space. It also does not apply if the device's native color space is not **DeviceCMYK**; in that case, source colors must be converted to the device's native color space, and all components participate in the conversion, whatever their values. (This is shown explicitly in the alternate color space and tint transformation function of the **DeviceN** color space in Example 4.18.)

See Section 7.6.3, "Overprinting and Transparency," for further discussion of the role of overprinting in the transparent imaging model.

4.5.7 Color Operators

Table 4.21 lists the PDF operators that control color spaces and color values. (Also color-related is the graphics state operator **ri**, listed in Table 4.7 on page 189 and discussed under "Rendering Intents" on page 230.) Color operators may appear at the page description level or inside text objects (see Figure 4.1 on page 167).

TABLE 4.21 Color operators

OPERANDS	OPERATOR	DESCRIPTION
name	**CS**	*(PDF 1.1)* Set the current color space to use for stroking operations. The operand *name* must be a name object. If the color space is one that can be specified by a name and no additional parameters (**DeviceGray**, **DeviceRGB**, **DeviceCMYK**, and certain cases of **Pattern**), the name may be specified directly. Otherwise, it must be a name defined in the **ColorSpace** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"); the associated value is an array describing the color space (see Section 4.5.2, "Color Space Families"). ***Note:*** *The names **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, and **Pattern** always identify the corresponding color spaces directly; they never refer to resources in the **ColorSpace** subdictionary.* The **CS** operator also sets the current stroking color to its initial value, which depends on the color space: • In a **DeviceGray**, **DeviceRGB**, **CalGray**, or **CalRGB** color space, the initial color has all components equal to 0.0. • In a **DeviceCMYK** color space, the initial color is [0.0 0.0 0.0 1.0]. • In a **Lab** or **ICCBased** color space, the initial color has all components equal to 0.0 unless that falls outside the intervals specified by the space's **Range** entry, in which case the nearest valid value is substituted. • In an **Indexed** color space, the initial color value is 0. • In a **Separation** or **DeviceN** color space, the initial tint value is 1.0 for all colorants. • In a **Pattern** color space, the initial color is a pattern object that causes nothing to be painted.
name	**cs**	*(PDF 1.1)* Same as **CS**, but for nonstroking operations.
$c_1 \ldots c_n$	**SC**	*(PDF 1.1)* Set the color to use for stroking operations in a device, CIE-based (other than **ICCBased**), or **Indexed** color space. The number of operands required and their interpretation depends on the current stroking color space: • For **DeviceGray**, **CalGray**, and **Indexed** color spaces, one operand is required ($n = 1$). • For **DeviceRGB**, **CalRGB**, and **Lab** color spaces, three operands are required ($n = 3$). • For **DeviceCMYK**, four operands are required ($n = 4$).

OPERANDS	OPERATOR	DESCRIPTION
$c_1 \ldots c_n$ $c_1 \ldots c_n$ *name*	**SCN** **SCN**	*(PDF 1.2)* Same as **SC**, but also supports **Pattern**, **Separation**, **DeviceN**, and **ICCBased** color spaces. If the current stroking color space is a **Separation**, **DeviceN**, or **ICCBased** color space, the operands $c_1 \ldots c_n$ are numbers. The number of operands and their interpretation depends on the color space. If the current stroking color space is a **Pattern** color space, *name* is the name of an entry in the **Pattern** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"). For an uncolored tiling pattern (**PatternType** = 1 and **PaintType** = 2), $c_1 \ldots c_n$ are component values specifying a color in the pattern's underlying color space. For other types of pattern, these operands must not be specified.
$c_1 \ldots c_n$	**sc**	*(PDF 1.1)* Same as **SC**, but for nonstroking operations.
$c_1 \ldots c_n$ $c_1 \ldots c_n$ *name*	**scn** **scn**	*(PDF 1.2)* Same as **SCN**, but for nonstroking operations.
gray	**G**	Set the stroking color space to **DeviceGray** (or the **DefaultGray** color space; see "Default Color Spaces" on page 227) and set the gray level to use for stroking operations. *gray* is a number between 0.0 (black) and 1.0 (white).
gray	**g**	Same as **G**, but for nonstroking operations.
r g b	**RG**	Set the stroking color space to **DeviceRGB** (or the **DefaultRGB** color space; see "Default Color Spaces" on page 227) and set the color to use for stroking operations. Each operand must be a number between 0.0 (minimum intensity) and 1.0 (maximum intensity).
r g b	**rg**	Same as **RG**, but for nonstroking operations.
c m y k	**K**	Set the stroking color space to **DeviceCMYK** (or the **DefaultCMYK** color space; see "Default Color Spaces" on page 227) and set the color to use for stroking operations. Each operand must be a number between 0.0 (zero concentration) and 1.0 (maximum concentration). The behavior of this operator is affected by the overprint mode (see Section 4.5.6, "Overprint Control").
c m y k	**k**	Same as **K**, but for nonstroking operations.

In certain circumstances, invoking operators that specify colors or other color-related parameters in the graphics state is not allowed. This restriction occurs when defining graphical figures whose colors are to be specified separately each time they are used. Specifically, the restriction applies:

- In any glyph description that uses the **d1** operator (see Section 5.5.4, "Type 3 Fonts")
- In the content stream of an uncolored tiling pattern (see "Uncolored Tiling Patterns" on page 261)

In these circumstances, the following will cause an error:

- Invoking any of the following operators:

CS	**scn**	**K**
cs	**G**	**k**
SC	**g**	**ri**
SCN	**RG**	**sh**
sc	**rg**	

- Invoking the **gs** operator with any of the following entries in the graphics state parameter dictionary:

TR	**BG**	**UCR**
TR2	**BG2**	**UCR2**
HT		

- Painting an image. However, painting an *image mask* (see "Stencil Masking" on page 313) is permitted, because it does not specify colors, but rather designates places where the current color is to be painted.

4.6 Patterns

When operators such as **S** (stroke), **f** (fill), and **Tj** (show text) paint an area of the page with the current color, they ordinarily apply a single color that covers the area uniformly. However, it is also possible to apply "paint" that consists of a repeating graphical 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; for example, they can be used to create various graphical textures, such as weaves, brick walls, sunbursts, and similar geometrical and chromatic effects. (See implementation note 42 in Appendix H.)

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. The graphics objects to use for tiling are described by a content stream.
- *Shading patterns* define a *gradient fill* that produces a smooth transition between colors across the area. The color to use is specified as a function of position using any of a variety of methods.

Note: *The ability to paint with patterns is a feature of PDF 1.2 (tiling patterns) and PDF 1.3 (shading patterns). With some effort, it is possible to achieve a limited form of tiling patterns in PDF 1.1 by defining them as character glyphs in a special font and painting them repeatedly with the* ***Tj*** *operator. Another technique, defining patterns as halftone screens, is not recommended, because the effects produced are device-dependent.*

Patterns are specified in a special family of color spaces named **Pattern**, whose "color values" are *pattern objects* instead of the numeric component values used with other spaces. A pattern object may be a dictionary or a stream, depending on the type of pattern; the term *pattern dictionary* is used generically throughout this section to refer to either a dictionary object or the dictionary portion of a stream object. (Those pattern objects that are streams are specifically identified as such in the descriptions of particular pattern types; unless otherwise stated, they are understood to be simple dictionaries instead.) This section describes **Pattern** color spaces and the specification of color values within them; see Section 4.5, "Color Spaces," for information about color spaces and color values in general, and Section 7.5.6, "Patterns and Transparency," for further discussion of the treatment of patterns in the transparent imaging model.

4.6.1 General Properties of 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 for a tiling pattern or type 2 for a shading pattern. The remaining contents of the dictionary depend on the pattern type, and are detailed below in the sections on individual pattern types.

All patterns are treated as colors; a **Pattern** color space is established with the **CS** or **cs** operator just like other color spaces, and a particular pattern is installed as the current color with the **SCN** or **scn** operator (see Table 4.21 on page 250).

A pattern's appearance is described with respect to its own internal coordinate system. Every pattern has a *pattern matrix*, a transformation matrix that maps the pattern's internal coordinate system to the default coordinate system of the pattern's *parent content stream* (the content stream in which the pattern is defined as a resource). The concatenation of the pattern matrix with that of the parent content stream establishes the *pattern coordinate space*, within which all graphics objects in the pattern are interpreted.

For example, if a pattern is used on a page, the pattern will appear in the **Pattern** subdictionary of that page's resource dictionary, and the pattern matrix maps pattern space to the default (initial) coordinate space of the page. Changes to the page's transformation matrix that occur within the page's content stream, such as rotation and scaling, have no effect on the pattern; it maintains its original relationship to the page no matter where on the page it is used. Similarly, if a pattern is used within a form XObject (see Section 4.9, "Form XObjects"), the pattern matrix maps pattern space to the form's default user space (that is, the form coordinate space at the time the form is painted with the **Do** operator). Finally, a pattern may used within another pattern; the inner pattern's matrix defines its relationship to the pattern space of the outer pattern.

Note: *PostScript allows a pattern to be defined in one context but used in another. For example, a pattern might be defined on a page (that is, its pattern matrix maps the pattern coordinate space to the user space of the page) but be used in a form on that page, so that its relationship to the page is independent of each individual placement of the form. PDF does not support this feature; in PDF, all patterns are local to the context in which they are defined.*

4.6.2 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. When performing painting operations such as **S** (stroke) or **f** (fill), the viewer application paints the cell on the current page as many times as necessary to fill an area. The order in which individual tiles (instances of the cell) are painted is unspecified and unpredictable; it is inadvisable for the figures on adjacent tiles to overlap.

The appearance of the pattern cell is defined by a content stream containing the painting operators needed to paint one instance of the cell. Besides the usual entries common to all streams (see Table 3.4 on page 38), this stream's dictionary has the additional entries listed in Table 4.22.

TABLE 4.22 Additional entries specific to a type 1 pattern dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Pattern** for a pattern dictionary.
PatternType	integer	*(Required)* A code identifying the type of pattern that this dictionary describes; must be 1 for a tiling pattern.
PaintType	integer	*(Required)* A code that determines how the color of the pattern cell is to be specified: 1 *Colored tiling pattern*. The pattern's content stream itself specifies the colors used to paint the pattern cell. When the content stream begins execution, the current color is the one that was initially in effect in the pattern's parent content stream. (This is similar to the definition of the pattern matrix; see Section 4.6.1, "General Properties of Patterns.") 2 *Uncolored tiling pattern*. The pattern's content stream does not specify any color information. Instead, the entire pattern cell is painted with a separately specified color each time the pattern is used. Essentially, the content stream describes a *stencil* through which the current color is to be poured. The content stream must not invoke operators that specify colors or other color-related parameters in the graphics state; otherwise, an error will occur (see Section 4.5.7, "Color Operators"). The content stream may paint an image mask, however, since it does not specify any color information (see "Stencil Masking" on page 313).

KEY	TYPE	VALUE
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, the viewer application 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 pattern is painted. This achieves the spacing requested by **XStep** and **YStep** *on average*, but not necessarily 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.
BBox	rectangle	*(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.
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.
Resources	dictionary	*(Required)* A resource dictionary containing all of the named resources required by the pattern's content stream (see Section 3.7.2, "Resource Dictionaries").
Matrix	array	*(Optional)* An array of six numbers specifying the pattern matrix (see Section 4.6.1, "General Properties of Patterns"). Default value: the identity matrix `[1 0 0 1 0 0]`.

The pattern dictionary's **BBox**, **XStep**, and **YStep** values are interpreted in the pattern coordinate system, and the graphics objects in the pattern's content stream are defined with respect to 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 **Matrix** entry in the pattern dictionary.

The first step in painting with a tiling pattern is to establish the pattern as the current color in the graphics state. Subsequent painting operations will tile the painted areas with the pattern cell described by the pattern's content stream. Whenever it needs to obtain the pattern cell, the viewer application does the following:

1. Saves the current graphics state (as if by invoking the **q** operator)
2. Installs the graphics state that was in effect at the beginning of the pattern's parent content stream, with the current transformation matrix altered by the pattern matrix as described in Section 4.6.1, "General Properties of Patterns"
3. Paints the graphics objects specified in the pattern's content stream
4. Restores the saved graphics state (as if by invoking the **Q** operator)

Note: *The pattern's content stream should not set any of the device-dependent parameters in the graphics state (see Table 4.3 on page 182). Doing so may result in incorrect output.*

Colored Tiling Patterns

A *colored tiling pattern* is one whose color is self-contained. In the course of painting the pattern cell, the pattern's content stream 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. This type of pattern is identified by a pattern type of 1 and a paint type of 1 in the pattern dictionary.

When the current color space is a **Pattern** space, a colored tiling pattern can be selected as the current color by supplying its name as the single operand to the **SCN** or **scn** operator. This name must be the key of an entry in the **Pattern** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"), whose value is the stream object representing the pattern. Since the pattern defines its own color information, no additional operands representing color components are specified to **SCN** or **scn**. For example, if P1 is the name of a pattern resource in the current resource dictionary, the following code establishes it as the current nonstroking color:

```
/Pattern cs
/P1 scn
```

Subsequent executions of nonstroking painting operators, such as **f** (fill), **Tj** (show text), or **Do** (paint external object) with an image mask, will use the designated pattern to tile the areas to be painted.

Example 4.19 defines a page (object 5) that paints three circles and a triangle using a colored tiling pattern (object 15) over a yellow background. The pattern consists of the symbols for the four suits of playing cards (spades, hearts, diamonds, and clubs), which are character glyphs taken from the ZapfDingbats font (see Section D.4, "ZapfDingbats Set and Encoding"); the pattern's content stream specifies the color of each glyph. Plate 8 shows the results.

Example 4.19

```
5 0 obj                                        % Page object
   << /Type /Page
        /Parent 2 0 R
        /Resources 10 0 R
        /Contents 30 0 R
        /CropBox [0 0 225 225]
   >>
endobj

10 0 obj                                       % Resource dictionary for page
   << /Pattern << /P1 15 0 R >>
   >>
endobj

15 0 obj                                       % Pattern definition
   << /Type /Pattern
        /PatternType 1                         % Tiling pattern
        /PaintType 1                           % Colored
        /TilingType 2
        /BBox [0 0 100 100]
        /XStep 100
        /YStep 100
```

```
            /Resources  16 0 R
            /Matrix  [0.4  0.0  0.0  0.4  0.0  0.0]
            /Length  183
    >>
stream
    BT                                               % Begin text object
        /F1  1  Tf                                   % Set text font and size
        64  0  0  64  7.1771  2.4414  Tm             % Set text matrix
        0  Tc                                        % Set character spacing
        0  Tw                                        % Set word spacing
        1.0  0.0  0.0  rg                            % Set nonstroking color to red
        (\001)  Tj                                   % Show spade glyph
        0.7478  –0.007  TD                           % Move text position
        0.0  1.0  0.0  rg                            % Set nonstroking color to green
        (\002)  Tj                                   % Show heart glyph
        –0.7323  0.7813  TD                          % Move text position
        0.0  0.0  1.0  rg                            % Set nonstroking color to blue
        (\003)  Tj                                   % Show diamond glyph
        0.6913  0.007  TD                            % Move text position
        0.0  0.0  0.0  rg                            % Set nonstroking color to black
        (\004)  Tj                                   % Show club glyph
    ET                                               % End text object
endstream
endobj

16  0  obj                                           % Resource dictionary for pattern
    <<  /Font  <<  /F1  20 0 R  >>
    >>
endobj

20  0  obj                                           % Font for pattern
    <<  /Type  /Font
            /Subtype  /Type1
            /Encoding  21 0 R
            /BaseFont  /ZapfDingbats
    >>
endobj

21  0  obj                                           % Font encoding
    <<  /Type  /Encoding
            /Differences  [1  /a109  /a110  /a111  /a112]
    >>
endobj
```

```
30 0 obj                                              % Contents of page
   << /Length 1252 >>
stream
   0.0 G                                              % Set stroking color to black
   1.0 1.0 0.0 rg                                     % Set nonstroking color to yellow
   25 175 175 –150 re                                 % Construct rectangular path
   f                                                  % Fill path
   /Pattern cs                                        % Set pattern color space
   /P1 scn                                            % Set pattern as nonstroking color
   99.92 49.92 m                                      % Start new path
   99.92 77.52 77.52 99.92 49.92 99.92 c              % Construct lower-left circle
   22.32 99.92 –0.08 77.52 –0.08 49.92 c
   –0.08 22.32 22.32 –0.08 49.92 –0.08 c
   77.52 –0.08 99.92 22.32 99.92 49.92 c
   B                                                  % Fill and stroke path
   224.96 49.92 m                                     % Start new path
   224.96 77.52 202.56 99.92 174.96 99.92 c           % Construct lower-right circle
   147.36 99.92 124.96 77.52 124.96 49.92 c
   124.96 22.32 147.36 -0.08 174.96 –0.08 c
   202.56 –0.08 224.96 22.32 224.96 49.92 c
   B                                                  % Fill and stroke path
   87.56 201.70 m                                     % Start new path
   63.66 187.90 55.46 157.32 69.26 133.40 c           % Construct upper circle
   83.06 109.50 113.66 101.30 137.56 115.10 c
   161.46 128.90 169.66 159.50 155.86 183.40 c
   142.06 207.30 111.46 215.50 87.56 201.70 c
   B                                                  % Fill and stroke path
   50 50 m                                            % Start new path
   175 50 l                                           % Construct triangular path
   112.5 158.253 l
   b                                                  % Close, fill, and stroke path
endstream
endobj
```

Several features of Example 4.19 are noteworthy:

- The three circles and the triangle are painted with the same pattern. The pattern cells align, even though the circles and triangle are not aligned with respect to the pattern cell. For example, the position of the blue diamonds varies relative to the three circles.

- The pattern cell does not completely cover the tile: it leaves the spaces between the glyphs unpainted. When the tiling pattern is used as a color, the existing background (the yellow rectangle) shows through these unpainted areas.

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 type of pattern is identified by a pattern type of 1 and a paint type of 2 in the pattern dictionary. The pattern's content stream does not explicitly specify any colors; it can paint an image mask (see "Stencil Masking" on page 313), but no other kind of image. This provides a way to tile different regions of the page with pattern cells having the same shape but different colors.

A **Pattern** color space representing an uncolored tiling pattern requires a parameter: an object identifying the *underlying color space* in which the actual color of the pattern is to be specified. The underlying color space is given as the second element of the array that defines the **Pattern** color space. For example, the array

```
[/Pattern /DeviceRGB]
```

defines a **Pattern** color space with **DeviceRGB** as its underlying color space.

Note: *The underlying color space cannot be another* ***Pattern*** *color space.*

Operands supplied to the **SCN** or **scn** operator in such a color space must include a color value in the underlying color space, specified by one or more numeric color components, as well as the name of a pattern object representing an uncolored tiling pattern. For example, if the current resource dictionary (see Section 3.7.2, "Resource Dictionaries") defines Cs3 as the name of a **ColorSpace** resource whose value is the **Pattern** color space shown above, and P2 as a **Pattern** resource denoting an uncolored tiling pattern, then the code

```
/Cs3 cs
0.30 0.75 0.21 /P2 scn
```

establishes Cs3 as the current nonstroking color space and P2 as the current nonstroking color, to be painted in the color represented by the specified components in the **DeviceRGB** color space. Subsequent executions of nonstroking painting operators, such as **f** (fill), **Tj** (show text), and **Do** (paint external object) with an image mask, will use the designated pattern and color to tile the areas to be painted. The same pattern can be used repeatedly with a different color each time.

Example 4.20 is similar to Example 4.19 on page 258, except that it uses an uncolored tiling pattern to paint the three circles and the triangle, each in a different color (see Plate 9). To do so, it supplies four operands each time it invokes the **scn** operator: three numbers denoting the components of the desired color in the underlying **DeviceRGB** color space, along with the name of the pattern.

Example 4.20

```
5 0 obj                                         % Page object
   << /Type /Page
      /Parent 2 0 R
      /Resources 10 0 R
      /Contents 30 0 R
      /CropBox [0 0 225 225]
   >>
endobj

10 0 obj                                        % Resource dictionary for page
   << /ColorSpace << /Cs12 12 0 R >>
      /Pattern << /P1 15 0 R >>
   >>
endobj

12 0 obj                                        % Color space
   [/Pattern /DeviceRGB]
endobj

15 0 obj                                        % Pattern definition
   << /Type /Pattern
      /PatternType 1                            % Tiling pattern
      /PaintType 2                              % Uncolored
      /TilingType 2
      /BBox [0 0 100 100]
      /XStep 100
      /YStep 100
      /Resources 16 0 R
      /Matrix [0.4 0.0 0.0 0.4 0.0 0.0]
      /Length 127
   >>
```

```
stream
    BT                                              % Begin text object
        /F1  1  Tf                                  % Set text font and size
        64  0  0  64  7.1771  2.4414  Tm            % Set text matrix
        0  Tc                                       % Set character spacing
        0  Tw                                       % Set word spacing
        (\001)  Tj                                  % Show spade glyph
        0.7478  –0.007  TD                          % Move text position
        (\002)  Tj                                  % Show heart glyph
        –0.7323  0.7813  TD                         % Move text position
        (\003)  Tj                                  % Show diamond glyph
        0.6913  0.007  TD                           % Move text position
        (\004)  Tj                                  % Show club glyph
    ET                                              % End text object
endstream
endobj

16  0  obj                                          % Resource dictionary for pattern
    <<  /Font  <<  /F1  20 0 R  >>
    >>
endobj

20  0  obj                                          % Font for pattern
    <<  /Type  /Font
         /Subtype  /Type1
         /Encoding  21 0 R
         /BaseFont  /ZapfDingbats
    >>
endobj

21  0  obj                                          % Font encoding
    <<  /Type  /Encoding
         /Differences  [1  /a109  /a110  /a111  /a112]
    >>
endobj

30  0  obj                                          % Contents of page
    <<  /Length  1316  >>
stream
    0.0  G                                          % Set stroking color to black
    1.0  1.0  0.0  rg                               % Set nonstroking color to yellow
    25  175  175  –150  re                          % Construct rectangular path
    f                                               % Fill path
```

```
        /Cs12  cs                                        % Set pattern color space
        0.77  0.20  0.00  /P1  scn                       % Set nonstroking color and pattern
        99.92  49.92  m                                  % Start new path
        99.92  77.52  77.52  99.92  49.92  99.92  c      % Construct lower-left circle
        22.32  99.92  –0.08  77.52  –0.08  49.92  c
        –0.08  22.32  22.32  –0.08  49.92  –0.08  c
        77.52  –0.08  99.92  22.32  99.92  49.92  c
        B                                                % Fill and stroke path
        0.2  0.8  0.4  /P1  scn                          % Change nonstroking color
        224.96  49.92  m                                 % Start new path
        224.96  77.52  202.56  99.92  174.96  99.92  c   % Construct lower-right circle
        147.36  99.92  124.96  77.52  124.96  49.92  c
        124.96  22.32  147.36  -0.08  174.96  –0.08  c
        202.56  –0.08  224.96  22.32  224.96  49.92  c
        B                                                % Fill and stroke path
        0.3  0.7  1.0  /P1  scn                          % Change nonstroking color
        87.56  201.70  m                                 % Start new path
        63.66  187.90  55.46  157.30  69.26  133.40  c   % Construct upper circle
        83.06  109.50  113.66  101.30  137.56  115.10  c
        161.46  128.90  169.66  159.50  155.86  183.40  c
        142.06  207.30  111.46  215.50  87.56  201.70  c
        B                                                % Fill and stroke path
        0.5  0.2  1.0  /P1  scn                          % Change nonstroking color
        50  50  m                                        % Start new path
        175  50  l                                       % Construct triangular path
        112.5  158.253  l
        b                                                % Close, fill, and stroke path
    endstream
    endobj
```

4.6.3 Shading Patterns

Shading patterns (PDF 1.3) 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 4.23 shows the contents of this type of dictionary.

TABLE 4.23 Entries in a type 2 pattern dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Pattern** for a pattern dictionary.
PatternType	integer	*(Required)* A code identifying the type of pattern that this dictionary describes; must be 2 for a shading pattern.
Shading	dictionary or stream	*(Required)* A shading object (see below) defining the shading pattern's gradient fill. The contents of the dictionary consist of the entries in Table 4.25 on page 268, plus those in one of Tables 4.26 to 4.31 on pages 271 to 287.
Matrix	array	*(Optional)* An array of six numbers specifying the pattern matrix (see Section 4.6.1, "General Properties of Patterns"). Default value: the identity matrix `[1 0 0 1 0 0]`.
ExtGState	dictionary	*(Optional)* A graphics state parameter dictionary (see Section 4.3.4, "Graphics State Parameter Dictionaries") containing graphics state parameters to be put into effect temporarily while the shading pattern is painted. Any parameters that are not so specified are inherited from the graphics state that was in effect at the beginning of the content stream in which the pattern is defined as a resource.

The most significant entry is **Shading**, whose value is a *shading object* 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. A shading object may be a dictionary or a stream, depending on the type of shading; the term *shading dictionary* is used generically throughout this section to refer to either a dictionary object or the dictionary portion of a stream object. (Those shading objects that are streams are specifically identified as such in the descriptions of particular shading types; unless otherwise stated, they are understood to be simple dictionaries instead.)

By setting a shading pattern as the current color in the graphics state, a PDF content stream can use it with painting operators such as **f** (fill), **S** (stroke), **Tj** (show text), or **Do** (paint external object) with an image mask to paint a path, character 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 that of the object being painted.

Shading Operator

When the area to be painted is a relatively simple shape whose geometry is the same as that of the gradient fill itself, the **sh** operator can be used instead of the usual painting operators. **sh** 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 or a path and works without reference to the current color in the graphics state. Table 4.24 describes the **sh** operator.

***Note:** Patterns defined by type 2 pattern dictionaries do not tile. To create a tiling pattern containing a gradient fill, invoke the **sh** operator from within the content stream of a type 1 (tiling) pattern.*

TABLE 4.24 Shading operator

OPERANDS	OPERATOR	DESCRIPTION
name	**sh**	*(PDF 1.3)* Paint the shape and color shading described by a shading dictionary, subject to the current clipping path. The current color in the graphics state is neither used nor altered. The effect is different from that of painting a path using a shading pattern as the current color. *name* is the name of a shading dictionary resource in the **Shading** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"). All coordinates in the shading dictionary are interpreted relative to the current user space. (By contrast, when a shading dictionary is used in a type 2 pattern, the coordinates are expressed in pattern space.) All colors are interpreted in the color space identified by the shading dictionary's **ColorSpace** entry (see Table 4.25 on page 268). The **Background** entry, if present, is ignored. This operator should be applied only to bounded or geometrically defined shadings. If applied to an unbounded shading, it will paint the shading's gradient fill across the entire clipping region, which may be time-consuming.

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 pseudo-rectangular lattice.
- *Coons patch meshes* (type 6) construct a shading from one or more color patches, each bounded by four cubic Bézier curves.
- *Tensor-product patch meshes* (type 7) are similar to type 6, but with additional control points in each patch, affording greater control over color mapping.

Table 4.25 shows the entries that all shading dictionaries share in common; entries specific to particular shading types are described in the relevant sections below.

Note: *The term* target coordinate space, *used in many of the following descriptions, refers to the coordinate space into which a shading is painted. For shadings used with a type 2 pattern dictionary, this is the pattern coordinate space, discussed in Section 4.6.1, "General Properties of Patterns." For shadings used directly with the* ***sh*** *operator, it is the current user space.*

TABLE 4.25 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. This may be any device, CIE-based, or special color space except a **Pattern** space. See "Color Space: Special Considerations," below, 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 those portions of the area to be painted that lie outside the bounds of the shading object itself. In the opaque imaging model, the effect is as if the painting operation were performed twice: first with the background color and then again with the shading. ***Note:*** *The background color is applied only when the shading is used as part of a shading pattern, not when it is painted directly with the* **sh** *operator.*
BBox	rectangle	*(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 shading's target coordinate space. 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. Anti-aliasing may not be implemented on some output devices, in which case this flag is ignored. Default value: **false**.

Shading types 4 to 7 are defined by a stream containing descriptive data characterizing the shading's gradient fill. In these cases, the shading dictionary is also a stream dictionary and can contain any of the standard entries common to all streams (see Table 3.4 on page 38). In particular, it will always include a **Length** entry, which is required for all streams.

In addition, some shading dictionaries also include a **Function** entry whose value is a function object (dictionary or stream) 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 defines the color transitions across that geometry. The function is required for some types of shading and optional for others. Functions are described in detail in Section 3.9, "Functions."

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. Viewer applications are free to use this strategy as long as the interpolated color values approximate those defined by the shading to within the smoothness tolerance specified in the graphics state (see Section 6.5.2, "Smoothness Tolerance"). 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 to 7) perform interpolation on a parametric value supplied as* input *to the shading's color 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 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 a **DeviceCMYK** color space than in a **Lab** 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 6.2, "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 have been 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 space's alternate color space must be used), gradient fill calculations will be performed in the designated **Separation** or **DeviceN** color space before conversion to the alternate space. Thus, nonlinear tint transformation 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 therefore 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 4.25 on page 268, all shading dictionaries have entries specific to the type of shading they represent, as indicated by the value of their **ShadingType** entry. 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 at every point in the domain is defined by a specified mathematical function. The function need not be 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. Table 4.26 shows the shading dictionary entries specific to this type of shading, in addition to those common to all shading dictionaries (Table 4.25 on page 268).

Note: *This type of shading may not be used with an* **Indexed** *color space.*

TABLE 4.26 Additional entries specific to a type 1 shading dictionary

KEY	TYPE	VALUE
Domain	array	*(Optional)* An array of four numbers [x_{min} x_{max} y_{min} y_{max}] specifying the rectangular domain of coordinates over which the color function(s) are defined. Default value: [0.0 1.0 0.0 1.0].
Matrix	array	*(Optional)* An array of six numbers specifying a transformation matrix mapping the coordinate space specified by the **Domain** entry into the shading's target coordinate space. For example, to map the domain rectangle [0.0 1.0 0.0 1.0] 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	function	*(Required)* A 2-in, *n*-out function or an array of *n* 2-in, 1-out functions (where *n* is the number of color components in the shading dictionary's color space). Each function's domain must be a superset of that of the shading dictionary. If the value returned by the function for a given color component is out of range, it 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 target 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 target coordinate space. 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 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. Table 4.27 shows the shading dictionary entries specific to this type of shading, in addition to those common to all shading dictionaries (Table 4.25 on page 268).

Note: *This type of shading may not be used with an* ***Indexed*** *color space.*

TABLE 4.27 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 shading's target coordinate space.
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 input argument to the color function(s). Default value: [0.0 1.0].
Function	function	*(Required)* A 1-in, *n*-out function or an array of *n* 1-in, 1-out functions (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 **Domain** entry. Each function's domain must be a superset of that of the shading dictionary. If the value returned by the function for a given color component is out of range, it 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 points (0, 0) and (1, 0) in the domain correspond respectively to (x_0, y_0) and (x_1, y_1) on the axis. 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:

$$x' = \frac{(x_1 - x_0) \times (x - x_0) + (y_1 - y_0) \times (y - y_0)}{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

The value of the parametric variable t is then determined from x' as follows:

- For $0 \le x' \le 1$, $t = t_0 + (t_1 - t_0) \times x'$.
- For $x' < 0$, if the first element of 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 element of the **Extend** array is **true**, then $t = t_1$; otherwise, t is undefined and the point is left unpainted.

The resulting value of t is then passed as input 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).

Plate 10 shows three examples of the use of an axial shading to fill a rectangle and display text. The area to be filled extends beyond the shading's bounding box. The shading is the same in all three cases, except for the values of the **Background** and **Extend** entries in the shading dictionary. In the first example, the shading is not extended at either end and no background color is specified, so the shading is clipped to its bounding box at both ends. The second example still has no background color specified, but the shading is extended at both ends; the result is to fill the remaining portions of the filled area with the colors defined at the ends of the shading. In the third example, the shading is extended at both ends and a background color is specified, so the background color is used for the portions of the filled area beyond the ends of the shading.

Type 3 (Radial) Shadings

Type 3 (radial) shadings define a color blend that varies between two circles. Shadings of this type are commonly used to depict three-dimensional spheres and cones. Table 4.28 shows the shading dictionary entries specific to this type of shading, in addition to those common to all shading dictionaries (Table 4.25 on page 268).

Note: *This type of shading may not be used with an* ***Indexed*** *color space.*

TABLE 4.28 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 shading's target coordinate space. 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 input argument to the color function(s). Default value: [0.0 1.0].
Function	function	*(Required)* A 1-in, *n*-out function or an array of *n* 1-in, 1-out functions (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's domain must be a superset of that of the shading dictionary. If the value returned by the function for a given color component is out of range, it 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 elements of the shading dictionary's **Extend** array determine whether and how the shading will be extended. If the first of the two elements is **true**, the shading is extended beyond the defined starting circle to values of s less than 0.0; if the second element is **true**, the shading is extended beyond the defined ending circle to s values greater than 1.0.

Note that either of the starting and 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. The two examples in the rightmost column of Plate 11 depict the results of extending the shading at the smaller and larger ends, respectively.

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 ($t = t_0, s = 0.0$); those extending beyond the ending circle are painted in the color defined for the ending circle ($t = t_1, s = 1.0$). 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, as in Plates 12 and 13. In Plate 12, the inner circle has zero radius; it is the starting circle in the figure on the left and the ending circle in the one on the right. Neither shading is extended at either the smaller or larger end. In Plate 13, the inner circle in both figures has a nonzero radius and the shading is extended at the larger end. In each plate, a background color is specified for the figure on the right, but not for the one on the left.

- 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 (see Plate 11).

Example 4.21 paints the leaf-covered branch shown in Plate 14. Each leaf is filled with the same radial shading (object number 5). The color function (object 10) is a stitching function (described in Section 3.9.3, "Type 3 (Stitching) Functions") whose two subfunctions (objects 11 and 12) are both exponential interpolation functions (see Section 3.9.2, "Type 2 (Exponential Interpolation) Functions"). Each leaf is drawn as a path and then filled with the shading, using code such as that shown in Example 4.22 (where the name Sh1 is associated with object 5 by the **Shading** subdictionary of the current resource dictionary; see Section 3.7.2, "Resource Dictionaries").

Example 4.21

```
5 0 obj                                                    % Shading dictionary
  << /ShadingType 3
     /ColorSpace /DeviceCMYK
     /Coords [0.0 0.0 0.096 0.0 0.0 1.000]                 % Concentric circles
     /Function 10 0 R
     /Extend [true true]
  >>
endobj

10 0 obj                                                   % Color function
  << /FunctionType 3
     /Domain [0.0 1.0]
     /Functions [11 0 R 12 0 R]
     /Bounds [0.708]
     /Encode [1.0 0.0 0.0 1.0]
  >>
endobj
```

```
11  0  obj                                                    % First subfunction
   <<  /FunctionType  2
       /Domain  [0.0  1.0]
       /C0  [0.929  0.357  1.000  0.298]
       /C1  [0.631  0.278  1.000  0.027]
       /N  1.048
   >>
endobj

12  0  obj                                                    % Second subfunction
   <<  /FunctionType  2
       /Domain  [0.0  1.0]
       /C0  [0.929  0.357  1.000  0.298]
       /C1  [0.941  0.400  1.000  0.102]
       /N  1.374
   >>
endobj
```

Example 4.22

```
316.789  140.311  m                                           % Move to start of leaf
303.222  146.388  282.966  136.518  279.122  121.983  c       % Curved segment
277.322  120.182  l                                           % Straight line
285.125  122.688  291.441  121.716  298.156  119.386  c       % Curved segment
336.448  119.386  l                                           % Straight line
331.072  128.643  323.346  137.376  316.789  140.311  c       % Curved segment
W  n                                                          % Set clipping path
q                                                             % Save graphics state
   27.7843  0.0000  0.0000  –27.7843  310.2461  121.1521  cm       % Set matrix
   /Sh1  sh                                                        % Paint shading
Q                                                             % Restore graphics state
```

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. Table 4.29 shows the entries specific to this type of shading dictionary, in addition to those common to all shading dictionaries (Table 4.25 on page 268) and stream dictionaries (Table 3.4 on page 38).

TABLE 4.29 Additional entries specific to a type 4 shading dictionary

KEY	TYPE	VALUE
BitsPerCoordinate	integer	*(Required)* The number of bits used to represent each vertex coordinate. Valid values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Valid values are 1, 2, 4, 8, 12, and 16.
BitsPerFlag	integer	*(Required)* The number of bits used to represent the edge flag for each vertex (see below). Valid values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Valid values for the edge flag itself are 0, 1, and 2.
Decode	array	*(Required)* An array of numbers specifying 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 "Decode Arrays" on page 308). The ranges are specified as follows: $[x_{\min}\ x_{\max}\ y_{\min}\ y_{\max}\ c_{1,\min}\ c_{1,\max}\ \ldots\ c_{n,\min}\ c_{n,\max}]$ Note that only one pair of c values should be specified if a **Function** entry is present.
Function	function	*(Optional)* A 1-in, n-out function or an array of n 1-in, 1-out functions (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. Each input value will be forced into the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. Each function's domain must be a superset of that interval. If the value returned by the function for a given color component is out of range, it will be adjusted to the nearest valid value. This entry may not be used with an **Indexed** color space.

Unlike shading types 1 to 3, types 4 to 7 are represented as streams. Each stream contains a sequence of vertex coordinates and color data that defines the triangle mesh. In a type 4 shading, 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 shading's target coordinate space. 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 4.16.

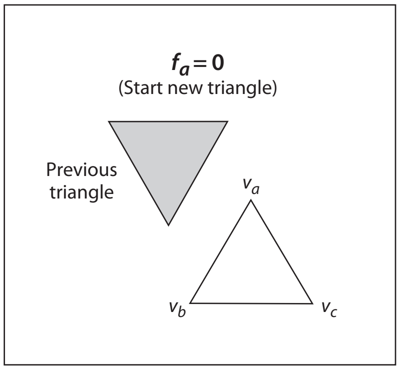


FIGURE 4.16 *Starting a new triangle in a free-form Gouraud-shaded triangle mesh*

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 stream 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 4.17. (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.

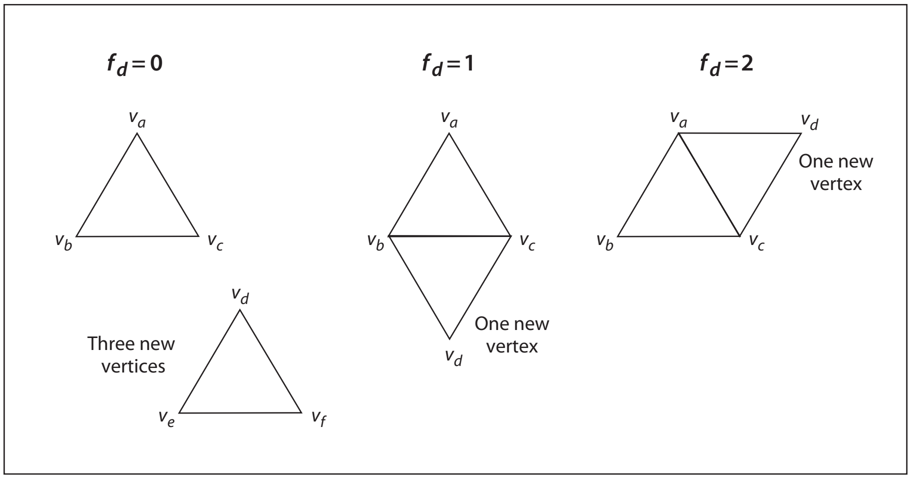


FIGURE 4.17 *Connecting triangles in a free-form Gouraud-shaded triangle mesh*

Complex shapes can be created by using the edge flags to control the edge on which subsequent triangles are formed. Figure 4.18 shows two simple examples. Mesh 1 begins with triangle 1 and uses the following edge flags to draw each succeeding triangle:

1 $(f_a = f_b = f_c = 0)$	7 $(f_i = 2)$
2 $(f_d = 1)$	8 $(f_j = 2)$
3 $(f_e = 1)$	9 $(f_k = 2)$
4 $(f_f = 1)$	10 $(f_l = 1)$
5 $(f_g = 1)$	11 $(f_m = 1)$
6 $(f_h = 1)$	

Mesh 2 again begins with triangle 1 and uses the edge flags

1 $(f_a = f_b = f_c = 0)$	4 $(f_f = 2)$
2 $(f_d = 1)$	5 $(f_g = 2)$
3 $(f_e = 2)$	6 $(f_h = 2)$

The stream must provide vertex data for a whole number of triangles with appropriate edge flags; otherwise, an error will occur.

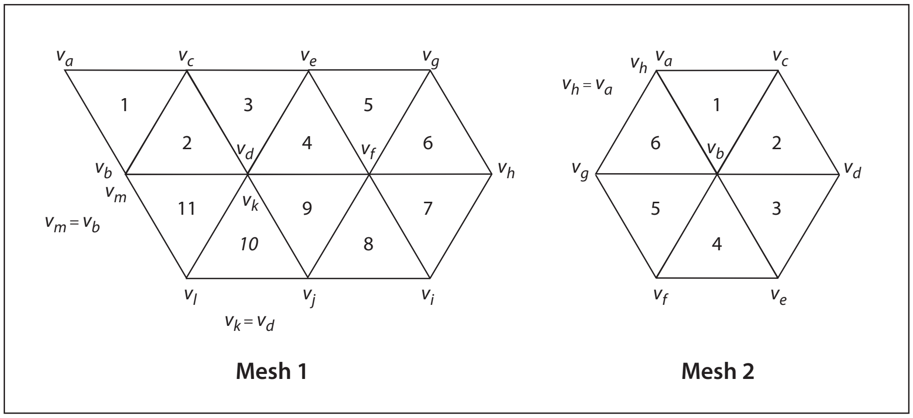


FIGURE 4.18 *Varying the value of the edge flag to create different shapes*

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, expressed in **BitsPerCoordinate** bits each
- A set of *n* color components (where *n* is the number of components in the shading's color space), expressed in **BitsPerComponent** bits each, in the order expected by the **sc** 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 "Decode Arrays" on page 308).

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 at 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 4.19).

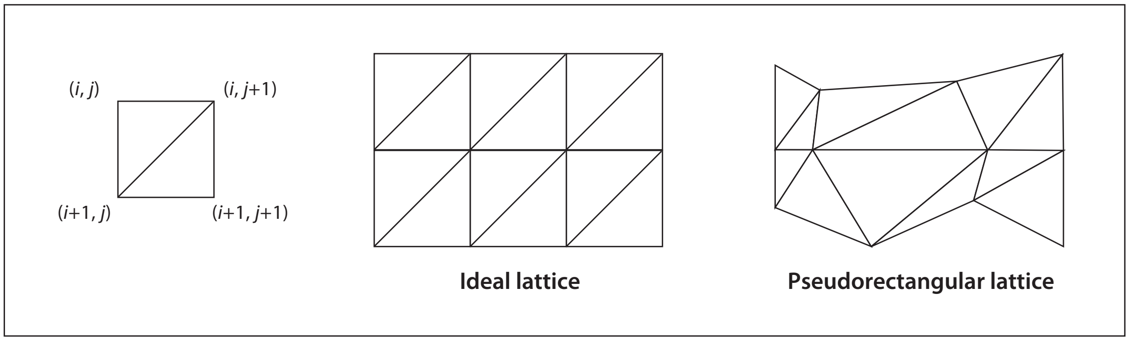


FIGURE 4.19 *Lattice-form triangle meshes*

Table 4.30 shows the shading dictionary entries specific to this type of shading, in addition to those common to all shading dictionaries (Table 4.25 on page 268) and stream dictionaries (Table 3.4 on page 38).

The data stream 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

TABLE 4.30 Additional entries specific to a type 5 shading dictionary

KEY	TYPE	VALUE
BitsPerCoordinate	integer	*(Required)* The number of bits used to represent each vertex coordinate. Valid values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Valid values are 1, 2, 4, 8, 12, and 16.
VerticesPerRow	integer	*(Required)* The number of vertices in each row of the lattice; the value must be greater than or equal to 2. The number of rows need not be specified.
Decode	array	*(Required)* An array of numbers specifying 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 "Decode Arrays" on page 308). 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	function	*(Optional)* A 1-in, *n*-out function or an array of *n* 1-in, 1-out functions (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. Each input value will be forced into the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. Each function's domain must be a superset of that interval. If the value returned by the function for a given color component is out of range, it will be adjusted to the nearest valid value. This entry may not be used with an **Indexed** color space.

All vertex coordinates are expressed in the shading's target coordinate space. 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 4.19:

$$\begin{array}{ll}(V_{i,j},\ V_{i,j+1},\ V_{i+1,j}) & \text{for } 0 \le i \le m-2,\ 0 \le j \le k-2 \\ (V_{i,j+1},\ V_{i+1,j},\ V_{i+1,j+1}) & \end{array}$$

See "Type 4 Shadings (Free-Form Gouraud-Shaded Triangle Meshes)" on page 277 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 cubic 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 (see the upper figure in Plate 15).
- Coordinates are mapped from the unit square into a four-sided patch whose sides are not necessarily linear (see the lower figure in Plate 15). 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 4.20.

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 vertically across the unit square. The four corners of the Coons patch satisfy the equations

$$\begin{aligned} C_1(0) &= D_1(0) \\ C_1(1) &= D_2(0) \\ C_2(0) &= D_1(1) \\ C_2(1) &= D_2(1) \end{aligned}$$

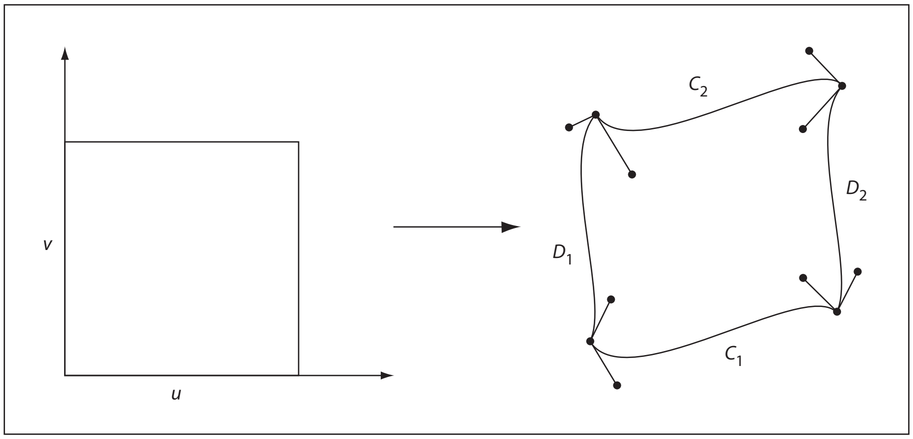


FIGURE 4.20 *Coordinate mapping from a unit square to a four-sided Coons patch*

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 stream 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) might not be the same as the patch boundary if, for example, the patch folds over on itself, as shown in Figure 4.21.

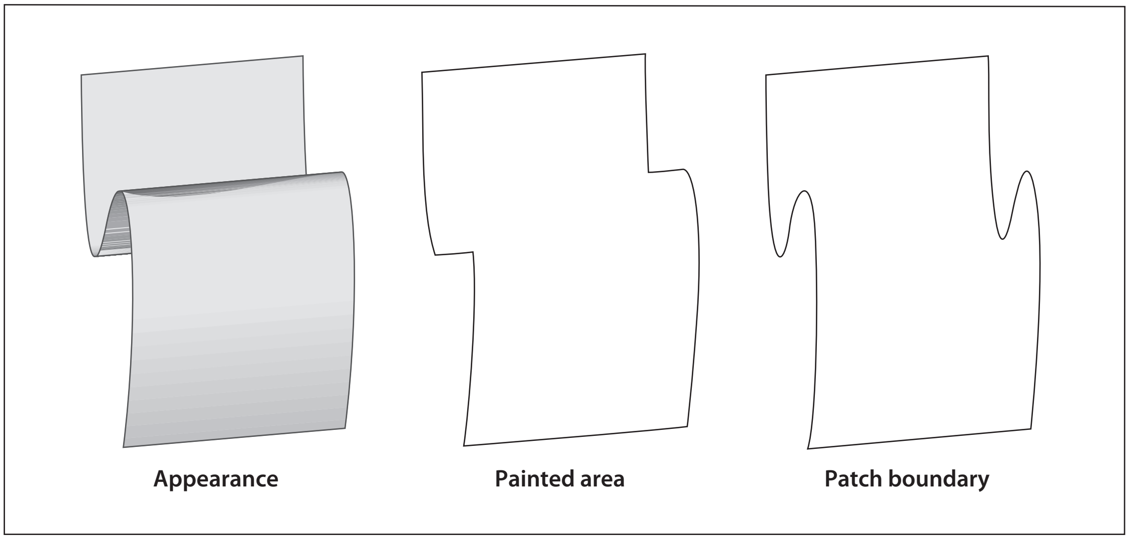


FIGURE 4.21 *Painted area and boundary of a Coons patch*

Table 4.31 shows the shading dictionary entries specific to this type of shading, in addition to those common to all shading dictionaries (Table 4.25 on page 268) and stream dictionaries (Table 3.4 on page 38).

TABLE 4.31 Additional entries specific to a type 6 shading dictionary

KEY	TYPE	VALUE
BitsPerCoordinate	integer	*(Required)* The number of bits used to represent each geometric coordinate. Valid values are 1, 2, 4, 8, 12, 16, 24, and 32.
BitsPerComponent	integer	*(Required)* The number of bits used to represent each color component. Valid values are 1, 2, 4, 8, 12, and 16.
BitsPerFlag	integer	*(Required)* The number of bits used to represent the edge flag for each patch (see below). Valid values of **BitsPerFlag** are 2, 4, and 8, but only the least significant 2 bits in each flag value are used. Valid values for the edge flag itself are 0, 1, 2, and 3.
Decode	array	*(Required)* An array of numbers specifying 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 "Decode Arrays" on page 308). The ranges are specified as follows: $[x_{\min}\ x_{\max}\ \ y_{\min}\ y_{\max}\ \ c_{1,\min}\ c_{1,\max}\ \ldots\ c_{n,\min}\ c_{n,\max}]$ Note that only one pair of *c* values should be specified if a **Function** entry is present.
Function	function	*(Optional)* A 1-in, *n*-out function or an array of *n* 1-in, 1-out functions (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. Each input value will be forced into the range interval specified for the corresponding color component in the shading dictionary's **Decode** array. Each function's domain must be a superset of that interval. If the value returned by the function for a given color component is out of range, it will be adjusted to the nearest valid value. This entry may not be used with an **Indexed** color space.

The data stream 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 shading's target coordinate space. See "Type 4 Shadings (Free-Form Gouraud-

Shaded Triangle Meshes)" on page 277 for further details on the format of the data.

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 4.22 shows how these control points correspond to the cubic Bézier curves C_1, C_2, D_1, and D_2 identified in Figure 4.20 on page 285. 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.

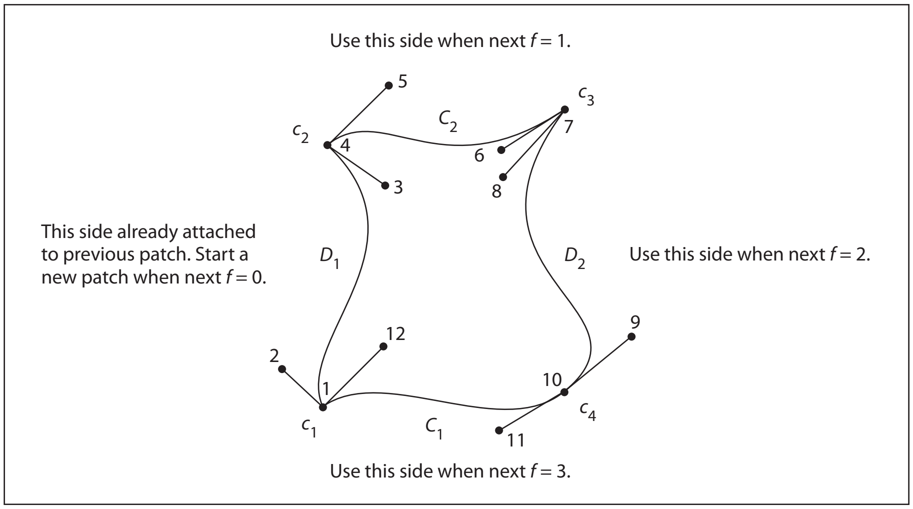

Use this side when next f = 1.
5
c3
C2
7
c2
4
6
3
8
This side already attached to previous patch. Start a new patch when next f = 0.
D1
D2
Use this side when next f = 2.
9
12
2
1
10
c4
c1
C1
11
Use this side when next f = 3.

FIGURE 4.22 *Color values and edge flags in Coons patch meshes*

Figure 4.22 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 4.23), and so are not explicitly repeated in the data stream. Table 4.32 summarizes the required data values for various values of the edge flag.

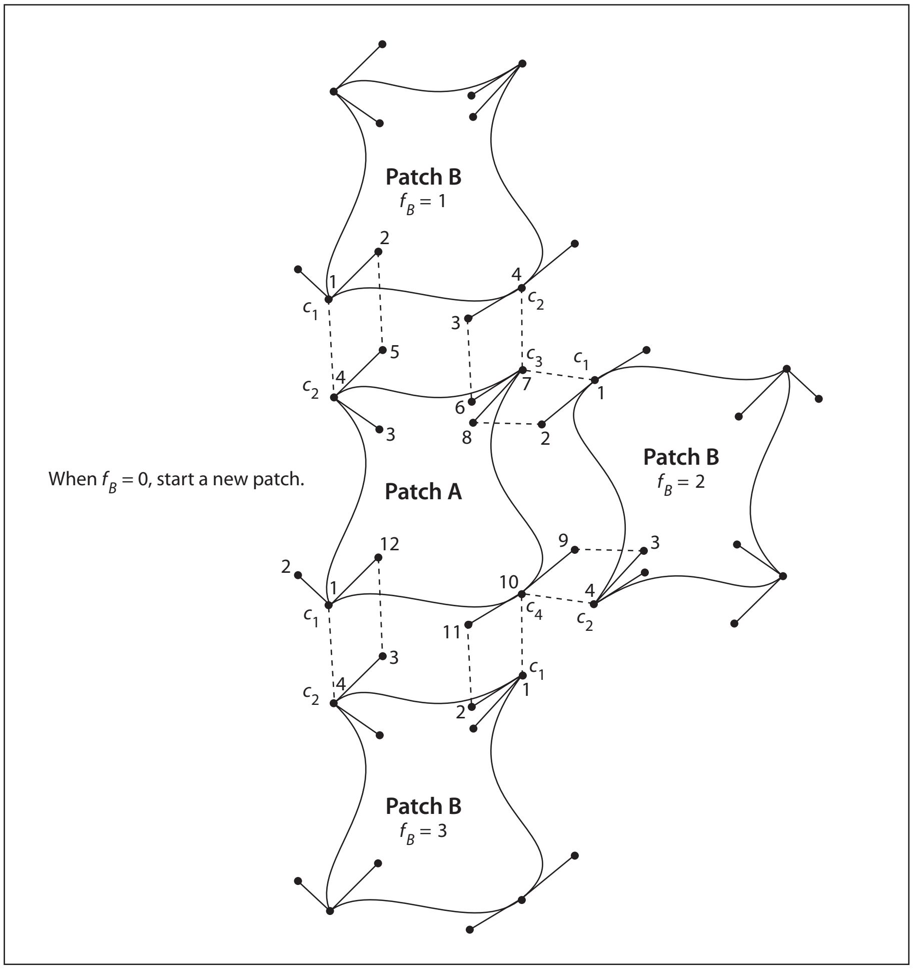


FIGURE 4.23 *Edge connections in a Coons patch mesh*

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 at each point.

TABLE 4.32 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

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 stream.

Although the Coons patch is more concise and easier to use, the tensor-product patch affords greater control over color mapping.

Note: *The data format for type 7 shadings (as for types 4 through 6) is the same in PDF as it is in PostScript. However, the numbering and order of control points was described incorrectly in the first printing of the* PostScript Language Reference, *Third Edition. That description has been corrected here.*

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 4.24):

$$\begin{array}{cccc} p_{03} & p_{13} & p_{23} & p_{33} \\ p_{02} & p_{12} & p_{22} & p_{32} \\ p_{01} & p_{11} & p_{21} & p_{31} \\ p_{00} & p_{10} & p_{20} & p_{30} \end{array}$$

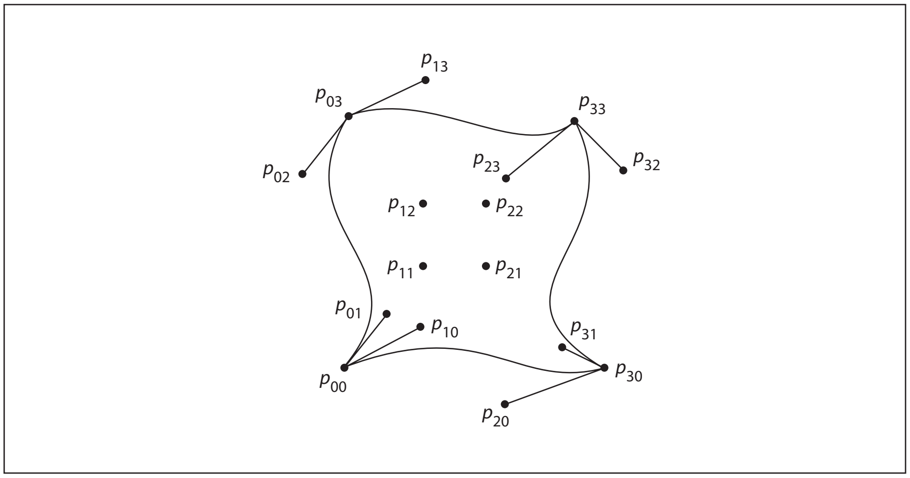


FIGURE 4.24 *Control points in a tensor-product patch*

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 column i and row 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 bottom boundary of the patch to the top. At the bottom and top, the control points of this curve coincide with those of the patch's bottom ($p_{00} \ldots p_{30}$) and top ($p_{03} \ldots p_{33}$) boundary curves, respectively. As the curve moves from the bottom edge of the patch to the top, 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 $p_{00} \ldots p_{03}$, the trajectory of the ending point is defined by points $p_{30} \ldots p_{33}$, and those of the two intermediate control points by $p_{10} \ldots p_{13}$ and $p_{20} \ldots p_{23}$. Equivalently, the patch can be considered to be traced by a cubic 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 implicitly defined by the boundary curves. The values of the internal control points are given by the equations

$$\begin{aligned}
p_{11} &= 1/9 \times \\
&\quad [-4 \times p_{00} + 6 \times (p_{01} + p_{10}) - 2 \times (p_{03} + p_{30}) + 3 \times (p_{31} + p_{13}) - 1 \times p_{33}] \\
p_{12} &= 1/9 \times \\
&\quad [-4 \times p_{03} + 6 \times (p_{02} + p_{13}) - 2 \times (p_{00} + p_{33}) + 3 \times (p_{32} + p_{10}) - 1 \times p_{30}] \\
p_{21} &= 1/9 \times \\
&\quad [-4 \times p_{30} + 6 \times (p_{31} + p_{20}) - 2 \times (p_{33} + p_{00}) + 3 \times (p_{01} + p_{23}) - 1 \times p_{03}] \\
p_{22} &= 1/9 \times \\
&\quad [-4 \times p_{33} + 6 \times (p_{32} + p_{23}) - 2 \times (p_{30} + p_{03}) + 3 \times (p_{02} + p_{20}) - 1 \times p_{00}]
\end{aligned}$$

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 specified in the shading's data stream in the following order:

4	5	6	7
3	14	15	8
2	13	16	9
1	12	11	10

All control point coordinates are expressed in the shading's target coordinate space. 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* is 0, all 16 control points and four corner colors must be explicitly specified in the data stream; if *f* is 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 stream. Table 4.33 summarizes the data values for various values of the edge flag *f*, expressed in terms of the row and column indices used in Figure 4.24 above. See "Type 4 Shadings (Free-Form

Gouraud-Shaded Triangle Meshes)” on page 277 for further details on the format of the data.

TABLE 4.33 Data values in a tensor-product patch mesh

EDGE FLAG	NEXT SET OF DATA VALUES
$f = 0$	$x_{00}\ y_{00}\ x_{01}\ y_{01}\ x_{02}\ y_{02}\ x_{03}\ y_{03}\ x_{13}\ y_{13}\ x_{23}\ y_{23}\ x_{33}\ y_{33}\ x_{32}\ y_{32}$ $x_{31}\ y_{31}\ x_{30}\ y_{30}\ x_{20}\ y_{20}\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}\ x_{22}\ y_{22}\ x_{21}\ y_{21}$ $c_{00}\ c_{03}\ c_{33}\ c_{30}$ New patch; no implicit values
$f = 1$	$x_{13}\ y_{13}\ x_{23}\ y_{23}\ x_{33}\ y_{33}\ x_{32}\ y_{32}\ x_{31}\ y_{31}\ x_{30}\ y_{30}$ $x_{20}\ y_{20}\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}\ x_{22}\ y_{22}\ x_{21}\ y_{21}$ $c_{33}\ c_{30}$ Implicit values: $(x_{00}, y_{00}) = (x_{03}, y_{03})$ previous $c_{00} = c_{03}$ previous $(x_{01}, y_{01}) = (x_{13}, y_{13})$ previous $c_{03} = c_{33}$ previous $(x_{02}, y_{02}) = (x_{23}, y_{23})$ previous $(x_{03}, y_{03}) = (x_{33}, y_{33})$ previous
$f = 2$	$x_{13}\ y_{13}\ x_{23}\ y_{23}\ x_{33}\ y_{33}\ x_{32}\ y_{32}\ x_{31}\ y_{31}\ x_{30}\ y_{30}$ $x_{20}\ y_{20}\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}\ x_{22}\ y_{22}\ x_{21}\ y_{21}$ $c_{33}\ c_{30}$ Implicit values: $(x_{00}, y_{00}) = (x_{33}, y_{33})$ previous $c_{00} = c_{33}$ previous $(x_{01}, y_{01}) = (x_{32}, y_{32})$ previous $c_{03} = c_{30}$ previous $(x_{02}, y_{02}) = (x_{31}, y_{31})$ previous $(x_{03}, y_{03}) = (x_{30}, y_{30})$ previous
$f = 3$	$x_{13}\ y_{13}\ x_{23}\ y_{23}\ x_{33}\ y_{33}\ x_{32}\ y_{32}\ x_{31}\ y_{31}\ x_{30}\ y_{30}$ $x_{20}\ y_{20}\ x_{10}\ y_{10}\ x_{11}\ y_{11}\ x_{12}\ y_{12}\ x_{22}\ y_{22}\ x_{21}\ y_{21}$ $c_{33}\ c_{30}$ Implicit values: $(x_{00}, y_{00}) = (x_{30}, y_{30})$ previous $c_{00} = c_{30}$ previous $(x_{01}, y_{01}) = (x_{20}, y_{20})$ previous $c_{03} = c_{00}$ previous $(x_{02}, y_{02}) = (x_{10}, y_{10})$ previous $(x_{03}, y_{03}) = (x_{00}, y_{00})$ previous

4.7 External Objects

An *external object* (commonly called an *XObject*) is a graphics object whose contents are defined by a self-contained content stream, separate from the content stream in which it is used. There are three types of external object:

- An *image XObject* (Section 4.8.4, "Image Dictionaries") represents a sampled visual image such as a photograph.
- A *form XObject* (Section 4.9, "Form XObjects") is a self-contained description of an arbitrary sequence of graphics objects.
- A *PostScript XObject* (Section 4.7.1, "PostScript XObjects") contains a fragment of code expressed in the PostScript page description language. PostScript XObjects are no longer recommended to be used.

Two further categories of external objects, *group XObjects* and *reference XObjects (both PDF 1.4)*, are actually specialized types of form XObject with additional properties; see Sections 4.9.2, "Group XObjects," and 4.9.3, "Reference XObjects," for additional information.

Any XObject can be painted as part of another content stream by means of the **Do** operator (see Table 4.34). This operator applies to any type of XObject—image, form, or PostScript. The syntax is the same in all cases, although details of the operator's behavior differ depending on the type. (See implementation note 43 in Appendix H.)

TABLE 4.34 XObject operator

OPERANDS	OPERATOR	DESCRIPTION
name	**Do**	Paint the specified XObject. The operand *name* must appear as a key in the **XObject** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"); the associated value must be a stream whose **Type** entry, if present, is **XObject**. The effect of **Do** depends on the value of the XObject's **Subtype** entry, which may be **Image** (see Section 4.8.4, "Image Dictionaries"), **Form** (Section 4.9, "Form XObjects"), or **PS** (Section 4.7.1, "PostScript XObjects").

4.7.1 PostScript XObjects

Starting with PDF 1.1, a content stream can include PostScript language fragments. These fragments are used only when printing to a PostScript output device; they have no effect either when viewing the document on-screen or when printing to a non-PostScript device. In addition, applications that understand PDF are unlikely to be able to interpret the PostScript fragments. Hence, this capability should be used with extreme caution and only if there is no other way to achieve the same result. Inappropriate use of PostScript XObjects can cause PDF files to print incorrectly.

***Note:** Since PDF 1.4 encompasses all of the Adobe imaging model features of the PostScript language, there is no longer any reason to use PostScript XObjects. This feature is likely to be removed from PDF in a future version.*

A *PostScript XObject* is an XObject stream whose **Subtype** entry has the value **PS**. Table 4.35 shows the contents of a PostScript XObject dictionary, in addition to the usual entries common to all streams (see Table 3.4 on page 38).

TABLE 4.35 Additional entries specific to a PostScript XObject dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **XObject** for a PostScript XObject.
Subtype	name	*(Required)* The type of XObject that this dictionary describes; must be **PS** for a PostScript XObject. ***Note:** Alternatively, the value of this entry may be **Form**, with an additional **Subtype2** entry whose value is **PS**.*
Level1	stream	*(Optional)* A stream whose contents are to be used in place of the PostScript XObject's stream when the target PostScript interpreter is known to support only LanguageLevel 1.

When a PDF content stream is translated into the PostScript language, any **Do** operation that references a PostScript XObject is replaced by the contents of the XObject stream itself. The stream is copied without interpretation. The PostScript fragment may use Type 1 and TrueType fonts listed in the **Font** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"), accessing them by their **BaseFont** names using the PostScript **findfont** operator. The fragment may not use other types of font listed in the **Font** subdictionary. It should not reference the PostScript definitions corresponding to PDF procedure sets (see Section 10.1, "Procedure Sets"), which are subject to change.

4.8 Images

PDF'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.

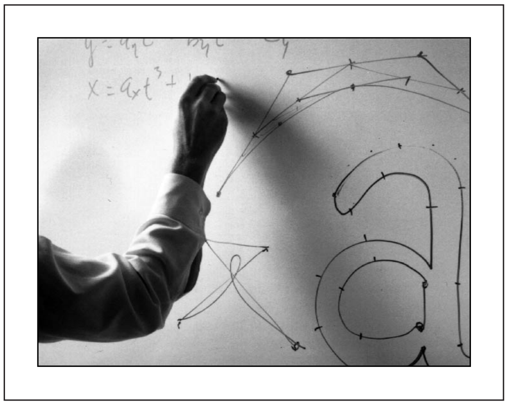


FIGURE 4.25 *Typical sampled image*

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 is a 1-, 2-, 4-, 8-, or (in PDF 1.5) 16-bit integer, permitting the representation of 2, 4, 16, 256, or (in PDF 1.5) 65536 distinct values for each component. (Other component sizes can be accommodated when a **JPXDecode** filter is used; see Section 3.3.8, "JPXDecode Filter.)

PDF provides two means for specifying images:

- An *image XObject* (described in Section 4.8.4, "Image Dictionaries") is a stream object whose dictionary specifies attributes of the image and whose data contains the image samples. Like all external objects, it is painted on the page by invoking the **Do** operator in a content stream (see Section 4.7, "External Objects"). Image XObjects have other uses as well, such as for alternate images (see "Alternate Images" on page 310), image masks (Section 4.8.5, "Masked Images"), and thumbnail images (Section 8.2.3, "Thumbnail Images").
- An *inline image* is a small image that is completely defined—both attributes and data—directly inline within a content stream. The kinds of image that can be represented in this way are limited; see Section 4.8.6, "Inline Images," for details.

4.8.1 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. A PDF viewer application usually renders images by a sampling 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, four interrelated items must be specified:

- The format of the image: number of columns (width), number of rows (height), number of color components per sample, and number of bits per color component
- The sample data constituting the image's visual content
- The correspondence between coordinates in user space and those in the image's own internal coordinate space, defining the region of user space that will receive the image
- The mapping from color component values in the image data to component values in the image's color space

All of these items are specified explicitly or implicitly by an image XObject or an inline image.

Note: *For convenience, the following sections refer consistently to the object defining an image as an* image dictionary. *Although this term properly refers only to the dictionary portion of the stream object representing an image XObject, it should be understood to apply equally to the stream's data portion or to the parameters and data of an inline image.*

4.8.2 Sample Representation

The source format for an image can be described by four parameters:

- The width of the image in samples
- The height of the image in samples
- The number of color components per sample
- The number of bits per color component

The image dictionary specifies the width, the height, and the number of bits per component explicitly; the number of color components can be inferred from the color space specified in the dictionary.

Note: *For images using the **JPXDecode** filter (see Section 3.3.8, "JPXDecode Filter"), the number of bits per component is determined from the image data and not specified in the image dictionary; the color space may or may not be specified in the dictionary.*

Sample data is represented as a stream of bytes, interpreted as 8-bit unsigned integers in the range 0 to 255. The bytes constitute a continuous bit stream, with the high-order bit of each byte first. This bit stream, in turn, is divided into units of *n* bits each, where *n* is the number of bits per component. Each unit encodes a color component value, given with high-order bit first; units of 16 bits are given with the most significant byte first. Byte boundaries are ignored, except that each row of sample data must begin on a byte boundary. If the number of data bits per row is not a multiple of 8, the end of the row is padded with extra bits to fill out the last byte. A PDF viewer application ignores these padding bits.

Each *n*-bit unit within the bit stream is interpreted as an unsigned integer in the range 0 to $2^n - 1$, with the high-order bit first; the image dictionary's **Decode** entry maps this to a color component value, equivalent to what could be used with color operators such as **sc** or **g**. Color components are 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.

Normally, the color samples in an image are interpreted according to the color space specified in the image dictionary (see Section 4.5, "Color Spaces"), without reference to the color parameters in the graphics state. However, if the image dictionary's **ImageMask** entry is **true**, the sample data is interpreted as a *stencil mask* for applying the graphics state's nonstroking color parameters (see "Stencil Masking" on page 313).

4.8.3 Image Coordinate System

Each image has its own internal coordinate system, or *image space*. The image occupies a rectangle in image space w units wide and h units high, where w and h are the width and height of image in samples. Each sample occupies one square unit. The coordinate origin $(0, 0)$ is at the upper-left corner of the image, with coordinates ranging from 0 to w horizontally and 0 to h vertically.

The image's sample data is ordered by row, with the horizontal coordinate varying most rapidly. This is shown in Figure 4.26, where the numbers inside the squares indicate the order of the samples, counting from 0. The upper-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 upper-left corner is at $(w - 1, 0)$ and whose upper-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 upper-left corner is at $(w - 1, h - 1)$ and whose lower-right corner is at (w, h).

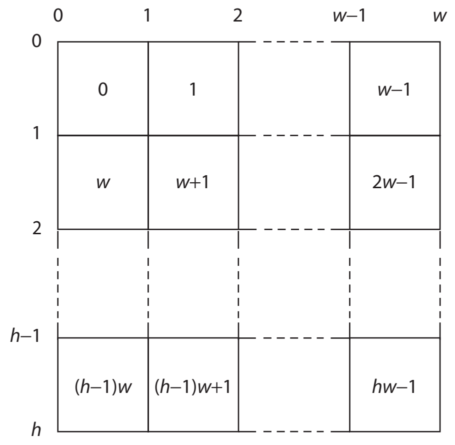


FIGURE 4.26 *Source image coordinate system*

Note: *The image coordinate system and scanning order imposed by PDF do not preclude using different conventions in the actual image. Coordinate transformations can be used to map from other conventions to the PDF convention.*

The correspondence between image space and user space is constant: the unit square of user space, bounded by user coordinates (0, 0) and (1, 1), corresponds to the boundary of the image in image space (see Figure 4.27). Following the normal convention for user space, the coordinate (0, 0) is at the *lower-left* corner of this square, corresponding to coordinates (0, *h*) in image space. The transformation from image space to user space could be described by the matrix [1/*w* 0 0 −1/*h* 0 1].

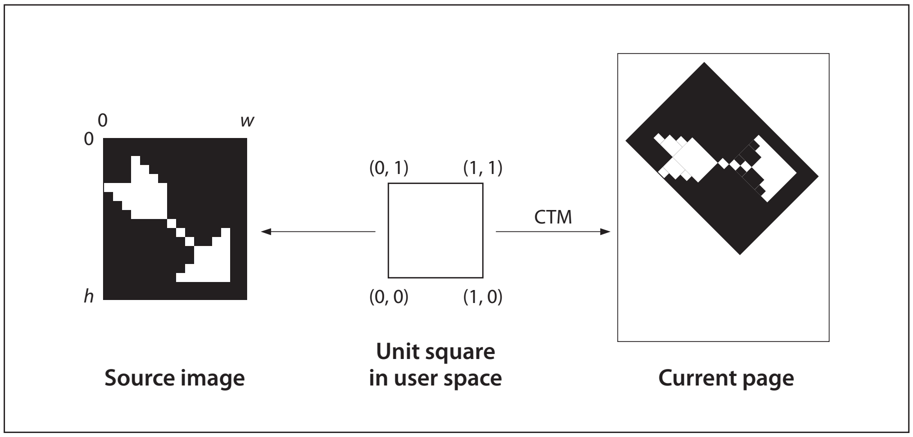


FIGURE 4.27 *Mapping the source image*

An image can be placed on the output page in any desired position, orientation, and size by using the **cm** operator to modify the current transformation matrix (CTM) so as to map the unit square of user space to the rectangle or parallelogram in which the image is to be painted. Typically, this is done within a pair of **q** and **Q** operators to isolate the effect of the transformation, which can include translation, rotation, reflection, and skew (see Section 4.2, "Coordinate Systems"). For example, if the **XObject** subdictionary of the current resource dictionary defines the name Image1 to denote an image XObject, the code shown in Example 4.23 paints the image in 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 units high.

Example 4.23

```
q                                              % Save graphics state
   1 0 0 1 100 200 cm                          % Translate
   0.7071 0.7071 –0.7071 0.7071 0 0 cm         % Rotate
   150 0 0 80 0 0 cm                           % Scale
   /Image1 Do                                  % Paint image
Q                                              % Restore graphics state
```

(As discussed in Section 4.2.3, "Transformation Matrices," these three transformations could be combined into one.) Of course, if the aspect ratio (width to height) of the original image in this example is different from 150:80, the result will be distorted.

4.8.4 Image Dictionaries

Table 4.36 lists the entries in an image dictionary—that is, in the dictionary portion of a stream representing an image XObject—in addition to the usual entries common to all streams (see Table 3.4 on page 38). 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 an error.

***Note:** The entries described here are those that are appropriate for a* base image*—one that is invoked directly with the **Do** operator. Some of these entries are not relevant for images used in other ways, such as for alternate images (see "Alternate Images" on page 310), image masks (Section 4.8.5, "Masked Images"), or thumbnail images (Section 8.2.3, "Thumbnail Images"). Except as noted, such irrelevant entries are simply ignored.*

TABLE 4.36 Additional entries specific to an image dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **XObject** for an image XObject.
Subtype	name	*(Required)* The type of XObject that this dictionary describes; must be **Image** for an image XObject.
Width	integer	*(Required)* The width of the image, in samples.
Height	integer	*(Required)* The height of the image, in samples.
ColorSpace	name or array	*(Required for images, except those that use the **JPXDecode** filter; not allowed for image masks)* The color space in which image samples are specified. This may be any type of color space except **Pattern**. If the image uses the **JPXDecode** filter, this entry is optional • If it is present, any color space specifications in the JPEG2000 data are ignored. • If it is absent, the color space specifications in the JPEG2000 data are used. The **Decode** array is then also ignored unless **ImageMask** is **true**.

KEY	TYPE	VALUE
BitsPerComponent	integer	*(Required except for image masks and images that use the **JPXDecode** filter)* The number of bits used to represent each color component. Only a single value may be specified; the number of bits is the same for all color components. Valid values are 1, 2, 4, 8, and (in PDF 1.5) 16. If **ImageMask** is **true**, this entry is optional, and if specified, its value must be 1. If the image stream uses a filter, the value of **BitsPerComponent** must be consistent with the size of the data samples that the filter delivers. In particular, a **CCITTFaxDecode** or **JBIG2Decode** filter always delivers 1-bit samples, a **RunLengthDecode** or **DCTDecode** filter delivers 8-bit samples, and an **LZWDecode** or **FlateDecode** filter delivers samples of a specified size if a predictor function is used. If the image stream uses the **JPXDecode** filter, this entry is optional and ignored if present. The bit depth is determined in the process of decoding the JPEG2000 image.
Intent	name	*(Optional; PDF 1.1)* The name of a color rendering intent to be used in rendering the image (see "Rendering Intents" on page 230). Default value: the current rendering intent in the graphics state.
ImageMask	boolean	*(Optional)* A flag indicating whether the image is to be treated as an image mask (see Section 4.8.5, "Masked Images"). If this flag is **true**, the value of **BitsPerComponent** must be 1 and **Mask** and **ColorSpace** should not be specified; unmasked areas will be painted using the current nonstroking color. Default value: **false**.
Mask	stream or array	*(Optional except for image masks; not allowed for image masks; PDF 1.3)* An image XObject defining an image mask to be applied to this image (see "Explicit Masking" on page 314), or an array specifying a range of colors to be applied to it as a color key mask (see "Color Key Masking" on page 315). If **ImageMask** is **true**, this entry must not be present. (See implementation note 44 in Appendix H.)

KEY	TYPE	VALUE
SMask	stream	*(Optional; PDF 1.4)* A subsidiary image XObject defining a *soft-mask image* (see “Soft-Mask Images” on page 512) to be used as a source of mask shape or mask opacity values in the transparent imaging model. The alpha source parameter in the graphics state determines whether the mask values are interpreted as shape or opacity. If present, this entry overrides the current soft mask in the graphics state, as well as the image’s **Mask** entry, if any. (However, the other transparency-related graphics state parameters—blend mode and alpha constant—remain in effect.) If **SMask** is absent, the image has no associated soft mask (although the current soft mask in the graphics state may still apply).
SMaskInData	integer	*(Optional for images that use the* ***JPXDecode*** *filter, meaningless otherwise; PDF 1.5)* A code specifying how soft-mask information (see “Soft-Mask Images” on page 512) encoded with image samples should be used: 0 If present, encoded soft-mask image information should be ignored. 1 The image’s data stream includes encoded soft-mask values. A viewer application can create a soft-mask image from the information to be used as a source of mask shape or mask opacity in the transparency imaging model. 2 The image’s data stream includes color channels that have been pre-blended with a background; the image data also includes an opacity channel. A viewer application can create a soft-mask image with a **Matte** entry from the opacity channel information to be used as a source of mask shape or mask opacity in the transparency model. If this entry has a non-zero value, **SMask** should not be specified. See also Section 3.3.8, “JPXDecode Filter.” Default value: 0.
Decode	array	*(Optional)* An array of numbers describing how to map image samples into the range of values appropriate for the image’s color space (see “Decode Arrays” on page 308). If **ImageMask** is **true**, the array must be either [0 1] or [1 0]; otherwise, its length must be twice the number of color components required by **ColorSpace**. If the image uses the **JPXDecode** filter and **ImageMask** is false, **Decode** is ignored. Default value: see “Decode Arrays” on page 308.

KEY	TYPE	VALUE
Interpolate	boolean	*(Optional)* A flag indicating whether image interpolation is to be performed (see "Image Interpolation" on page 310). Default value: **false**.
Alternates	array	*(Optional; PDF 1.3)* An array of alternate image dictionaries for this image (see "Alternate Images" on page 310). The order of elements within the array has no significance. This entry may not be present in an image XObject that is itself an alternate image.
Name	name	*(Required in PDF 1.0; optional otherwise)* The name by which this image XObject is referenced in the **XObject** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"). ***Note:*** *This entry is obsolescent and its use is no longer recommended. (See implementation note 45 in Appendix H.)*
StructParent	integer	*(Required if the image is a structural content item; PDF 1.3)* The integer key of the image's entry in the structural parent tree (see "Finding Structure Elements from Content Items" on page 739).
ID	string	*(Optional; PDF 1.3; indirect reference preferred)* The digital identifier of the image's parent Web Capture content set (see Section 10.9.5, "Object Attributes Related to Web Capture").
OPI	dictionary	*(Optional; PDF 1.2)* An OPI version dictionary for the image (see Section 10.10.6, "Open Prepress Interface (OPI)"). If **ImageMask** is **true**, this entry is ignored.
Metadata	stream	*(Optional; PDF 1.4)* A *metadata stream* containing metadata for the image (see Section 10.2.2, "Metadata Streams").
OC	dictionary	*(Optional; PDF 1.5)* An optional content group or optional content membership dictionary (see Section 4.10, "Optional Content"), specifying the optional content properties for this image XObject. Before the image is processed, its visibility is determined based on this entry; if it is determined to be invisible, the entire image is skipped, as if there were no **Do** operator to invoke it.

Example 4.24 defines an image 256 samples wide by 256 high, with 8 bits per sample in the **DeviceGray** color space. It paints the image on a page with its lower-left corner positioned at coordinates (45, 140) in current user space and scaled to a width and height of 132 user space units.

Example 4.24

```
20 0 obj                                        % Page object
   << /Type /Page
      /Parent 1 0 R
      /Resources 21 0 R
      /MediaBox [0 0 612 792]
      /Contents 23 0 R
   >>
endobj

21 0 obj                                        % Resource dictionary for page
   << /ProcSet [/PDF /ImageB]
      /XObject << /Im1 22 0 R >>
   >>
endobj

22 0 obj                                        % Image XObject
   << /Type /XObject
      /Subtype /Image
      /Width 256
      /Height 256
      /ColorSpace /DeviceGray
      /BitsPerComponent 8
      /Length 83183
      /Filter /ASCII85Decode
   >>
stream
9LhZI9h\GY9i+bb;,p:e;G9SP92/)X9MJ>^:f14d;,U(X8P;cO;G9e];c$=k9Mn\]
… Image data representing 65,536 samples …
8P;cO;G9e];c$=k9Mn\]~>
endstream
endobj

23 0 obj                                        % Contents of page
   << /Length 56 >>
stream
   q                                            % Save graphics state
      132 0 0 132 45 140 cm                     % Translate to (45,140) and scale by 132
      /Im1 Do                                   % Paint image
   Q                                            % Restore graphics state
endstream
endobj
```

Decode Arrays

An image's data stream is initially decomposed into integers in the domain 0 to $2^n - 1$, where n is the value of the image dictionary's **BitsPerComponent** entry. The image's **Decode** array specifies a linear mapping of each integer component value to a number that would be appropriate as a component value in the image's color space.

Each pair of numbers in a **Decode** array specifies the lower and upper values to which the domain of sample values in the image is mapped. A **Decode** array contains one pair of numbers for each component in the color space specified by the image's **ColorSpace** entry. The mapping for each color component is a linear transformation. That is, it uses the following formula for linear interpolation:

$$\begin{aligned} y &= \text{Interpolate}(x, x_{\min}, x_{\max}, y_{\min}, y_{\max}) \\ &= y_{\min} + \left((x - x_{\min}) \times \frac{y_{\max} - y_{\min}}{x_{\max} - x_{\min}} \right) \end{aligned}$$

Generally, this is used to convert a value x between $x_{\min}$ and $x_{\max}$ to a corresponding value y between $y_{\min}$ and $y_{\max}$, projecting along the line defined by the points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. While this formula applies to values outside the domain $x_{\min}$ to $x_{\max}$ and does not require that $x_{\min} < x_{\max}$, note that interpolation used for color conversion, such as the **Decode** array, does require that $x_{\min} < x_{\max}$ and "clips" x values to this domain, so that $y = y_{\min}$ for all $x \leq x_{\min}$, and $y = y_{\max}$ for all $x \geq x_{\max}$.

For a **Decode** array of the form $[D_{\min}\ \ D_{\max}]$, this can be written as

$$\begin{aligned} y &= \text{Interpolate}(x, 0, 2^n - 1, D_{\min}, D_{\max}) \\ &= D_{\min} + \left(x \times \frac{D_{\max} - D_{\min}}{2^n - 1} \right) \end{aligned}$$

where

n is the value of **BitsPerComponent**

x is the input value, in the domain 0 to $2^n - 1$

$D_{\min}$ and $D_{\max}$ are the values specified in the **Decode** array

y is the output value, to be interpreted in the image's color space

Samples with a value of 0 are mapped to D_{min}, those with a value of $2^n - 1$ are mapped to D_{max}, and those with intermediate values are mapped linearly between D_{min} and D_{max}. Table 4.37 lists the default **Decode** arrays for use with the various color spaces. For most color spaces, the **Decode** arrays listed in the table map into the full range of allowed component values. For an **Indexed** color space, the default **Decode** array ensures that component values that index a color table are passed through unchanged.

TABLE 4.37 Default Decode arrays

COLOR SPACE	Decode ARRAY
DeviceGray	[0.0 1.0]
DeviceRGB	[0.0 1.0 0.0 1.0 0.0 1.0]
DeviceCMYK	[0.0 1.0 0.0 1.0 0.0 1.0 0.0 1.0]
CalGray	[0.0 1.0]
CalRGB	[0.0 1.0 0.0 1.0 0.0 1.0]
Lab	[0 100 a_{min} a_{max} b_{min} b_{max}] where a_{min}, a_{max}, b_{min}, and b_{max} correspond to the values in the **Range** array of the image's color space
ICCBased	Same as the value of **Range** in the ICC profile of the image's color space
Indexed	[0 N], where $N = 2^n - 1$
Pattern	(Not permitted with images)
Separation	[0.0 1.0]
DeviceN	[0.0 1.0 0.0 1.0 … 0.0 1.0] (one pair of elements for each color component)

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 image's color space is **DeviceGray** and the **Decode** array is [1.0 0.0], 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 represent those samples in PDF 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 [0.00000 4.04762], which maps input values from 0 to 63 into the range 0.0 to 1.0 (4.04762 being approximately equal to 255 ÷ 63). 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** enables it.

Note: *The interpolation algorithm is implementation-dependent and is not specified by PDF. Image interpolation may not always be performed for some classes of images or on some output devices.*

Alternate Images

Alternate images (PDF 1.3) provide a straightforward and backward-compatible way to include multiple versions of an image in a PDF file for different purposes. These variant representations of the image may differ, for example, in resolution or in color space. The primary goal is to reduce the need to maintain separate versions of a PDF document for low-resolution on-screen viewing and high-resolution printing.

In PDF 1.3, a *base image* (that is, the image XObject referred to in a resource dictionary) can contain an **Alternates** entry. The value of this entry is an array of *alternate image dictionaries* specifying variant representations of the base image. Each alternate image dictionary contains an image XObject for one variant and specifies its properties. Table 4.38 shows the contents of an alternate image dictionary.

TABLE 4.38 Entries in an alternate image dictionary

KEY	TYPE	VALUE
Image	stream	*(Required)* The image XObject for the alternate image.
DefaultForPrinting	boolean	*(Optional)* A flag indicating whether this alternate image is the default version to be used for printing. At most one alternate for a given base image may be so designated. If no alternate has this entry set to **true**, the base image itself is used for printing.
OC	dictionary	*(Optional; PDF 1.5)* An optional content group (see Section 4.10.1, "Optional Content Groups") or optional content membership dictionary (see "Optional Content Membership Dictionaries" on page 329"), used to facilitate the selection of which alternate image to use.

Example 4.25 shows an image with a single alternate. The base image is a grayscale image, and the alternate is a high-resolution *RGB* image stored on a Web server.

Example 4.25

```
10 0 obj                                    % Image XObject
   << /Type /XObject
      /Subtype /Image
      /Width 100
      /Height 200
      /ColorSpace /DeviceGray
      /BitsPerComponent 8
      /Alternates 15 0 R
      /Length 2167
      /Filter /DCTDecode
   >>
stream
… Image data …
endstream
endobj
```

```
15 0 obj                                    % Alternate images array
   [ << /Image 16 0 R
         /DefaultForPrinting true
      >>
   ]
endobj

16 0 obj                                    % Alternate image
   << /Type /XObject
        /Subtype /Image
        /Width 1000
        /Height 2000
        /ColorSpace /DeviceRGB
        /BitsPerComponent 8
        /Length 0                           % This is an external stream
        /F << /FS /URL
                 /F (http://www.myserver.mycorp.com/images/exttest.jpg)
             >>
        /FFilter /DCTDecode
   >>
stream
endstream
endobj
```

In PDF 1.5, optional content (see Section 4.10) can be used to facilitate selection between alternate images. If an image XObject contains both an **Alternates** entry and an **OC** entry, the choice of which image to use is determined as follows:

1. If the image's **OC** entry specifies that the base image is visible, that image is displayed.
2. Otherwise, the list of alternates specified by the **Alternates** entry is examined, and the first alternate containing an **OC** entry specifying that its content should be visible is shown. (Alternate images that have no **OC** entry are not shown.)

4.8.5 Masked Images

Ordinarily, in the opaque imaging model, images mark all areas they occupy on the page as if with opaque paint. 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 PDF features are available for achieving such masking effects (see implementation note 46 in Appendix H):

- The **ImageMask** entry in the image dictionary, available in all versions of PDF, specifies that the image data is to be used as a *stencil mask* for painting in the current color.
- The **Mask** entry in the image dictionary *(PDF 1.3)* may specify a separate image XObject to be used as an *explicit mask* specifying which areas of the image to paint and which to mask out.
- Alternatively, the **Mask** entry *(PDF 1.3)* may specify a range of colors to be masked out wherever they occur within the image; this technique is known as *color key masking.*

Note: *Although the* ***Mask*** *entry is a PDF 1.3 feature, its effects are commonly simulated in earlier versions of PDF by defining a clipping path enclosing only those of an image's samples that are to be painted. However, implementation limits can cause 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 an explicit mask in PDF 1.2 is to define the image being clipped as a pattern, make it the current color, and then paint the explicit mask as an image whose* ***ImageMask*** *entry is* ***true****. In any case, the PDF 1.3 features allow masked images to be placed on the page without regard to the complexity of the clipping path.*

In the transparent imaging model, a fourth type of masking effect, *soft masking*, is available via the **SMask** entry *(PDF 1.4)* or the **SMaskInData** entry *(PDF 1.5)* in the image dictionary; see Section 7.5.4, "Specifying Soft Masks," for further discussion.

Stencil Masking

An *image mask* (an image XObject whose **ImageMask** entry is **true**) is a monochrome image, in which each sample is specified by a single bit. However, instead of being painted in opaque black and white, the image mask is treated 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, which allows the paint to reach the page in some places and masks it out in others.

An image mask differs from an ordinary image in the following significant ways:

- The image dictionary does not contain a **ColorSpace** entry, because sample values represent masking properties (1 bit per sample) rather than colors.
- The value of the **BitsPerComponent** entry must be 1.
- The **Decode** entry determines how the source samples are to be interpreted. If the **Decode** array is `[0 1]` (the default for an image mask), a sample value of 0 marks the page with the current color, while a 1 leaves the previous contents unchanged; if the **Decode** array is `[1 0]`, these meanings are reversed.

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 5.5.4, "Type 3 Fonts," an image mask, rather than an image, should almost always be used to paint glyph bitmaps.

***Note:** If image interpolation (see "Image Interpolation" on page 310) is requested during stencil masking, the effect is to smooth the edges of the mask, not to interpolate the painted color values. This can minimize the "jaggy" appearance of a low-resolution stencil mask.*

Explicit Masking

In PDF 1.3, the **Mask** entry in an image dictionary may be an image mask, as described above under "Stencil Masking," which serves as an *explicit mask* for the primary (base) image. The base image and the image mask need not have the same resolution (**Width** and **Height** values), but since all images are defined on the unit square in user space, their boundaries on the page will coincide; that is, they will overlay each other. The image mask 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 base image; masked areas are not.

Color Key Masking

In PDF 1.3, the **Mask** entry in an image dictionary may alternatively be an array specifying a range of colors to be masked out. Samples in the image that 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*.

For color key masking, the value of the **Mask** entry is an array of $2 \times n$ integers, $[min_1\ max_1\ \ldots\ min_n\ max_n]$, where n is the number of color components in the image's color space. Each integer must be in the range 0 to $2^{\textbf{BitsPerComponent}} - 1$, representing color values *before* decoding with the **Decode** array. An image sample is masked (not painted) if all of its color components before decoding, $c_1 \ldots c_n$, fall within the specified ranges (that is, if $min_i \le c_i \le max_i$ for all $1 \le i \le n$).

Note: *When color key masking is specified, the use of a* ***DCTDecode*** *filter for the stream is not recommended.* ***DCTDecode*** *is a lossy filter, meaning that the 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.*

4.8.6 Inline Images

As an alternative to the image XObjects described in Section 4.8.4, "Image Dictionaries," a sampled image may be specified in the form of an *inline image*. This type of image is defined directly within the content stream in which it will be painted, rather than as a separate object. Because the inline format gives the viewer application less flexibility in managing the image data, it should be used only for small images (4 KB or less).

An inline image object is delimited in the content stream by the operators **BI** (begin image), **ID** (image data), and **EI** (end image); these operators are summarized in Table 4.39. **BI** and **ID** bracket a series of key-value pairs specifying the characteristics of the image, such as its dimensions and color space; the image data follows between the **ID** and **EI** operators. The format is thus analogous to that of a stream object such as an image XObject:

BI
…Key-value pairs…
ID

. . . Image data . . .
EI

TABLE 4.39 Inline image operators

OPERANDS	OPERATOR	DESCRIPTION
—	**BI**	Begin an inline image object.
—	**ID**	Begin the image data for an inline image object.
—	**EI**	End an inline image object.

Inline image objects may not be nested; that is, two **BI** operators may not appear without an intervening **EI** to close the first object. Similarly, an **ID** operator may appear only between a **BI** and its balancing **EI**. Unless the image uses **ASCIIHexDecode** or **ASCII85Decode** as one of its filters, the **ID** operator should be followed by a single white-space character; the next character after that is interpreted as the first byte of image data.

The key-value pairs appearing between the **BI** and **ID** operators are analogous to those in the dictionary portion of an image XObject (though the syntax is different). Table 4.40 shows the entries that are valid for an inline image, all of which have the same meanings as in a stream dictionary (Table 3.4 on page 38) or an image dictionary (Table 4.36 on page 303). Entries other than those listed will be ignored; in particular, the **Type**, **Subtype**, and **Length** entries normally found in a stream or image dictionary are unnecessary. For convenience, the abbreviations shown in the table may be used in place of the fully spelled-out keys; Table 4.41 shows additional abbreviations that can be used for the names of color spaces and filters. Note, however, that these abbreviations are valid only in inline images; they may *not* be used in image XObjects. Also note that **JBIG2Decode** and **JPXDecode** are not listed in Table 4.41, because those filters can be applied only to image XObjects.

TABLE 4.40 Entries in an inline image object

FULL NAME	ABBREVIATION
BitsPerComponent	**BPC**
ColorSpace	**CS**
Decode	**D**

FULL NAME	ABBREVIATION
DecodeParms	**DP**
Filter	**F**
Height	**H**
ImageMask	**IM**
Intent *(PDF 1.1)*	No abbreviation
Interpolate	**I** (uppercase I)
Width	**W**

TABLE 4.41 Additional abbreviations in an inline image object

FULL NAME	ABBREVIATION
DeviceGray	**G**
DeviceRGB	**RGB**
DeviceCMYK	**CMYK**
Indexed	**I** (uppercase I)
ASCIIHexDecode	**AHx**
ASCII85Decode	**A85**
LZWDecode	**LZW**
FlateDecode *(PDF 1.2)*	**Fl** (uppercase F, lowercase L)
RunLengthDecode	**RL**
CCITTFaxDecode	**CCF**
DCTDecode	**DCT**

The color space specified by the **ColorSpace** (or **CS**) entry may be any of the standard device color spaces (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**). It may not be a CIE-based color space or a special color space, with the exception of a limited form of **Indexed** color space whose base color space is a device space and whose color table is specified by a string (see "Indexed Color Spaces" on page 232). Beginning with PDF 1.2, the value of the **ColorSpace** entry may also be the name of a color space in the **ColorSpace** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"); in this case, the name may designate any color space that can be used with an image XObject.

Note: *The names* ***DeviceGray****,* ***DeviceRGB****, and* ***DeviceCMYK*** *(as well as their abbreviations* ***G****,* ***RGB****, and* ***CMYK****) always identify the corresponding color spaces directly; they never refer to resources in the* ***ColorSpace*** *subdictionary.*

The image data in an inline image may be encoded using any of the standard PDF filters. The bytes between the **ID** and **EI** operators are treated much the same as a stream object's data (see Section 3.2.7, "Stream Objects"), even though they do not follow the standard stream syntax. (This is an exception to the usual rule that the data in a content stream is interpreted according to the standard PDF syntax for objects.)

Example 4.26 shows an inline image 17 samples wide by 17 high with 8 bits per component in the **DeviceRGB** color space. The image has been encoded using LZW and ASCII base-85 encoding. The **cm** operator is used to scale it to a width and height of 17 units in user space and position it at coordinates (298, 388). The **q** and **Q** operators encapsulate the **cm** operation to limit its effect to resizing the image.

Example 4.26

```
q                                       % Save graphics state
17 0 0 17 298 388 cm                    % Scale and translate coordinate space
BI                                      % Begin inline image object
    /W 17                               % Width in samples
    /H 17                               % Height in samples
    /CS /RGB                            % Color space
    /BPC 8                              % Bits per component
    /F [/A85 /LZW]                      % Filters
ID                                      % Begin image data
J1/gKA>.]AN&J?]-<HW]aRVcg*bb.\eKAdVV%/PcZ
…Omitted data…
R.s(4KE3&d&7hb*7[%Ct2HCqC~>
EI                                      % End inline image object
Q                                       % Restore graphics state
```

4.9 Form XObjects

A *form XObject* is a self-contained description of any sequence of graphics objects (including path objects, text objects, and sampled images), defined as a PDF content stream. It may be painted multiple times—either on several pages or at several locations on the same page—and will produce the same results each time, subject only to the graphics state at the time it is invoked. Not only is this shared definition economical to represent in the PDF file, but under suitable circumstances the PDF viewer application can optimize execution by caching the results of rendering the form XObject for repeated reuse.

Note: *The term* form *also refers to a completely different kind of object, an* interactive form *(sometimes called an* AcroForm*), discussed in Section 8.6, "Interactive Forms." Whereas the form XObjects described in this section correspond to the notion of forms in the PostScript language, interactive forms are the PDF equivalent of the familiar paper instrument. Any unqualified use of the word* form *is understood to refer to an interactive form; the type of form described here is always referred to explicitly as a* form XObject.

Form XObjects have various uses:

- As its name suggests, a form XObject 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 XObject and then paint the variable information on top of it.
- Any graphical element that is to be used repeatedly, such as a company logo or a standard component in the output from a computer-aided design system, can be defined as a form XObject.
- Certain document elements that are not part of a page's contents, such as annotation appearances (see Section 8.4.4, "Appearance Streams"), are represented as form XObjects.
- A specialized type of form XObject, called a *group XObject (PDF 1.4)*, can be used to group graphical elements together as a unit for various purposes (see Section 4.9.2, "Group XObjects"). In particular, group XObjects are used to define transparency groups and soft masks for use in the transparent imaging model (see "Soft-Mask Dictionaries" on page 510 and Section 7.5.5, "Transparency Group XObjects").

- Another specialized type of form XObject, a *reference XObject (PDF 1.4)*, can be used to import content from one PDF document into another (see Section 4.9.3, "Reference XObjects").

The use of form XObjects requires two steps:

1. *Define the appearance of the form XObject*. A form XObject is a PDF content stream. The dictionary portion of the stream (called the *form dictionary*) contains descriptive information about the form XObject, while the body of the stream describes the graphics objects that produce its appearance. The contents of the form dictionary are described in Section 4.9.1, "Form Dictionaries."
2. *Paint the form XObject*. The **Do** operator (see Section 4.7, "External Objects") paints a form XObject whose name is supplied as an operand. (The name is defined in the **XObject** subdictionary of the current resource dictionary.) Before invoking this operator, the content stream in which it appears 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 XObject in user space.

Each form XObject is defined in its own coordinate system, called *form space*. The **BBox** entry in the form dictionary is expressed in form space, as are any coordinates used in the form XObject's content stream, such as path coordinates. The **Matrix** entry in the form dictionary specifies the mapping from form space to the current user space; each time the form XObject is painted by the **Do** operator, this matrix is concatenated with the current transformation matrix to define the mapping from form space to device space. (This differs from the **Matrix** entry in a pattern dictionary, which maps pattern space to the *initial* user space of the content stream in which the pattern is used.)

When the **Do** operator is applied to a form XObject, it does the following:

1. Saves the current graphics state, as if by invoking the **q** operator (see Section 4.3.3, "Graphics State Operators")
2. Concatenates the matrix from the form dictionary's **Matrix** entry with the current transformation matrix (CTM)
3. Clips according to the form dictionary's **BBox** entry
4. Paints the graphics objects specified in the form's content stream

5. Restores the saved graphics state, as if by invoking the **Q** operator (see Section 4.3.3, "Graphics State Operators")

Except as described above, the initial graphics state for the form is inherited from the graphics state that is in effect at the time **Do** is invoked.

4.9.1 Form Dictionaries

Every form XObject has a *form type*, which determines the format and meaning of the entries in its form dictionary. At the time of publication, only one form type, type 1, has been defined. Form XObject dictionaries may contain the entries shown in Table 4.42, in addition to the usual entries common to all streams (see Table 3.4 on page 38).

TABLE 4.42 Additional entries specific to a type 1 form dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **XObject** for a form XObject.
Subtype	name	*(Required)* The type of XObject that this dictionary describes; must be **Form** for a form XObject.
FormType	integer	*(Optional)* A code identifying the type of form XObject that this dictionary describes. The only valid value defined at the time of publication is 1. Default value: 1.
Name	name	*(Required in PDF 1.0; optional otherwise)* The name by which this form XObject is referenced in the **XObject** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries"). ***Note:** This entry is obsolescent and its use is no longer recommended. (See implementation note 47 in Appendix H.)*
LastModified	date	*(Required if **PieceInfo** is present; optional otherwise; PDF 1.3)* The date and time (see Section 3.8.3, "Dates") when the form XObject's contents were most recently modified. If a page-piece dictionary (**PieceInfo**) is present, the modification date is used to ascertain which of the application data dictionaries it contains correspond to the current content of the form (see Section 10.4, "Page-Piece Dictionaries").

KEY	TYPE	VALUE
BBox	rectangle	*(Required)* An array of four numbers in the form coordinate system (see above), giving the coordinates of the left, bottom, right, and top edges, respectively, of the form XObject's bounding box. These boundaries are used to clip the form XObject and to determine its size for caching.
Matrix	array	*(Optional)* An array of six numbers specifying the *form matrix*, which maps form space into user space (see Section 4.2.3, "Transformation Matrices"). Default value: the identity matrix `[1 0 0 1 0 0]`.
Resources	dictionary	*(Optional but strongly recommended; PDF 1.2)* A dictionary specifying any resources (such as fonts and images) required by the form XObject (see Section 3.7, "Content Streams and Resources"). In PDF 1.1 and earlier, all named resources used in the form XObject must be included in the resource dictionary of each page object on which the form XObject appears, whether or not they also appear in the resource dictionary of the form XObject itself. It can be useful to specify these resources in the form XObject's own resource dictionary as well, in order to determine which resources are used inside the form XObject. If a resource is included in both dictionaries, it should have the same name in both locations. In PDF 1.2 and later versions, form XObjects can be independent of the content streams in which they appear, and this is strongly recommended although not required. In an independent form XObject, the resource dictionary of the form XObject is required and contains all named resources used by the form XObject. These resources are not "promoted" to the outer content stream's resource dictionary, although that stream's resource dictionary will refer to the form XObject itself.
Group	dictionary	*(Optional; PDF 1.4)* A *group attributes dictionary* indicating that the contents of the form XObject are to be treated as a group and specifying the attributes of that group (see Section 4.9.2, "Group XObjects"). ***Note:*** *If a **Ref** entry (see below) is present, the group attributes also apply to the external page imported by that entry. This allows such an imported page to be treated as a group without further modification.*
Ref	dictionary	*(Optional; PDF 1.4)* A reference dictionary identifying a page to be imported from another PDF file, and for which the form XObject serves as a proxy (see Section 4.9.3, "Reference XObjects").
Metadata	stream	*(Optional; PDF 1.4)* A *metadata stream* containing metadata for the form XObject (see Section 10.2.2, "Metadata Streams").

KEY	TYPE	VALUE
PieceInfo	dictionary	*(Optional; PDF 1.3)* A page-piece dictionary associated with the form XObject (see Section 10.4, "Page-Piece Dictionaries").
StructParent	integer	*(Required if the form XObject is a structural content item; PDF 1.3)* The integer key of the form XObject's entry in the structural parent tree (see "Finding Structure Elements from Content Items" on page 739).
StructParents	integer	*(Required if the form XObject contains marked-content sequences that are structural content items; PDF 1.3)* The integer key of the form XObject's entry in the structural parent tree (see "Finding Structure Elements from Content Items" on page 739). ***Note:*** *At most one of the entries* ***StructParent*** *or* ***StructParents*** *may be present. A form XObject can be either a content item in its entirety or a container for marked-content sequences that are content items, but not both.*
OPI	dictionary	*(Optional; PDF 1.2)* An OPI version dictionary for the form XObject (see Section 10.10.6, "Open Prepress Interface (OPI)").
OC	dictionary	*(Optional; PDF 1.5)* An optional content group or optional content membership dictionary (see Section 4.10, "Optional Content"), specifying the optional content properties for the form XObject. Before the form is processed, its visibility is determined is processed based on this entry; if it is determined to be invisible, the entire form is skipped, as if there were no **Do** operator to invoke it.

Example 4.27 shows a simple form XObject that paints a filled square 1000 units on each side.

Example 4.27

```
6 0 obj                                   % Form XObject
   << /Type /XObject
       /Subtype /Form
       /FormType 1
       /BBox [0 0 1000 1000]
       /Matrix [1 0 0 1 0 0]
       /Resources << /ProcSet [/PDF] >>
       /Length 58
   >>
```

```
stream
    0 0 m
    0 1000 l
    1000 1000 l
    1000 0 l
    f
endstream
endobj
```

4.9.2 Group XObjects

A *group XObject (PDF 1.4)* is a special type of form XObject that can be used to group graphical elements together as a unit for various purposes. It is distinguished by the presence of the optional **Group** entry in the form dictionary (see Section 4.9.1, "Form Dictionaries"). The value of this entry is a subsidiary *group attributes dictionary* describing the properties of the group.

As shown in Table 4.43, every group XObject has a *group subtype* (specified by the **S** entry in the group attributes dictionary) that determines the format and meaning of the dictionary's remaining entries. Only one such subtype is currently defined, a *transparency group XObject* (subtype **Transparency**) representing a transparency group for use in the transparent imaging model (see Section 7.3, "Transparency Groups"). The remaining contents of this type of dictionary are described in Section 7.5.5, "Transparency Group XObjects."

TABLE 4.43 Entries common to all group attributes dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Group** for a group attributes dictionary.
S	name	*(Required)* The *group subtype*, which identifies the type of group whose attributes this dictionary describes and determines the format and meaning of the dictionary's remaining entries. The only group subtype defined in PDF 1.4 is **Transparency**; see Section 7.5.5, "Transparency Group XObjects," for the remaining contents of this type of dictionary. Other group subtypes may be added in the future.

4.9.3 Reference XObjects

Reference XObjects (PDF 1.4) enable one PDF document to import content from another. The document in which the reference occurs is called the *containing document*; the one whose content is being imported is the *target document*. The target document may reside in a file external to the containing document or may be included within it as an embedded file stream (see Section 3.10.3, "Embedded File Streams").

The reference XObject in the containing document is a form XObject containing the optional **Ref** entry in its form dictionary, as described below. This form XObject serves as a *proxy* that can be displayed or printed in place of the imported content. The proxy might consist of a low-resolution image of the imported content, a piece of descriptive text referring to it, a gray box to be displayed in its place, or any other similar placeholder. Viewer applications that do not recognize the **Ref** entry will simply display or print the proxy as an ordinary form XObject (see implementation note 48 in Appendix H); those that do implement reference XObjects can use the proxy in place of the imported content if the latter is unavailable. A viewer application may also provide a user interface to allow editing and updating of imported content links.

The imported content consists of a single, complete PDF page in the target document. It is designated by a *reference dictionary*, which in turn is the value of the **Ref** entry in the reference XObject's form dictionary (see Section 4.9.1, "Form Dictionaries"). It is the presence of the **Ref** entry that distinguishes reference XObjects from other types of form XObject. Table 4.44 shows the contents of the reference dictionary.

TABLE 4.44 Entries in a reference dictionary

KEY	TYPE	VALUE
F	file specification	*(Required)* The file containing the target document.
Page	integer or text string	*(Required)* A page index or page label (see Section 8.3.1, "Page Labels") identifying the page of the target document containing the content to be imported. Note that the reference is a weak one and can be inadvertently invalidated if the referenced page is changed or replaced in the target document after the reference is created.

KEY	TYPE	VALUE
ID	array	*(Optional)* An array of two strings constituting a file identifier (see Section 10.3, "File Identifiers") for the file containing the target document. The use of this entry improves a viewer application's chances of finding the intended file and allows it to warn the user if the file has changed since the reference was created.

When the imported content replaces the proxy, it is transformed according to the proxy object's transformation matrix and clipped to the boundaries of its bounding box, as specified by the **Matrix** and **BBox** entries in the proxy's form dictionary (see Section 4.9.1, "Form Dictionaries"). The combination of the proxy object's matrix and bounding box thus implicitly defines the bounding box of the imported page. This bounding box will typically coincide with the imported page's crop box or art box (see Section 10.10.1, "Page Boundaries"), but it is not required to correspond to any of the defined page boundaries. If the proxy object's form dictionary contains a **Group** entry, the specified group attributes apply to the imported page as well; this allows the imported page to be treated as a group without further modification.

Printing Reference XObjects

When printing a page containing reference XObjects, a viewer application may emit any of the following, depending on the capabilities of the viewer application, the user's preferences, and the nature of the print job:

- The imported content designated by the reference XObject
- The reference XObject itself, as a proxy for the imported content
- An OPI proxy or substitute image taken from the reference XObject's OPI dictionary, if any (see Section 10.10.6, "Open Prepress Interface (OPI)")

The imported content or the reference XObject itself may also be emitted in place of an OPI proxy when generating OPI comments in a PostScript output stream.

Special Considerations

Certain special considerations arise when reference XObjects interact with other PDF features:

- When the page imported by a reference XObject contains annotations (see Section 8.4, "Annotations"), all annotations that contain a printable, unhidden, visible appearance stream (Section 8.4.4, "Appearance Streams") must be included in the rendering of the imported page. If the proxy is a snapshot image of the imported page, it must also include the annotation appearances. These appearances must therefore be converted into part of the proxy's content stream, either as subsidiary form XObjects or by "flattening" them directly into the content stream.
- Logical structure information associated with a page (see Section 10.6, "Logical Structure") should normally be ignored when importing the page into another document with a reference XObject. In a target document with multiple pages, structure elements occurring on the imported page will typically be part of a larger structure pertaining to the document as a whole; such elements cannot meaningfully be incorporated into the structure of the containing document. In a one-page target document or one made up of independent, structurally unrelated pages, the logical structure for the imported page may be wholly self-contained; in this case, it may be possible to incorporate this structure information into that of the containing document. However, PDF provides no mechanism for the logical structure hierarchy of one document to refer indirectly to that of another.

4.10 Optional Content

Optional content (PDF 1.5) refers to sections of content in a PDF document that can be selectively viewed or hidden by document authors or consumers. This capability is useful in items such as CAD drawings, layered artwork, maps, and multi-language documents.

The following sections describe the PDF structures used to implement optional content:

- Section 4.10.1, "Optional Content Groups," describes the primary structures used to control the visibility of content.

- Section 4.10.2, "Making Graphical Content Optional," describes how individual pieces of content in a document may declare themselves as belonging to one or more optional content groups.
- Section 4.10.3, "Configuring Optional Content," describes how the states of optional content groups are set.

4.10.1 Optional Content Groups

An optional content group is a dictionary representing a collection of graphics that can be made visible or invisible dynamically by users of viewer applications. The graphics belonging to such a group can reside anywhere in the document: they need not be consecutive in drawing order, nor even belong to the same content stream. Table 4.45 shows the entries in an optional content group dictionary.

TABLE 4.45 Entries in an optional content group dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **OCG** for an optional content group dictionary.
Name	text string	*(Required)* The name of the optional content group, suitable for presentation in a viewer application's user interface.
Intent	name or array	*(Optional)* May be one of the following: • A single intent name. PDF 1.5 defines two names, **View** and **Design**; future versions may define others. These names indicate the intended use of the graphics in the group; a processing application may choose to use only groups that are of a specific intent and ignore others. • An array containing any combination of names. Default value: **View**. See "Intent" on page 331 for more information.
Usage	dictionary	*(Optional)* A *usage dictionary* describing the nature of the content controlled by the group; it may be used by features that automatically control the state of the group based on outside factors. See "Usage and Usage Application Dictionaries" on page 341 for more information.

In its simplest form, each dictionary contains a **Type** entry and a **Name** for presentation in a user interface. It may also have an **Intent** entry that describes its intended use (see "Intent" on page 331) and a **Usage** entry that describes the nature of its content (see "Usage and Usage Application Dictionaries" on page 341).

Individual content elements in a document specify the optional content group or groups that affect their visibility (see Section 4.10.2, "Making Graphical Content Optional"). Any content whose visibility can be affected by a given optional content group is said to belong to that group.

A group is assigned a state, which is either **ON** and **OFF**. States are not themselves part of the PDF document, but can be set programmatically or through the viewer user interface, to change the visibility of content. When a document is first opened, the groups' states are initialized based on the document's default configuration dictionary (see "Optional Content Configuration Dictionaries" on page 338).

In the typical case, content belonging to a group is visible when the group is **ON** and invisible when it is **OFF**. In more complex cases, content can belong to multiple groups, which may have conflicting states. These cases are described by the use of optional content membership dictionaries, described in the next section.

Optional Content Membership Dictionaries

As mentioned above, content typically belongs to a single optional content group, and is visible when the group is **ON** and invisible when it is **OFF**. To express more complex visibility policies, content should not declare itself to belong to an optional content group directly, but rather to an *optional content membership dictionary*, whose entries are shown in Table 4.46. (Section 4.10.2 describes how content declares its membership in a group or membership dictionary).

TABLE 4.46 Entries in an optional content membership dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **OCMD** for an optional content membership dictionary.

KEY	TYPE	VALUE
OCGs	dictionary or array	*(Optional)* A dictionary or array of dictionaries specifying the optional content groups whose states determine the visibility of content controlled by this membership dictionary. ***Note:*** *Null values or references to deleted objects are ignored. If this entry is not present, is an empty array, or contains references only to null or deleted objects, the membership dictionary has no effect on the visibility of any content.*
P	name	*(Optional)* A name specifying the *visibility policy* for content belonging to this membership dictionary. Possible values are: • **AllOn**: visible only if all of the entries in **OCGs** are **ON**. • **AnyOn**: visible if any of the entries in **OCGs** are **ON**. • **AnyOff**: visible if any of the entries in **OCGs** are **OFF**. • **AllOff**: visible only if all of the entries in **OCGs** are **OFF**. Default value: **AnyOn**

The following are examples of cases that can be dealt with through the use of a membership dictionary:

- Some content may choose to be *invisible* when a group is **ON** and *visible* when it is **OFF**. In this case, the content would belong to a membership dictionary whose **OCGs** entry consists of a single optional content group and whose **P** entry is **AnyOff** or **AllOff**.

  Note: *It is legal to have an* ***OCGs*** *entry consisting of a single group and a* ***P*** *entry that is* ***AnyOn*** *or* ***AllOn****; however, in this case it is preferable to use an optional content group directly, because it uses fewer objects.*

- Some content may belong to more than one group, and must specify its policy when the groups are in conflicting states. In this case, the content would belong to a membership dictionary whose **OCGs** entry consists of an array of optional content groups, and whose **P** entry specifies the visibility policy, as illustrated in Example 4.28 below.

Example 4.28

```
<< /Type /OCMD                    % Content belonging to this optional content
                                  % membership dictionary is controlled by the states
    /OCGs [12 0 R 13 0 R 14 0 R]  % of three optional content groups.
    /P /AllOn                        % Content is visible only if the state of all three groups
>>                                   % is ON; otherwise it's hidden.
```

Intent

The **Intent** entry in Table 4.45 provides a way to distinguish between different intended uses of optional content. For example, many document design applications, such as CAD packages, offer layering features for collecting groups of graphics together and selectively hiding or viewing them for the convenience of the author. However, this layering may be different (at a finer granularity, for example) than would be useful to consumers of the document. Therefore, it is possible to specify different intents for optional content groups within a single document. A given application may decide to use only groups that are of a specific intent.

PDF 1.5 defines two intents: **Design**, which is intended to represent a document designer's structural organization of artwork, and **View**, which is intended for interactive use by document consumers. More intents may be added in future PDF versions; for compatibility with future versions, PDF consumers should allow unrecognized **Intent** values.

Configuration dictionaries (see "Optional Content Configuration Dictionaries" on page 338) also contain an **Intent** entry. If one or more of a group's intents is found in the current configuration's set of intents, the group is used in determining visibility. If there is no match, the group has no effect on visibility.

***Note:** If the configuration's **Intent** is an empty array, no groups are used in determining visibility; therefore all content is considered visible.*

4.10.2 Making Graphical Content Optional

Graphical content in a PDF file can be made optional by specifying membership in an optional content group or optional content membership dictionary. There are two primary mechanisms for doing this:

- Sections of content streams delimited by marked-content operators can be made optional, as described in "Optional Content in Content Streams," below.

- Form and image XObjects and annotations can be made optional in their entirety by means of a new dictionary entry, as described in "Optional Content in XObjects and Annotations" on page 336.

When a piece of optional content in a PDF file is determined to be hidden, the following occurs:

- The content is not drawn.
- Graphics state operations, such as setting the color, transformation matrix and clipping, are still applied. In addition, graphics state side effects that arise from drawing operators are applied; in particular, the current text position is updated even for text wrapped in optional content. In other words, graphics state parameters that persist past the end of a marked-content section must be the same whether the optional content is visible or not; for example, hiding a section of optional content does not change the color of objects that do not belong to the same optional content group.

  Note: *This rule also applies to operators that set state that is not strictly graphics state; for example,* **BX** *and* **EX**.
- Objects such as form XObjects and annotations that are made optional may be skipped entirely, since their contents are encapsulated such that no changes to the graphics state (or other state) persist beyond the processing of their content stream.

Other features in PDF consuming applications, such as searching and editing, may be affected by the ability to selectively show or hide content. Features must choose whether to use the document's current state of optional content groups (and, correspondingly, the document's visible graphics) or to supply their own states of optional content groups to control the graphics they process. For example, tools to select and move annotations should honor the current on-screen visibility of annotations when performing cursor tracking and mouse-click processing. On the other hand, a full text search engine may need to process all content in a document, regardless of its current visibility on-screen. Export filters might choose the current on-screen visibility, the full content, or present the user with a selection of OCGs to control visibility.

Note: *All the new optional content-related PDF structures will be unknown to, and hence ignored by, PDF 1.4 and earlier viewers. Therefore, they will draw and otherwise process all content in the document.*

Optional Content in Content Streams

Sections of content in a content stream (including a page's **Contents** stream, a form or pattern's content stream, glyph descriptions a Type 3 font as specified by its **CharProcs** entry, or an annotation's appearance) can be made optional by enclosing them between the marked-content operators **BDC** and **EMC** (see Section 10.5, "Marked Content"), with a marked-content tag of OC. In addition, a **DP** marked-content operator can be placed in a page's content stream to force a reference to an optional content group or groups on the page, even when the page has no current content in that layer.

The property list associated with the marked content specifies either an optional content group or optional content membership dictionary to which the content belongs. Because a group must be an indirect object and a membership dictionary contains references to indirect objects, the property list must be a named resource listed in the **Properties** subdictionary of the current resource dictionary (see Section 10.5.1, "Property Lists"), as shown in Examples 4.29 and 4.30.

Note: *Although the marked-content tag must be OC, this does not preclude other applications of marked content from using OC as a tag. The marked content is considered to be for optional content only if the tag is OC and the dictionary operand is a valid optional content group or optional content membership dictionary.*

To avoid conflict with other features that used marked content (such as logical structure; see Section 10.6, "Logical Structure"), the following strategy is recommended:

- Where content is to be tagged with optional content markers as well as other markers, the optional content markers should be nested inside the other marked content.
- Where optional content and the other markers would overlap but there is not strict containment, the optional content should be broken up into two or more **BDC**/**EMC** sections, nesting the optional content sections inside the others as necessary. Breaking up optional content spans does not damage the nature of the visibility of the content, whereas the same guarantee cannot be made for all other uses of marked content.

Note: *Any marked content tagged for optional content that is nested inside other marked content tagged for optional content is visible only if all the levels indicate* *visibility. In other words, if the settings that apply to the outer level indicate that the content should be hidden, the inner level is hidden regardless of its settings.*

In the following example, the state of the "Show Greeting" optional content group directly controls the visibility of the text string "Hello" on the page. When the group is **ON**, the text is visible; when the group is **OFF**, the text is hidden.

Example 4.29

```
% Within a content stream
/OC /oc1 BDC                        % Optional content follows
    BT
        /F1 1 Tf
        12 0 0 12 100 600 Tm
        (Hello) Tj
    ET
EMC                                 % End of optional content

<<                                  % In the resources dictionary
    /Properties << /oc1 5 0 R >>    % This dictionary maps the name oc1 to an
...                                 % optional content group (object 5)
>>

5 0 obj                             % The OCG controlling the visibility
<<                                  % of the text.
    /Type /OCG
    /Name (Show Greeting)
>>
endobj
```

The example above shows one piece of content associated with one optional content group. Other possibilities include:

- More than one section of content may refer to the same group or membership dictionary, in which case the visibility of both sections is always the same.
- Equivalently, different sections can have separate membership dictionaries with the same **OCGs** and **P** entries. This also results in the two sections having identical visibility behavior, although it is less space-efficient.
- Two sections of content can belong to membership dictionaries that refer to the same group(s), but with different **P** settings. For example, if one section has no **P** entry, and the other has a **P** entry of **AllOff**, the visibility of the two sections of

content will be opposite. That is, the first section will be visible when the second is hidden, and vice versa.

The following example demonstrates both the direct use of optional content groups and the indirect use of groups through a membership dictionary. The content (a black rectangle frame) is drawn if either of the images controlled by the groups named “Image A” or “Image B” is shown. If both groups are hidden, then the rectangle frame is hidden.

Example 4.30

```
% Within a content stream
...
/OC /OC2 BDC                  % Draws a black rectangle frame
    0 g
    4 w
    100 100 412 592 re s
EMC
/OC /OC3 BDC                  % Draws an image XObject
    q
    412 0 0 592 100 100 cm
    /Im3 Do
    Q
EMC
/OC /OC4 BDC                  % Draws an image XObject
    q
    412 0 0 592 100 100 cm
    /Im4 Do
    Q
EMC
...

<<                            % The resource dictionary
    /Properties << /OC2 20 0 R /OC3 30 0 R /OC4 40 0 R >>
    /XObject << /Im3 50 0 R /Im4 /60 0 R >>
>>

20 0 obj
<<                            % Optional content membership dictionary
    /Type /OCMD
    /OCGs [30 0 R 40 0 R]
    /P /AnyOn
>>
```

```
endobj
30 0 obj                        % Optional content group "Image A"
<<
    /Type /OCG
    /Name (Image A)
>>
endobj
40 0 obj                        % Optional content group "Image B"
<<
    /Type /OCG
    /Name (Image B)
>>
endobj
```

Optional Content in XObjects and Annotations

In addition to marked content within content streams, form XObjects and image XObjects (see Section 4.7, "External Objects") and annotations (see Section 8.4, "Annotations") may contain an **OC** entry, which is an optional content group or an optional content membership dictionary.

A form or image XObject's visibility is determined by the state of the group, or those of the groups referenced by the membership dictionary in conjunction with its **P** entry, along with the current visibility state in the context in which the XObject is invoked (that is, whether objects are visible in the contents stream at the place where the **Do** operation occurred).

Annotations have various flags controlling on-screen and print visibility (see Section 8.4.2, "Annotation Flags"). If an annotation contains an **OC** entry, it is visible for screen or print only if the flags have the appropriate settings and the group or membership dictionary indicates it is visible.

4.10.3 Configuring Optional Content

A PDF document containing optional content can specify the default states for the optional content groups in the document, as well as indicating which external factors should be used to alter the states. The following sections describe the PDF structures that are used to specify this information.

- "Optional Content Properties Dictionary" on page 337 describes the structure that lists all the optional content groups in the document and their possible configurations.
- "Optional Content Configuration Dictionaries" on page 338 describes the structures that specify initial state settings and other information about the groups in the document.
- "Usage and Usage Application Dictionaries" on page 341 and "Determining the State of Optional Content Groups" on page 347 describe how the states of groups can be affected based on external factors.

Optional Content Properties Dictionary

The optional **OCProperties** entry in the document catalog (see Section 3.6.1, "Document Catalog") holds the *optional content properties dictionary*, which contains a list of all the optional content groups in the document, as well as information about the default and alternate configurations for optional content. This dictionary is required if the file contains any optional content; if it is missing, a PDF consumer should ignore any optional content structures in the document.

This dictionary contains the following entries:

TABLE 4.47 Entries in the optional content properties dictionary

KEY	TYPE	VALUE
OCGs	array	*(Required)* An array of indirect references to all the optional content groups in the document (see Section 4.10.1, "Optional Content Groups"), in any order. Every optional content group must be included in this array.
D	dictionary	*(Required)* The default viewing optional content configuration dictionary (see "Optional Content Configuration Dictionaries," below).
Configs	array	*(Optional)* An array of alternate optional content configuration dictionaries (see "Optional Content Configuration Dictionaries," below) for PDF processing applications or features.

Optional Content Configuration Dictionaries

The **D** and **Configs** entries in Table 4.47 are *configuration dictionaries*, which represent different presentations of a document's optional content groups for use by PDF processing applications or features. The **D** configuration dictionary specifies the initial state of the optional content groups when a document is first opened. **Configs** lists other configurations that may be used under particular circumstances. The entries in a configuration dictionary are shown in Table 4.48.

TABLE 4.48 Entries in an optional content configuration dictionary

KEY	TYPE	VALUE
Name	text string	*(Optional)* A name for the configuration, suitable for presentation in a user interface.
Creator	text string	*(Optional)* Name of the application or feature that created this configuration dictionary.
BaseState	name	*(Optional)* Used to initialize the states of all the optional content groups in a document when this configuration is applied. The value of this entry must be one of the following names: • **ON**: the states of all groups are turned **ON**. • **OFF**: the states of all groups are turned **OFF**. • **Unchanged**: the states of all groups are left unchanged. After this initialization, the contents of the **ON** and **OFF** arrays are processed, overriding the state of the groups included in the arrays. Default value: **ON**. ***Note:*** *If **BaseState** is present in the document's default configuration dictionary, its value must be **ON**.*
ON	array	*(Optional)* An array of optional content groups whose state should be set to **ON** when this configuration is applied. ***Note:*** *If the **BaseState** entry is **ON**, this entry is redundant.*
OFF	array	*(Optional)* An array of optional content groups whose state should be set to **OFF** when this configuration is applied. ***Note:*** *If the **BaseState** entry is **OFF**, this entry is redundant.*

KEY	TYPE	VALUE
Intent	name or array	*(Optional)* Used to determine which optional content groups' states to consider and ignore in calculating the visibility of content (see "Intent" on page 331). The value may be one of the following: • A single name. PDF 1.5 defines two intent names, **View** and **Design**; future versions may define others. In addition, the name **All** indicates the set of all intents, including those not yet defined. • An array containing any combination of names. Default value: **View**. The value must be **View** for the document's default configuration.
AS	array	*(Optional)* An array of usage application dictionaries (see Table 4.50) specifying which usage dictionary categories (see Table 4.49) should be consulted by viewer applications to automatically set the states of optional content groups based on external factors, such as the current system language or viewing magnification, and when they should be applied.
Order	array	*(Optional)* An array specifying the recommended order for presentation of optional content groups in a user interface. The array elements may include the following (as illustrated in below): • Optional content group dictionaries, whose **Name** entry is to be displayed in the user interface. • Arrays of optional content groups, to allow nesting as in a tree or outline structure. Each nested array may optionally have as its first element a text string, to be used as a non-selectable label in the user interface. ***Note:*** *Text labels in nested arrays should be used to present collections of related optional content groups, and not to communicate actual nesting of content inside multiple layers of groups (see Example 4.31). To reflect actual nesting of groups in the content, such as for layers with sub-layers, nested arrays of groups without a text label should be used (see Example 4.32).* An empty array [] explicitly specifies that no groups should be presented. In the default configuration dictionary, the default value is an empty array; in other configuration dictionaries, the default is the **Order** value from the default configuration dictionary. ***Note:*** *Any groups not listed in this array should not be presented in any user interface that uses the configuration.*

KEY	TYPE	VALUE
ListMode	name	*(Optional)* A name specifying which optional content groups in the **Order** array should be displayed to the user. Possible values are: • AllPages: display all groups in the **Order** array. • VisiblePages: display only those groups in the **Order** array that are referenced by one or more visible pages. Default value: **AllPages**.
RBGroups	array	*(Optional)* An array consisting of one or more arrays, each of which represents a collection of optional content groups whose states are intended to follow a "radio button" paradigm. That is, the state of at most one optional content group in each array should be **ON** at a time: if one group is turned **ON**, all others must be turned **OFF**; however, turning a group from **ON** to **OFF** does not force any other group to be turned **ON**. An empty array [] explicitly specifies that there are no such collections. In the default configuration dictionary, the default value is an empty array; in other configuration dictionaries, the default is the **RBGroups** value from the default configuration dictionary.

Examples 4.31 and 4.32 illustrates the use of the **Order** entry to control the display of groups in a user interface.

Example 4.31

Given the following PDF objects:

```
1 0 obj <</Type /OCG /Name (Skin)>> endobj        % Optional content groups
2 0 obj <</Type /OCG /Name (Bones)>> endobj
3 0 obj <</Type /OCG /Name (Bark)>> endobj
4 0 obj <</Type /OCG /Name (Wood)>> endobj

5 0 obj                                            % Configuration dictionary
   << /Order [[(Frog Anatomy) 1 0 R 2 0 R] [(Tree Anatomy) 3 0 R 4 0 R] ] >>
```

A PDF viewer should display the optional content groups as follows:

```
Frog Anatomy
   Skin
   Bones
Tree Anatomy
   Bark
   Wood
```

Example 4.32

Given the following PDF objects:

```
                                        % Page contents
  /OC /L1 BDC                           % Layer 1
    /OC /L1a BDC                        % Sublayer A of layer 1
      0 0 100 100 re f
    EMC
    /OC /L1b BDC                        % Sublayer B of layer 1
      0 100 100 100 re f
    EMC
  EMC
  ...
  << /L1 1 0 R                          % Resource names
    /L1a 2 0 R
    /L1b 3 0 R
  >>
  ...                                   %Optional content groups
  1 0 obj <</Type /OCG /Name (Layer 1)>> endobj
  2 0 obj <</Type /OCG /Name (Sublayer A)>> endobj
  3 0 obj <</Type /OCG /Name (Sublayer B)>> endobj
  ...
  4 0 obj                               % Configuration dictionary
    << /Order [1 0 R [2 0 R 3 0 R]] >>
```

A PDF viewer should display the OCGs as follows:

```
  Layer 1
    Sublayer A
    Sublayer B
```

The **AS** entry is an *auto state* array, consisting of one or more *usage application dictionaries* that specify how viewer applications should automatically set the state of optional content groups based on external factors, as discussed in the following section.

Usage and Usage Application Dictionaries

Optional content groups are typically constructed to control the visibility of graphic objects that are related in some way. There are several ways in which objects can be related; for example, a group may contain content in a particular language, or content suitable for viewing at a particular magnification.

An optional content group's usage dictionary (the value of the **Usage** entry in an optional content group dictionary; see Table 4.45) contains information describing the nature of the content controlled by the group. This dictionary can contain any combination of the entries shown in Table 4.49.

TABLE 4.49 Entries in an optional content usage dictionary

KEY	TYPE	VALUE
CreatorInfo	dictionary	*(Optional)* A dictionary used by the creating application to store application-specific data associated with this optional content group. It contains two required entries: • **Creator**: a text string specifying the application that created the group. • **Subtype**: a name defining the type of content controlled by the group. Suggested values include but are not limited to **Artwork**, for graphic-design or publishing applications, and **Technical**, for technical designs such as building plans or schematics. Additional entries may be included to present information relevant to the creating application or related applications. ***Note:*** *Groups whose* **Intent** *entry contains* **Design** *typically include a* **CreatorInfo** *entry.*
Language	dictionary	*(Optional)* A dictionary specifying the language of the content controlled by this optional content group. It has two entries: • **Lang** *(required)*: a language string (see Section 10.8.1, "Natural Language Specification") which specifies a language and possibly a locale (for example, es-MX represents Mexican Spanish). • **Preferred** *(optional)*: a name whose values may be **ON** or **OFF**. Default value: **OFF**. It is used by viewer applications when there is a partial match but no exact match between the system language and the language strings in all usage dictionaries. See "Usage and Usage Application Dictionaries" on page 341 for more information.
Export	dictionary	*(Optional)* A dictionary specifying the recommended state for content in this group when the document (or part of it) is saved by a viewer application to a format that does not support optional content (for example, an earlier version of PDF or a raster image format). It contains one entry, **ExportState**, a name whose value may be **ON** or **OFF**, which represents the recommended state of the group.

KEY	TYPE	VALUE
Zoom	dictionary	*(Optional)* A dictionary specifying a range of magnifications at which the content in this optional content group is best viewed. It may contain one or both of the following entries: • **max**: the maximum recommended magnification factor at which the group should be **ON**. Default: the largest possible magnification supported by the viewer application. • **min**: the minimum recommended magnification factors at which the group should be **ON**. Default: 0.
Print	dictionary	*(Optional)* A dictionary specifying that the content in this group is intended for use in printing. It contains the following optional entries: • **Subtype**: a name object specifying the kind of content controlled by the group; for example, **Trapping**, **PrintersMarks** and **Watermark**. • **PrintState**: a name that may be **ON** or **OFF**, indicating that the group should be set to that state when the document is printed from a viewer application.
View	dictionary	*(Optional)* A dictionary that has a single entry, **ViewState**, a name that may have a value of **ON** or **OFF**, indicating that the group should be set to that state when the document is opened in a viewer application.
User	dictionary	*(Optional)* A dictionary specifying one or more users for whom this optional content group is primarily intended. Each dictionary has two required entries: • **Type**: a name object that can be **Ind** (individual), **Ttl** (title) or **Org** (organization). • **Name**: a text string or array of text strings representing the name(s) of the individual, position or organization.
PageElement	dictionary	*(Optional)* A dictionary declaring that the group contains a pagination artifact. It contains one entry, **Subtype**, whose value is a name that can be **HF** (header/footer), **FG** (foreground image or graphic), **BG** (background image or graphic) or **L** (logo).

While the data in the usage dictionary can be viewed as information for a document user to examine, it may also be used by viewer applications to automatically manipulate the state of optional content groups based on external factors such as current system language settings or zoom level. Document authors may specify which entries in the usage dictionary should be consulted to automatically set the state of optional content groups based on such factors. This is done by means of *usage application dictionaries*, which are specified by the **AS** entry in an optional content configuration dictionary (see Table 4.48). If no **AS** entry is present, states are not automatically adjusted based on usage information.

A usage application dictionary specifies the rules for which usage entries should be used by viewer applications to automatically manipulate the state of optional content groups, which groups should be affected, and under which circumstances. Table 4.50 shows the entries in a usage application dictionary.

***Note:** Usage application dictionaries are only intended for use by interactive viewer applications, not for applications that use PDF as final form output (see "Determining the State of Optional Content Groups" on page 347 for more information).*

TABLE 4.50 Entries in a usage application dictionary

KEY	TYPE	VALUE
Event	name	*(Required)* A name defining the situation in which this usage application dictionary should be used. May be **View**, **Print**, or **Export**.
OCGs	array	*(Optional)* An array listing the optional content groups that should have their states automatically managed based on information in their usage dictionary (see "Usage and Usage Application Dictionaries" on page 341). Default value: an empty array, indicating that no groups are affected.
Category	array	*(Required)* An array of names, each of which corresponds to a usage dictionary entry (see Table 4.49). When managing the states of the optional content groups in the **OCGs** array, each of the corresponding categories in the group's usage dictionary should be considered.

The **Event** entry specifies whether the usage settings should be applied during viewing, printing or exporting the document. The **OCGs** entry specifies the set of optional content groups to which usage settings should be applied. For each of the groups in **OCGs**, the entries in its usage dictionary (see Table 4.49) specified by **Category** are examined to yield a *recommended state* for the group. If all the entries yield a recommended state of **ON**, the group's state is set to **ON**; otherwise, its state is set to **OFF**.

The entries in the usage dictionary are used as follows:

- **View**: The recommended state is the value of the **ViewState** entry. This entry allows a document to contain content that is relevant only when the document is viewed interactively, such as instructions for how to interact with the document.
- **Print**: The recommended state is the value of the **PrintState** entry; if **PrintState** is not present, the state of the optional content group is left unchanged.
- **Export**: The recommended state is the value of the **ExportState** entry.
- **Zoom**: If the current magnification level of the document is greater than **min** and less than or equal to **max**, an **ON** state is recommended; otherwise **OFF** is recommended.
- **User**: The **Name** entry specifies a name or names to match with the user's identification; the **Type** entry determines how the **Name** entry is interpreted (name, title, or organization). If there is an exact match, an **ON** state is recommended; otherwise **OFF** is recommended.
- **Language**: This category allows the selection of content based on the language and locale of the viewer application. If an exact match to the language and locale is found among the **Lang** entries of the optional content groups in the usage application dictionary's **OCGs** list, all groups that have exact matches receive an **ON** recommendation. If no exact match is found, but a partial match is found (that is, the language matches, but not the locale), all partially matching groups that have **Preferred** entries with a value of **ON** receive an **ON** recommendation. All other groups receive an **OFF** recommendation.

Example 4.33 shows the use of an auto state array with usage application dictionaries. The **AS** entry in the default configuration dictionary is an array of three usage application dictionaries, one for each of the **Event** values **View**, **Print**, and **Export**.

***Note:** While this is a typical case, there is no restriction on multiple entries with the same value of **Event**, which allows documents with incompatible usage application dictionaries to be combined into larger documents and have their behavior preserved. If a given optional content group appears in more than one **OCGs** array, its state is **ON** only if all categories in all the usage application dictionaries it appears in recommend a state of **ON**.*

Example 4.33

```
/OCProperties                          % OCProperties dictionary in document catalog
    << /OCGs [1 0 R 2 0 R 3 0 R 4 0 R]
        /D << /BaseState /OFF          % The default configuration
                /ON [1 0 R]
                /AS [                  % Auto state array of usage application dictionaries
                    << /Event /View /Category [/Zoom] /OCGs [1 0 R 2 0 R 3 0 R 4 0 R] >>
                    << /Event /Print /Category [/Print] /OCGs [4 0 R] >>
                    << /Event /Export /Category [/Export] /OCGs [3 0 R 4 0 R] >>
                    ]
            >>
    >>
...

1 0 obj
    << /Type /OCG
        /Name (20000 foot view)
        /Usage << /Zoom << /max 1.0 >> >>
    >>
endobj
2 0 obj
    << /Type /OCG
        /Name (10000 foot view)
        /Usage << /Zoom << /min 1.0 /max 2.0 >> >>
>>
endobj
3 0 obj
    << /Type /OCG
        /Name (1000 foot view)
        /Usage << /Zoom << /min 2.0 /max 20.0 >>
                    /Export << /ExportState /OFF >> >>
        >>
endobj
4 0 obj
    << /Type /OCG
        /Name (Copyright notice)
        /Usage << /Print << /PrintState /ON >>
                    /Export << /ExportState /ON>> >>
    >>
endobj
```

In the example, the usage application dictionary with event type **View** specifies that all optional content groups are to have their states managed based on zoom level when viewing. Three groups (objects 1, 2, and 3) contain **Zoom** usage information; object 4 has none, so it is not affected by zoom level changes. Object 3 receives an **OFF** recommendation when exporting. When printing or exporting, object 4 receives an **ON** recommendation.

Determining the State of Optional Content Groups

This section summarizes the rules by which applications make use of the configuration and usage application dictionaries to set the state of optional content groups. For purposes of this discussion, it is useful to distinguish the following types of applications:

- Viewer applications, such as Adobe Acrobat, which allow users to interact with the document in various ways.
- Design applications, which offer layering features for collecting groups of graphics together and selectively hiding or viewing them.

  Note: *The following rules are not meant to apply to design applications; they may manage their states in an entirely different manner if they choose.*
- Aggregating applications, which import PDF files as graphics.
- Printing applications, which print PDF files.

When a document is first opened, its optional content groups are assigned a state based on the **D** (default) configuration dictionary in the **OCProperties** dictionary.

1. The value of **BaseState** is applied to all the groups.
2. The groups listed in either the **ON** or **OFF** array (depending on which one is opposite to **BaseState**) have their states adjusted.

This state is the recommended state for printing and aggregating applications, which should not apply the changes based on usage application dictionaries described below. However, for more advanced functionality, they may provide user control for manipulating the individual states of optional content groups.

Note: *Viewer applications should also provide users with an option to view documents in this state (that is, to disable the automatic adjustments discussed below), in order to provide an accurate preview of the content as it would appear when placed into an aggregating application or sent to a stand-alone printing system.*

The remaining discussion in this section applies only to viewer applications. Such applications should examine the **AS** array for usage application dictionaries that have an **Event** of type **View**. For each one found, the groups listed in its **OCGs** array should be adjusted as described in "Usage and Usage Application Dictionaries" on page 341.

Subsequently, the document is ready for interactive viewing by a user. Whenever there is a change to a factor that the usage application dictionaries with event type **View** depend on (such as zoom level), the corresponding dictionaries should be re-applied.

The user may manipulate optional content group states manually or by triggering **SetOCGState** actions (see "Set-OCG-State Actions" on page 608) by, for example, clicking links or bookmarks. Manual changes override the states that were set automatically; the states of these groups remain overridden and are not readjusted based on usage application dictionaries with event type **View** as long as the document is open (or until the user reverts the document to its original state).

When a document is printed by a viewer application, usage application dictionaries with an event type **Print** are applied over the current states of optional content groups. These changes persist only for the duration of the print operation; then all groups revert to their prior states.

Similarly, when a document is exported to an earlier version of PDF or other format that does not support optional content, usage application dictionaries with an event type **Export** are applied over the current states of optional content groups. Changes persist only for the duration of the export operation; then all groups revert to their prior states.

Note: *Although the event types **Print** and **Export** have identically named counterparts that are usage categories, the corresponding usage application dictionaries are permitted to specify that other categories may be applied.*

CHAPTER 5

Text

THIS CHAPTER DESCRIBES the special facilities in PDF for dealing with text—specifically, 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, PDF provides higher-level facilities that permit an application to describe, select, and render glyphs conveniently and efficiently.

The first section is a general description of how glyphs from fonts are painted on the page. Subsequent sections cover the following topics in detail:

- *Text state*. A subset of the graphics state parameters pertain to text, including parameters that select the font, scale the glyphs to an appropriate size, and accomplish other graphical effects.
- *Text objects and operators*. The text operators specify the glyphs to be painted, represented by string objects whose values are interpreted as sequences of character codes. A text object encloses a sequence of text operators and associated parameters.
- *Font data structures*. Font dictionaries and associated data structures provide information that a viewer application needs to interpret the text and position the glyphs properly. The definitions of the glyphs themselves are contained in *font programs*, which may be embedded in the PDF file, built into the viewer application, or obtained from an external font file.

5.1 Organization and Use of Fonts

A *character* is an abstract symbol, whereas a *glyph* is a specific graphical 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 PDF dictionary keys and PostScript operators), 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 fonts define glyphs for a set of standard Latin characters. A font for use with a PDF viewer application is prepared in the form of a program. Such a *font program* is written in a special-purpose language, such as the *Type 1* or *TrueType* font format, that is understood by a specialized font interpreter.

In PDF, the term *font* refers to a *font dictionary*, a PDF object that identifies the font program and contains additional information about it. There are several different font types, identified by the **Subtype** entry of the font dictionary.

For most font types, the font program itself is defined in a separate *font file*, which may be either embedded in a PDF stream object or obtained from an external source. The font program contains *glyph descriptions* that generate glyphs.

A content stream paints glyphs on the page by specifying a font dictionary and a string object that is interpreted as a sequence of one or more character codes identifying glyphs in the font. This operation is called *showing* the text string; the text strings drawn in this way are called *show strings*. The glyph description consists of a sequence of graphics operators that produce the specific shape for that character in this font. To render a glyph, the viewer application executes the glyph description.

Programmers who have experience with scan conversion of general shapes 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, an efficient implementation can be achieved through careful caching and reuse of previously rendered glyphs.

5.1.1 Basics of Showing Text

Example 5.1 illustrates the most straightforward use of a font. It places 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

```
BT
   /F13  12  Tf
   288  720  Td
   (ABC)  Tj
ET
```

The five lines of this example perform the following steps:

1. Begin a text object.
2. Set the font and font size to use, installing them as parameters in the text state. (The font resource identified by the name F13 specifies the font externally known as Helvetica.)
3. Specify a starting position on the page, setting parameters in the text object.
4. Paint the glyphs for a string of characters there.
5. End the text object.

The following paragraphs explain these operations in more detail.

To paint glyphs, a content stream must first identify the font to be used. The **Tf** operator specifies the name of a font resource—that is, an entry in the **Font** subdictionary of the current resource dictionary. The value of that entry is a font dictionary. The font dictionary in turn identifies the font's externally known name, such as Helvetica, and supplies some additional information that the viewer application needs to paint glyphs from that font; it optionally provides the definition of the font program itself.

Note: *The font resource name presented to the* ***Tf*** *operator is arbitrary, as are the names for all kinds of resources. It bears no relationship to an actual font name, such as Helvetica.*

Example 5.2 illustrates an excerpt from the current page's resource dictionary, defining the font dictionary that is referenced as F13 in Example 5.1.

Example 5.2

```
/Resources
    << /Font << /F13 23 0 R >>
    >>
23 0 obj
    << /Type /Font
         /Subtype /Type1
         /BaseFont /Helvetica
    >>
endobj
```

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 scale factor is specified as the second operand of the **Tf** operator, thereby setting the *text font size* parameter in the graphics state. Example 5.1 establishes the Helvetica font with a 12-unit size in the graphics state.

Once the font has been selected and scaled, it can be used to paint glyphs. The **Td** operator adjusts the current text position (actually, the translation components of the text matrix, as described in Section 5.3.1, "Text-Positioning Operators"). When executed for the first time after **BT**, it establishes the text position in the current user coordinate system. This determines the position on the page at which to begin painting glyphs.

The **Tj** operator takes a string operand and paints the corresponding glyphs using the current font and other text-related parameters in the graphics state. In Example 5.1, the **Tj** 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, and the glyph description is executed to paint that glyph on the page. This is the behavior of **Tj** for simple fonts, such as ordinary Latin text fonts; interpretation of the string as a sequence of character codes is more complex for composite fonts, described in Section 5.6, "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 text space at the time the glyphs are rendered on the page. The actual size of the glyph is determined by the text matrix* (T_m) *in the text object, several text state parameters, and the current transforma-*

tion matrix (CTM) in the graphics state; see Section 5.3.3, "Text Space Details." If the text space is later scaled to make the unit size 1 centimeter, painting glyphs from the same 12-unit font will generate results that are 12 centimeters high.

5.1.2 Achieving Special Graphical Effects

Normal uses of **Tj** 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: either the nonstroking or the stroking color (or both), depending on the text rendering mode (see Section 5.2.5, "Text Rendering Mode"). The default color is black, but other colors can be obtained by executing an appropriate color-setting operator or operators (see Section 4.5.7, "Color Operators") before painting the glyphs. Example 5.3 uses text rendering mode 0 and the **g** operator to fill glyphs in 50 percent gray, as shown in Figure 5.1.

Example 5.3

```
BT
   /F13 48 Tf
   20 40 Td
   0 Tr
   0.5 g
   (ABC) Tj
ET
```

FIGURE 5.1 *Glyphs painted in 50% gray*

Other graphical effects can be achieved by treating the glyph outline as a path instead of filling it. The *text rendering mode* parameter in the graphics state specifies whether glyph outlines are to be filled, stroked, used as a clipping boundary, or some combination of these effects. (This parameter does not apply to Type 3 fonts.)

Example 5.4 treats glyph outlines as a path to be stroked. The **Tr** operator sets the text rendering mode to 1 (stroke). The **w** operator sets the line width to 2 units in user space. Given those graphics state parameters, the **Tj** operator strokes the glyph outlines with a line 2 points thick (see Figure 5.2).

Example 5.4

```
BT
    /F13  48  Tf
    20  38  Td
    1  Tr
    2  w
    (ABC)  Tj
ET
```

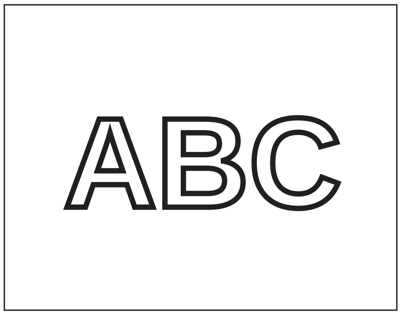


FIGURE 5.2 *Glyph outlines treated as a stroked path*

Example 5.5 treats the glyphs' outlines as a clipping boundary. The **Tr** operator sets the text rendering mode to 7 (clip), causing the subsequent **Tj** operator to impose the glyph outlines as the current clipping path. All subsequent painting operations will mark the page only within this path, as illustrated in Figure 5.3. This state persists until some earlier clipping path is reinstated by the **Q** operator.

Example 5.5

```
BT
    /F13  48  Tf
    20  38  Td
    7  Tr
    (ABC)  Tj
ET
```
…Graphics operators to draw a starburst…

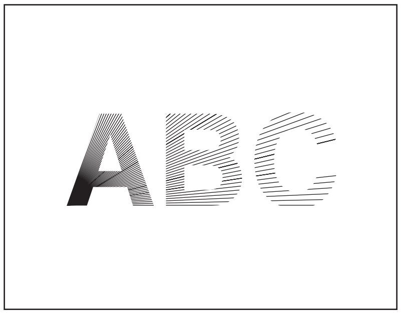


FIGURE 5.3 *Graphics clipped by a glyph path*

5.1.3 Glyph Positioning and Metrics

A glyph's *width*—formally, its *horizontal displacement*—is the amount of space it occupies along the baseline of a line of text that is written horizontally. In other words, it is the distance the current text position moves (by translating text space) when the glyph is painted. Note that the width is distinct from the dimensions of the glyph outline.

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* or *variable-pitch* fonts. In either case, the **Tj** operator positions the consecutive glyphs of a string according to their widths.

The width information for each glyph is stored both in the font dictionary and in the font program itself. (The two sets of widths must be identical; storing this information in the font dictionary, although redundant, enables a viewer application to determine glyph positioning without having to look inside the font program.) The operators for showing text are designed on the assumption that glyphs are ordinarily positioned according to their standard widths. However, means are provided to vary the positioning in certain limited ways. For example, the **TJ** operator enables the text position to be adjusted between any consecutive pair of glyphs corresponding to characters in a text string. There are graphics state parameters to adjust character and word spacing systematically.

In addition to width, a glyph has several other metrics that influence glyph positioning and painting. For most font types, this information is largely internal to the font program and is not specified explicitly in the PDF font dictionary; however, in a Type 3 font, all metrics are specified explicitly (see Section 5.5.4, "Type 3 Fonts").

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. For all font types except Type 3, the units of glyph space are one-thousandth of a unit of text space; for a Type 3 font, the transformation from glyph space to text space is defined by a *font matrix* specified in an explicit **FontMatrix** entry in the font. Figure 5.4 shows a typical glyph outline and its metrics.

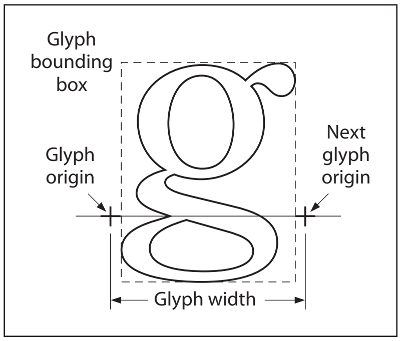


FIGURE 5.4 *Glyph metrics*

The *glyph origin* is the point (0, 0) in the glyph coordinate system. **Tj** and other text-showing operators position the origin of the first glyph to be painted at the origin of text space. For example, the following code adjusts the origin of text space to (40, 50) in the user coordinate system, and then places the origin of the A glyph at that point:

```
BT
   40 50 Td
   (ABC) Tj
ET
```

The *glyph displacement* 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 line of text. This distance is a vector (called the *displacement vector*) in the glyph coordinate system; it has horizontal and vertical components. (A displacement that is horizontal is usually called a *width*.) Most Western writing systems, including those based on the Latin alphabet, have a positive horizontal displacement and a zero vertical displacement; some Asian writing systems have a nonzero vertical displacement. In all cases, the text-showing operators transform the displacement vector into text space and then translate text space by that amount.

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.

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. Which set of metrics to use is selected according to a *writing mode*, where 0 specifies horizontal writing and 1 specifies vertical writing. This feature is available only for composite fonts, discussed in Section 5.6, "Composite Fonts."

When a glyph has two sets of metrics, each set specifies a glyph origin and a displacement vector for that writing mode. In vertical writing, the glyph position is described by a *position vector* from the origin used for horizontal writing (origin 0) to the origin used for vertical writing (origin 1). Figure 5.5 illustrates the metrics for the two writing modes, as follows:

- The left diagram illustrates the glyph metrics associated with writing mode 0, horizontal writing. The coordinates *ll* and *ur* specify the bounding box of the glyph relative to origin 0. *w0* is the displacement vector that specifies how the text position is changed after the glyph is painted in writing mode 0; its vertical component is always 0.

- The center diagram illustrates writing mode 1, vertical writing. *w1* is the displacement vector for writing mode 1; its horizontal component is always 0.
- In the right diagram, *v* is a position vector defining the position of origin 1 relative to origin 0.

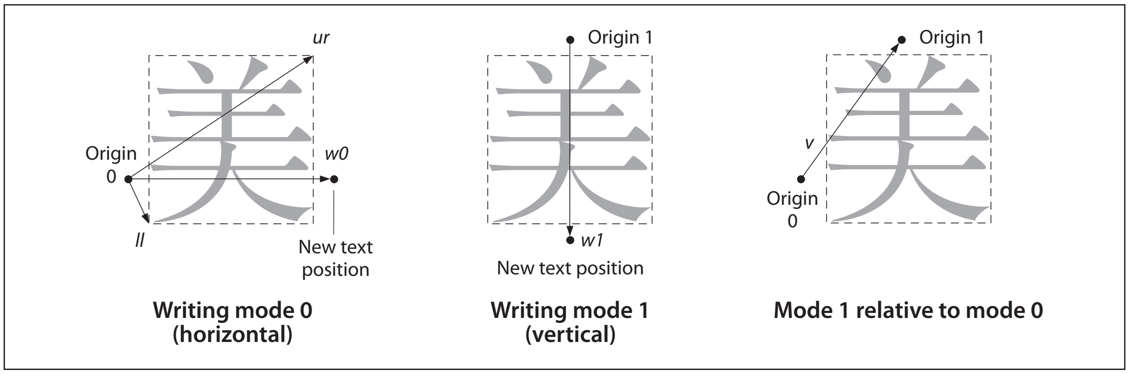

ur
Origin 0
w0
ll
New text position
Writing mode 0 (horizontal)
Origin 1
w1
New text position
Writing mode 1 (vertical)
Origin 1
v
Origin 0
Mode 1 relative to mode 0

FIGURE 5.5 *Metrics for horizontal and vertical writing modes*

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 PDF page descriptions and must make formatting decisions based on the widths and other metrics of glyphs. Also available in the AFM and ACFM files is kerning information, which allows an application generating a PDF file to determine spacing adjustments between glyphs depending on context. Specifications for the AFM and ACFM file formats are available in Adobe Technical Note #5004, *Adobe Font Metrics File Format Specification*; the files themselves can be obtained from the Adobe Solutions Network Web site (see the Bibliography).

5.2 Text State Parameters and Operators

The *text state* comprises those graphics state parameters that only affect text. There are nine parameters in the text state (see Table 5.1).

TABLE 5.1 Text state parameters

PARAMETER	DESCRIPTION
T_c	Character spacing
T_w	Word spacing
T_h	Horizontal scaling
T_l	Leading
T_f	Text font
T_{fs}	Text font size
T_{mode}	Text rendering mode
T_{rise}	Text rise
T_k	Text knockout

Except for the self-explanatory T_f and T_{fs}, these parameters are discussed further in the following sections. (As described in Section 5.3, "Text Objects," three additional text-related parameters are defined only within a text object: T_m, the text matrix; T_{lm}, the text line matrix; and T_{rm}, the text rendering matrix.) The values of the text state parameters are consulted when text is positioned and shown (using the operators described in Sections 5.3.1, "Text-Positioning Operators," and 5.3.2, "Text-Showing Operators"). In particular, the spacing and scaling parameters participate in a computation described in Section 5.3.3, "Text Space Details." The text state parameters can be set using the operators listed in Table 5.2.

Note:** The text knockout parameter,* $\mathrm{T_k}$*, is set via the* ***TK *entry in a graphics state parameter dictionary, using the* ***gs*** *operator (see Section 4.3.4, "Graphics State Parameter Dictionaries"). There is no specific operator for setting this parameter.*

The text state operators can appear outside text objects, and the values they set are retained across text objects in a single content stream. Like other graphics state parameters, these parameters are initialized to their default values at the beginning of each page.

TABLE 5.2 Text state operators

OPERANDS	OPERATOR	DESCRIPTION
charSpace	**Tc**	Set the character spacing, T_c, to *charSpace*, which is a number expressed in unscaled text space units. Character spacing is used by the **Tj**, **TJ**, and **'** operators. Initial value: 0.
wordSpace	**Tw**	Set the word spacing, T_w, to *wordSpace*, which is a number expressed in unscaled text space units. Word spacing is used by the **Tj**, **TJ**, and **'** operators. Initial value: 0.
scale	**Tz**	Set the horizontal scaling, T_h, to (*scale* ÷ 100). *scale* is a number specifying the percentage of the normal width. Initial value: 100 (normal width).
leading	**TL**	Set the text leading, T_l, to *leading*, which is a number expressed in unscaled text space units. Text leading is used only by the **T***, **'**, and **"** operators. Initial value: 0.
font size	**Tf**	Set the text font, T_f, to *font* and the text font size, T_{fs}, to *size*. *font* is the name of a font resource in the **Font** subdictionary of the current resource dictionary; *size* is a number representing a scale factor. There is no initial value for either *font* or *size*; they must be specified explicitly using **Tf** before any text is shown.
render	**Tr**	Set the text rendering mode, T_{mode}, to *render*, which is an integer. Initial value: 0.
rise	**Ts**	Set the text rise, T_{rise}, to *rise*, which is a number expressed in unscaled text space units. Initial value: 0.

Note that some of these parameters are expressed in *unscaled* text space units. This means that they are specified in a coordinate system that is defined by the text matrix, T_m, but is not scaled by the font size parameter, T_{fs}.

5.2.1 Character Spacing

The character spacing parameter, T_c, is a number specified in unscaled text space units (although it is subject to scaling by the T_h parameter if the writing mode is horizontal). When the glyph for each character in the string is rendered, T_c is *added* to the horizontal or vertical component of the glyph's displacement, depending on the writing mode. (See Section 5.1.3, "Glyph Positioning and Metrics," for a discussion of glyph displacements.) In the default coordinate system, horizontal coordinates increase from left to right and vertical coordinates from bottom to top. So for horizontal writing, a positive value of T_c has the effect of expanding the distance between glyphs (see Figure 5.6), whereas for vertical writing, a *negative* value of T_c has this effect.

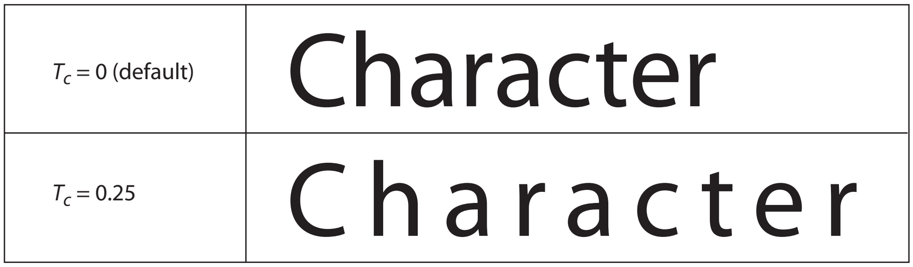


FIGURE 5.6 *Character spacing in horizontal writing*

5.2.2 Word Spacing

Word spacing works the same way as character spacing, but applies only to the space character, code 32. The word spacing parameter, T_w, is added to the glyph's horizontal or vertical displacement (depending on the writing mode). For horizontal writing, a positive value for T_w has the effect of increasing the spacing between words. For vertical writing, a positive value for T_w *decreases* the spacing between words (and a negative value increases it), since vertical coordinates increase from bottom to top. Figure 5.7 illustrates the effect of word spacing in horizontal writing.

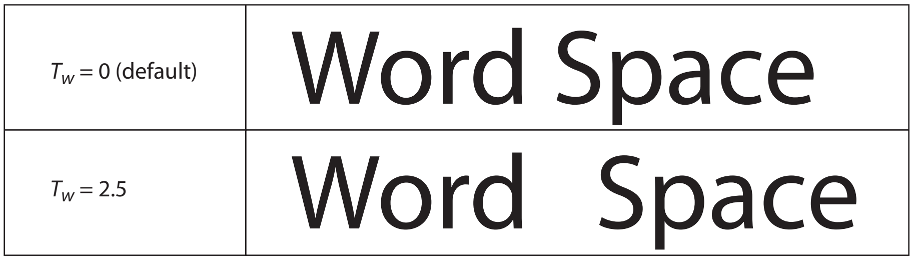


FIGURE 5.7 *Word spacing in horizontal writing*

***Note:** Word spacing is applied to every occurrence of the single-byte character code 32 in a string. This can occur when using a simple font or a composite font that de-*

fines code 32 as a single-byte code. It does not apply to occurrences of the byte value 32 in multiple-byte codes.

5.2.3 Horizontal Scaling

The horizontal scaling parameter, T_h, adjusts the width of glyphs by stretching or compressing them in the horizontal direction. Its value is specified as a percentage of the normal width of the glyphs, with 100 being the normal width. The scaling always applies to the horizontal coordinate in text space, independently of the writing mode. It affects both the glyph's shape and its horizontal displacement (that is, its displacement vector). If the writing mode is horizontal, it also affects the spacing parameters T_c and T_w, as well as any positioning adjustments performed by the **TJ** operator. Figure 5.8 shows the effect of horizontal scaling.

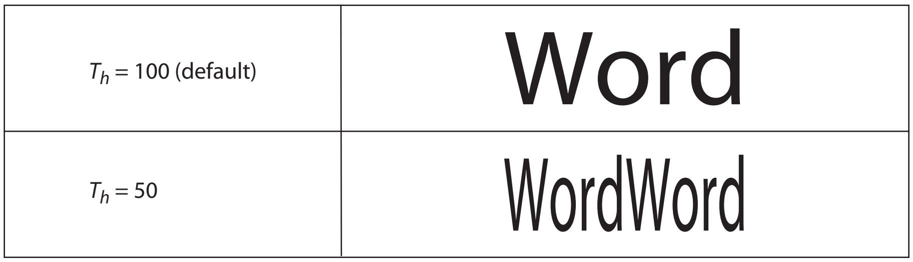


FIGURE 5.8 *Horizontal scaling*

5.2.4 Leading

The leading parameter, T_l, is measured in unscaled text space units. It specifies the vertical distance between the baselines of adjacent lines of text, as shown in Figure 5.9.

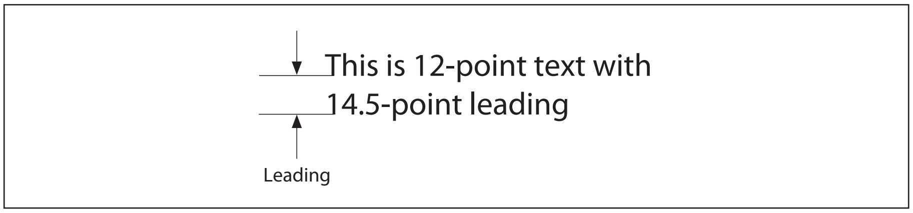


FIGURE 5.9 *Leading*

The leading parameter is used by the **TD**, **T***, **'**, and **"** operators; see Table 5.5 on page 368 for a precise description of its effects. This parameter always applies to the vertical coordinate in text space, independently of the writing mode.

5.2.5 Text Rendering Mode

The text rendering mode, T_{mode}, determines whether showing text causes glyph outlines to be stroked, filled, used as a clipping boundary, or some combination of the three. Stroking, filling, and clipping have the same effects for a text object as they do for a path object (see Sections 4.4.2, "Path-Painting Operators," and 4.4.3, "Clipping Path Operators"), although they are specified in an entirely different way. The graphics state parameters affecting those operations, such as line width, are interpreted in user space rather than in text space.

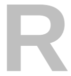
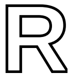
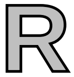
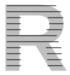
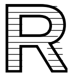
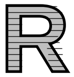
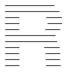
Note: *The text rendering mode has no effect on text displayed in a Type 3 font (see Section 5.5.4, "Type 3 Fonts").*

The text rendering modes are shown in Table 5.3. In the examples, a stroke color of black and a fill color of light gray are used. For the clipping modes (4 to 7), a series of lines has been drawn through the glyphs to show where the clipping occurs.

If the text rendering mode calls for filling, the current nonstroking color in the graphics state is used; if it calls for stroking, the current stroking color is used. In modes that perform both filling and stroking, the effect is as if each glyph outline were filled and then stroked in separate operations. If any of the glyphs overlap, the result is equivalent to filling and stroking them one at a time, producing the appearance of stacked opaque glyphs, rather than first filling and then stroking them all at once (see implementation note 49 in Appendix H). In the transparent imaging model, these combined filling and stroking modes are subject to further considerations; see "Special Path-Painting Considerations" on page 528.

The behavior of the clipping modes requires further explanation. Glyph outlines begin accumulating if a **BT** operator is executed while the text rendering mode is set to a clipping mode or if it is set to a clipping mode within a text object. Glyphs accumulate until the text object is ended by an **ET** operator; the text rendering mode must not be changed back to a nonclipping mode before that point.

TABLE 5.3 Text rendering modes

MODE	EXAMPLE	DESCRIPTION
0		Fill text.
1		Stroke text.
2		Fill, then stroke text.
3		Neither fill nor stroke text (invisible).
4		Fill text and add to path for clipping (see above).
5		Stroke text and add to path for clipping.
6		Fill, then stroke text and add to path for clipping.
7		Add text to path for clipping.

At the end of the text object, the accumulated glyph outlines, if any, are combined into a single path, treating the individual outlines as subpaths of that path and applying the nonzero winding number rule (see "Nonzero Winding Number Rule" on page 202). The current clipping path in the graphics state is set to the intersection of this path with the previous clipping path. As is the case for path objects, this clipping occurs *after* all filling and stroking operations for the text object have occurred. It remains in effect until some previous clipping path is restored by an invocation of the **Q** operator.

Note: *If no glyphs are shown, or if the only glyphs shown have no outlines (for example, if they are space characters), no clipping occurs.*

5.2.6 Text Rise

Text rise, T_{rise}, specifies the distance, in unscaled text space units, to move the baseline up or down from its default location. Positive values of text rise move the baseline up. Adjustments to the baseline are useful for drawing superscripts or subscripts. The default location of the baseline can be restored by setting the text rise to 0. Figure 5.10 illustrates the effect of the text rise. Text rise always applies to the vertical coordinate in text space, regardless of the writing mode.

(This text is) Tj 5 Ts (superscripted) Tj	This text is $^{\text{superscripted}}$
(This text is) Tj –5 Ts (subscripted) Tj	This text is $_{\text{subscripted}}$
(This) Tj –5 Ts (text) Tj 5 Ts (moves) Tj 0 Ts (around) Tj	This $_{\text{text}}$ $^{\text{moves}}$ around

FIGURE 5.10 *Text rise*

5.2.7 Text Knockout

The text knockout parameter, T_k *(PDF 1.4)*, is a boolean flag that determines what text elements are considered elementary objects for purposes of color compositing in the transparent imaging model. Unlike other text state parameters, there is no specific operator for setting this parameter; it can be set only via the **TK** entry in a graphics state parameter dictionary, using the **gs** operator (see Section 4.3.4, "Graphics State Parameter Dictionaries").

The text knockout parameter applies only to entire text objects; it may not be set between the **BT** and **ET** operators delimiting a text object. Its initial value is **true**. If it is **false**, each glyph in a text object is treated as a separate elementary object; when glyphs overlap, they will composite with one another.

If the parameter is **true**, all glyphs in the text object are treated together as a single elementary object; when glyphs overlap, later glyphs will overwrite ("knock out") earlier ones in the area of overlap. This behavior is equivalent to treating the entire text object as if it were a non-isolated knockout transparency group; see Section 7.3.5, "Knockout Groups." Transparency parameters are applied to the glyphs individually, rather than to the implicit transparency group as a whole, as follows:

- Graphics state parameters, including transparency parameters, are inherited from the context in which the text object appears. They are not saved and restored, nor are the transparency parameters reset at the beginning of the transparency group (as they are when a transparency group XObject is explicitly invoked). Changes made to graphics state parameters within the text object persist beyond the end of the text object.
- After the implicit transparency group for the text object has been completely evaluated, the group results are composited with the backdrop using the **Normal** blend mode and alpha and soft mask values of 1.0.

5.3 Text Objects

A PDF *text object* consists of operators that can show text strings, move the text position, and set text state and certain other parameters. In addition, there are three parameters that are defined only within a text object and do not persist from one text object to the next:

- T_m, the *text matrix*
- T_{lm}, the *text line matrix*
- T_{rm}, the *text rendering matrix*, actually just an intermediate result that combines the effects of text state parameters, the text matrix (T_m), and the current transformation matrix

A text object begins with the **BT** operator and ends with the **ET** operator, as shown below and described in Table 5.4.

```
BT
    …Zero or more text operators or other allowed operators…
ET
```

TABLE 5.4 Text object operators

OPERANDS	OPERATOR	DESCRIPTION
—	**BT**	Begin a text object, initializing the text matrix, T_m, and the text line matrix, T_{lm}, to the identity matrix. Text objects cannot be nested; a second **BT** cannot appear before an **ET**.
—	**ET**	End a text object, discarding the text matrix.

The specific categories of text-related operators that can appear in a text object are:

- *Text state operators*, described in Section 5.2, "Text State Parameters and Operators."
- *Text-positioning operators*, described in Section 5.3.1, "Text-Positioning Operators."
- *Text-showing operators*, described in Section 5.3.2, "Text-Showing Operators."

The latter two sections also provide further details about the text object parameters described above. The other operators that can appear in a text object are those related to the general graphics state, color, and marked content, as shown in Figure 4.1 on page 167.

Note: *If a content stream does not contain any text, the* **Text** *procedure set may be omitted (see Section 10.1, "Procedure Sets"). In those circumstances, no text operators (including operators that merely set the text state) may be present in the content stream, since those operators are defined in the same procedure set.*

Note: *Although text objects cannot be statically nested, text might be shown using a Type 3 font whose glyph descriptions include any graphics objects, including another text object. Likewise, the current color might be a tiling pattern whose pattern cell includes a text object.*

5.3.1 Text-Positioning Operators

Text space is the coordinate system in which text is shown. It is defined by the text matrix, T_m, and the text state parameters T_{fs}, T_h, and T_{rise}, which together determine the transformation from text space to user space. Specifically, the origin of the first glyph shown by a text-showing operator will be placed at the origin of text space. If text space has been translated, scaled, or rotated, then the position, size, or orientation of the glyph in user space will be correspondingly altered.

At the beginning of a text object, T_m is the identity matrix, so the origin of text space is initially the same as that of user space. The *text-positioning operators*, described in Table 5.5, alter T_m and thereby control the placement of glyphs that are subsequently painted. Also, the *text-showing operators*, described in Table 5.6 in the next section, update T_m (by altering its *e* and *f* translation components) to take into account the horizontal or vertical displacement of each glyph painted as well as any character or word spacing parameters in the text state.

TABLE 5.5 Text-positioning operators

OPERANDS	OPERATOR	DESCRIPTION
t_x t_y	**Td**	Move to the start of the next line, offset from the start of the current line by (t_x, t_y). t_x and t_y are numbers expressed in unscaled text space units. More precisely, this operator performs the following assignments: $$T_m = T_{lm} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \times T_{lm}$$
t_x t_y	**TD**	Move to the start of the next line, offset from the start of the current line by (t_x, t_y). As a side effect, this operator sets the leading parameter in the text state. This operator has the same effect as the code $-t_y$ TL t_x t_y Td

OPERANDS	OPERATOR	DESCRIPTION
a b c d e f	**Tm**	Set the text matrix, T_m, and the text line matrix, T_{lm}, as follows: $$T_m = T_{lm} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$ The operands are all numbers, and the initial value for T_m and T_{lm} is the identity matrix, `[1 0 0 1 0 0]`. Although the operands specify a matrix, they are passed to **Tm** as six separate numbers, not as an array. The matrix specified by the operands is not concatenated onto the current text matrix, but replaces it.
—	**T***	Move to the start of the next line. This operator has the same effect as the code `0` T_l `Td` where T_l is the current leading parameter in the text state.

Additionally, a text object keeps track of a text line matrix, T_{lm}, which captures the value of T_m at the beginning of a line of text. This is convenient for aligning evenly spaced lines of text. The text-positioning and text-showing operators read and set T_{lm} on specific occasions mentioned in Tables 5.5 and 5.6.

Note: *The text-positioning operators can appear only within text objects.*

5.3.2 Text-Showing Operators

The *text-showing operators* (Table 5.6) show text on the page, repositioning text space as they do so. All of the operators interpret the text string and apply the text state parameters as described below.

TABLE 5.6 Text-showing operators

OPERANDS	OPERATOR	DESCRIPTION
string	**Tj**	Show a text string.
string	**'**	Move to the next line and show a text string. This operator has the same effect as the code `T*` *string* `Tj`

OPERANDS	OPERATOR	DESCRIPTION
a_w a_c string	**"**	Move to the next line and show a text string, using a_w as the word spacing and a_c as the character spacing (setting the corresponding parameters in the text state). a_w and a_c are numbers expressed in unscaled text space units. This operator has the same effect as the code a_w Tw a_c Tc *string* '
array	**TJ**	Show one or more text strings, allowing individual glyph positioning (see implementation note 50 in Appendix H). Each element of *array* can be a string or a number. If the element is a string, this operator shows the string. If it is a number, the operator adjusts the text position by that amount; that is, it translates the text matrix, T_m. The number is expressed in thousandths of a unit of text space (see Section 5.3.3, "Text Space Details," and implementation note 51 in Appendix H). This amount is *subtracted* from the current horizontal or vertical coordinate, depending on the writing mode. In the default coordinate system, a positive adjustment has the effect of moving the next glyph painted either to the left or down by the given amount. Figure 5.11 shows an example of the effect of passing offsets to **TJ**.

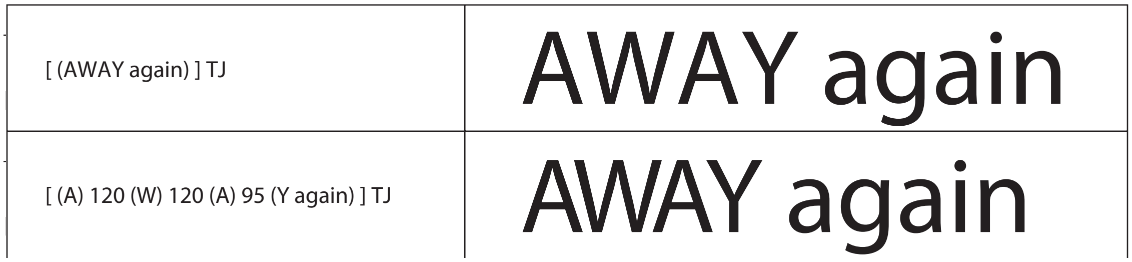

```
[ (AWAY again) ] TJ
[ (A) 120 (W) 120 (A) 95 (Y again) ] TJ
```

FIGURE 5.11 *Operation of the* ***TJ*** *operator in horizontal writing*

Note: *The text-showing operators can appear only within text objects.*

A string operand of a text-showing operator is interpreted as a sequence of character codes identifying the glyphs to be painted. With most font types, each byte of the string is treated as a separate character code. The character code is then looked up in the font's encoding to select the glyph, as described in Section 5.5.5, "Character Encoding."

Beginning with PDF 1.2, a string may be shown in a composite font that uses multiple-byte codes to select some of its glyphs. In that case, one or more consecutive bytes of the string are treated as a single character code. The code lengths and the mappings from codes to glyphs are defined in a data structure called a *CMap*, described in Section 5.6, "Composite Fonts."

The strings must conform to the syntax for string objects. When a string is written by enclosing the data in parentheses, bytes whose values are the same as those of the ASCII characters left parenthesis (40), right parenthesis (41), and backslash (92) must be preceded by a backslash character. All other byte values between 0 and 255 may be used in a string object. These rules apply to each individual byte in a string object, whether the string is interpreted by the text-showing operators as single-byte or multiple-byte character codes.

Strings presented to the text-showing operators may be of any length—even a single character code per string—and may be placed on the page in any order. The grouping of glyphs into strings has no significance for the display of text; showing multiple glyphs with one invocation of a text-showing operator such as **Tj** produces the same results as showing them with a separate invocation for each glyph. However, there the performance of text searching (and other text extraction operations) is significantly better if the text strings are as long as possible and are shown in natural reading order.

Note: *In some cases, the text that is extracted can vary depending on the grouping of glyphs into strings. See, for example, "Reverse-Order Show Strings" on page 759.*

5.3.3 Text Space Details

As stated in Section 5.3.1, "Text-Positioning Operators," text is shown in *text space*, which is defined by the combination of the text matrix, T_m, and the text state parameters T_{fs}, T_h, and T_{rise}. This determines how text coordinates are transformed into user space. Both the glyph's shape and its displacement (horizontal or vertical) are interpreted in text space.

Note: *Glyphs are actually defined in glyph space, whose definition varies according to the font type as discussed in Section 5.1.3, "Glyph Positioning and Metrics." Glyph coordinates are first transformed from glyph space to text space before being subjected to the transformations described below.*

The entire transformation from text space to device space can be represented by a *text rendering matrix*, T_{rm}:

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

T_{rm} is a temporary matrix; conceptually, it is recomputed before each glyph is painted during a text-showing operation.

After the glyph is painted, the text matrix is updated according to the glyph displacement and any spacing parameters that apply. First, a combined displacement is computed, denoted by either t_x (in horizontal writing mode) or t_y (in vertical writing mode); the variable corresponding to the other writing mode is set to 0.

$$t_x = \left(\left(w0 - \frac{T_j}{1000} \right) \times T_{fs} + T_c + T_w \right) \times T_h$$

$$t_y = \left(w1 - \frac{T_j}{1000} \right) \times T_{fs} + T_c + T_w$$

where

w0 and *w1* are the glyph's horizontal and vertical displacements

T_j is a position adjustment specified by a number in a **TJ** array, if any

T_{fs} and T_h are the current text font size and horizontal scaling parameters in the graphics state

T_c and T_w are the current character and word spacing parameters in the graphics state, if applicable

The text matrix is then updated as follows:

$$T_m = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \times T_m$$

5.4 Introduction to Font Data Structures

A font is represented in PDF as a dictionary specifying the type of font, its PostScript name, its encoding, and information that can be used to provide a substitute when the font program is not available. Optionally, the font program itself can be embedded as a stream object in the PDF file.

The font types are distinguished by the **Subtype** entry in the font dictionary. Table 5.7 lists the font types defined in PDF. Type 0 fonts are called *composite fonts*; other types of font are called *simple fonts*. In addition to fonts, PDF supports two classes of font-related objects, called *CIDFonts* and *CMaps*, described in Section 5.6.1, "CID-Keyed Fonts Overview." CIDFonts are listed in Table 5.7 because, like fonts, they are collections of glyphs; however, a CIDFont is never used directly, but only as a component of a Type 0 font.

TABLE 5.7 Font types

TYPE	SUBTYPE VALUE	DESCRIPTION
Type 0	**Type0**	*(PDF 1.2)* A *composite* font—a font composed of other fonts, organized hierarchically (see Section 5.6, "Composite Fonts")
Type 1	**Type1**	A font that defines glyph shapes by using a special encoded format (see Section 5.5.1, "Type 1 Fonts")
	MMType1	A *multiple master* font—an extension of the Type 1 font that allows the generation of a wide variety of typeface styles from a single font (see "Multiple Master Fonts" on page 378)
Type 3	**Type3**	A font that defines glyphs with streams of PDF graphics operators (see Section 5.5.4, "Type 3 Fonts")
TrueType	**TrueType**	A font based on the TrueType font format (see Section 5.5.2, "TrueType Fonts")
CIDFont	**CIDFontType0**	*(PDF 1.2)* A CIDFont whose glyph descriptions are based on Type 1 font technology (see Section 5.6.3, "CIDFonts")
	CIDFontType2	*(PDF 1.2)* A CIDFont whose glyph descriptions are based on TrueType font technology (see Section 5.6.3, "CIDFonts")

For all font types, the term *font dictionary* refers to a PDF dictionary containing information about the font; likewise, a *CIDFont dictionary* contains information about a CIDFont. Except for Type 3, this dictionary is distinct from the *font pro-*

gram that defines the font's glyphs. That font program may be embedded in the PDF file as a stream object or be obtained from some external source.

Note: *This terminology differs from that used in the PostScript language. In PostScript, a font dictionary is a PostScript data structure that is created as a direct result of interpreting a font program. In PDF, a font program is always treated as if it were a separate file, even if its contents are embedded in the PDF file. The font program is interpreted by a specialized font interpreter when necessary; its contents never materialize as PDF objects.*

Most font programs (and related programs, such as CIDFonts and CMaps) conform to external specifications, such as the *Adobe Type 1 Font Format*. This book does not include those specifications. See the Bibliography for more information about the specifications mentioned in this chapter.

The most predictable and dependable results are produced when all font programs used to show text are embedded in the PDF file. The following sections describe precisely how to do so. On the other hand, if a PDF file refers to font programs that are not embedded, the results depend on the availability of fonts in the viewer application's environment. The following sections specify some conventions for referring to external font programs; however, some details of font naming, font substitution, and glyph selection are implementation-dependent and may vary among different viewer applications and operating system environments.

5.5 Simple Fonts

There are several types of simple font, all of which have the following properties:

- Glyphs in the font are selected by single-byte character codes obtained from a string that is shown by the text-showing operators. Logically, these codes index into a table of 256 glyphs; the mapping from codes to glyphs is called the font's *encoding*. Each font program has a built-in encoding. Under some circumstances, the encoding can be altered by means described in Section 5.5.5, "Character Encoding."
- Each glyph has a single set of metrics, including a *horizontal displacement* or *width*, as described in Section 5.1.3, "Glyph Positioning and Metrics." That is, simple fonts support only horizontal writing mode.

- Except for Type 3 fonts and certain standard Type 1 fonts, every font dictionary contains a subsidiary dictionary, the *font descriptor*, containing fontwide metrics and other attributes of the font; see Section 5.7, "Font Descriptors." Among those attributes is an optional *font file* stream containing the font program itself.

5.5.1 Type 1 Fonts

A Type 1 font program is a stylized PostScript program that describes glyph shapes. It uses a compact encoding for the glyph descriptions, and it includes hint information that enables high-quality rendering even at small sizes and low resolutions. Details on this format are provided in a separate book, *Adobe Type 1 Font Format*. An alternative, more compact but functionally equivalent representation of a Type 1 font program is documented in Adobe Technical Note #5176, *The Compact Font Format Specification*.

***Note:** Although a Type 1 font program uses PostScript language syntax, using it does not require a full PostScript interpreter; a specialized Type 1 font interpreter suffices.*

A Type 1 font dictionary contains the entries listed in Table 5.8. Some entries are optional for the standard 14 fonts listed under "Standard Type 1 Fonts" on page 378, but are required otherwise.

TABLE 5.8 Entries in a Type 1 font dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Font** for a font dictionary.
Subtype	name	*(Required)* The type of font; must be **Type1** for a Type 1 font.
Name	name	*(Required in PDF 1.0; optional otherwise)* The name by which this font is referenced in the **Font** subdictionary of the current resource dictionary. ***Note:** This entry is obsolescent and its use is no longer recommended. (See implementation note 52 in Appendix H.)*

KEY	TYPE	VALUE
BaseFont	name	*(Required)* The PostScript name of the font. For Type 1 fonts, this is usually the value of the **FontName** entry in the font program; for more information, see Section 5.2 of the *PostScript Language Reference*, Third Edition. The PostScript name of the font can be used to find the font's definition in the viewer application or its environment. It is also the name that will be used when printing to a PostScript output device.
FirstChar	integer	*(Required except for the standard 14 fonts)* The first character code defined in the font's **Widths** array.
LastChar	integer	*(Required except for the standard 14 fonts)* The last character code defined in the font's **Widths** array.
Widths	array	*(Required except for the standard 14 fonts; indirect reference preferred)* An array of (**LastChar** − **FirstChar** + 1) widths, each element being the glyph width for the character code that equals **FirstChar** plus the array index. For character codes outside the range **FirstChar** to **LastChar**, the value of **MissingWidth** from the **FontDescriptor** entry for this font is used. The glyph widths are measured in units in which 1000 units corresponds to 1 unit in text space. These widths must be consistent with the actual widths given in the font program itself. (See implementation note 53 in Appendix H.) For more information on glyph widths and other glyph metrics, see Section 5.1.3, "Glyph Positioning and Metrics."
FontDescriptor	dictionary	*(Required except for the standard 14 fonts; must be an indirect reference)* A font descriptor describing the font's metrics other than its glyph widths (see Section 5.7, "Font Descriptors"). ***Note:*** *For the standard 14 fonts, the entries* ***FirstChar****,* ***LastChar****,* ***Widths****, and* ***FontDescriptor*** *must either all be present or all absent. Ordinarily, they are absent; specifying them enables a standard font to be overridden (see "Standard Type 1 Fonts," below).*
Encoding	name or dictionary	*(Optional)* A specification of the font's character encoding, if different from its built-in encoding. The value of **Encoding** may be either the name of a predefined encoding (**MacRomanEncoding**, **MacExpertEncoding**, or **WinAnsiEncoding**, as described in Appendix D) or an encoding dictionary that specifies differences from the font's built-in encoding or from a specified predefined encoding (see Section 5.5.5, "Character Encoding").
ToUnicode	stream	*(Optional; PDF 1.2)* A stream containing a CMap file that maps character codes to Unicode values (see Section 5.9, "Extraction of Text Content").

Example 5.6 shows the font dictionary for the Adobe Garamond® Semibold font. The font has an encoding dictionary (object 25), although neither the encoding dictionary nor the font descriptor (object 7) is shown in the example.

Example 5.6

```
14 0 obj
   << /Type /Font
      /Subtype /Type1
      /BaseFont /AGaramond–Semibold
      /FirstChar 0
      /LastChar 255
      /Widths 21 0 R
      /FontDescriptor 7 0 R
      /Encoding 25 0 R
   >>
endobj

21 0 obj
   [ 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
     255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
     255 280 438 510 510 868 834 248 320 320 420 510 255 320 255 347
     510 510 510 510 510 510 510 510 510 510 255 255 510 510 510 330
     781 627 627 694 784 580 533 743 812 354 354 684 560 921 780 792
     588 792 656 504 682 744 650 968 648 590 638 320 329 320 510 500
     380 420 510 400 513 409 301 464 522 268 259 484 258 798 533 492
     516 503 349 346 321 520 434 684 439 448 390 320 255 320 510 255
     627 627 694 580 780 792 744 420 420 420 420 420 420 402 409 409
     409 409 268 268 268 268 533 492 492 492 492 492 520 520 520 520
     486 400 510 510 506 398 520 555 800 800 1044 360 380 549 846 792
     713 510 549 549 510 522 494 713 823 549 274 354 387 768 615 496
     330 280 510 549 510 549 612 421 421 1000 255 627 627 792 1016 730
     500 1000 438 438 248 248 510 494 448 590 100 510 256 256 539 539
     486 255 248 438 1174 627 580 627 580 580 354 354 354 354 792 792
     790 792 744 744 744 268 380 380 380 380 380 380 380 380 380 380
   ]
endobj
```

Standard Type 1 Fonts

The PostScript names of 14 Type 1 fonts, known as the *standard fonts*, are as follows:

Times–Roman	Helvetica	Courier	Symbol
Times–Bold	Helvetica–Bold	Courier–Bold	ZapfDingbats
Times–Italic	Helvetica–Oblique	Courier–Oblique	
Times–BoldItalic	Helvetica–BoldOblique	Courier–BoldOblique	

These fonts, or their font metrics and suitable substitution fonts, are required to be available to the viewer application. The character sets and encodings for these fonts are given in Appendix D. The Adobe font metrics (AFM) files for the standard 14 fonts are available from the ASN Web site (see the Bibliography). For more information on font metrics, see Adobe Technical Note #5004, *Adobe Font Metrics File Format Specification*.

Ordinarily, a font dictionary that refers to one of the standard fonts should omit the **FirstChar**, **LastChar**, **Widths**, and **FontDescriptor** entries. However, it is permissible to override a standard font by including these entries and embedding the font program in the PDF file. (See implementation note 54 in Appendix H.)

Multiple Master Fonts

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 program. This is accomplished through the presence of various design dimensions in the font. Examples of design dimensions are weight (light to extra-bold) and width (condensed to expanded). Coordinates along these design dimensions (such as the degree of boldness) are specified by numbers. A particular choice of numbers selects an *instance* of the multiple master font. Adobe Technical Note #5015, *Type 1 Font Format Supplement*, describes multiple master fonts in detail.

The font dictionary for a multiple master font instance has the same entries as a Type 1 font dictionary (Table 5.8 on page 375), except note the following:

- The value of **Subtype** is **MMType1**.

- If the PostScript name of the instance contains spaces, the spaces are replaced by underscores in the value of **BaseFont**. For instance, as illustrated in Example 5.7, the name "MinionMM 366 465 11 " (which ends with a space character) becomes /MinionMM_366_465_11_.

Example 5.7

```
7 0 obj
    << /Type /Font
        /Subtype /MMType1
        /BaseFont /MinionMM_366_465_11_
        /FirstChar 32
        /LastChar 255
        /Widths 19 0 R
        /FontDescriptor 6 0 R
        /Encoding 5 0 R
    >>
endobj

19 0 obj
    [ 187 235 317 430 427 717 607 168 326 326 421 619 219 317 219 282 427
      …Omitted data…
      569 0 569 607 607 607 239 400 400 400 400 253 400 400 400 400 400
    ]
endobj
```

This example illustrates a convention for including the numeric values of the design coordinates as part of the instance's **BaseFont** name. This convention is commonly used for accessing multiple master font instances from an external source in the viewer application's environment; it is documented in Adobe Technical Note #5088, *Font Naming Issues*. However, this convention is not prescribed as part of the PDF specification. In particular, if the font program for this instance is embedded in the PDF file, it must be an ordinary Type 1 font program, not a multiple master font program. This font program is called a *snapshot* of the multiple master font instance, incorporating the chosen values of the design coordinates.

5.5.2 TrueType Fonts

The *TrueType* font format was developed by Apple Computer, Inc., and has been adopted as a standard font format for the Microsoft Windows operating system. Specifications for the TrueType font file format are available in Apple's *TrueType Reference Manual* and Microsoft's *TrueType 1.0 Font Files Technical Specification*.

Note: *A TrueType font program can be embedded directly in a PDF file as a stream object. The Type 42 font format that is defined for PostScript does not apply to PDF.*

A TrueType font dictionary can contain the same entries as a Type 1 font dictionary (Table 5.8 on page 375), except note the following:

- The value of **Subtype** is **TrueType**.
- The value of **BaseFont** is derived differently, as described below.
- The value of **Encoding** is subject to limitations that are described in Section 5.5.5, "Character Encoding."

The PostScript name for the value of **BaseFont** is determined in one of two ways:

- Use the PostScript name that is an optional entry in the "name" table of the TrueType font itself.
- In the absence of such an entry in the "name" table, derive a PostScript name from the name by which the font is known in the host operating system: on a Windows system, it is based on the lfFaceName field in a LOGFONT structure; in the Mac OS, it is based on the name of the FOND resource. If the name contains any spaces, the spaces are removed.

If the font in a source document uses a bold or italic style, but there is no font data for that style, the host operating system will synthesize the style. In this case, a comma and the style name (one of Bold, Italic, or BoldItalic) are appended to the font name. For example, for a TrueType font that is a bold variant of the New York font, the **BaseFont** value is written as /NewYork,Bold (as illustrated in Example 5.8).

Example 5.8

```
17 0 obj
   << /Type /Font
      /Subtype /TrueType
      /BaseFont /NewYork,Bold
      /FirstChar 0
      /LastChar 255
      /Widths 23 0 R
      /FontDescriptor 7 0 R
      /Encoding /MacRomanEncoding
   >>
endobj

23 0 obj
   [ 0 333 333 333 333 333 333 333 0 333 333 333 333 333 333 333 333 333
     …Omitted data…
     803 790 803 780 780 780 340 636 636 636 636 636 636 636 636 636 636
   ]
endobj
```

Note that for CJK (Chinese, Japanese, and Korean) fonts, the host font system's "font name" is often encoded in the host operating system's script. For instance, a Japanese font may have a name that is written in Japanese using some (unidentified) Japanese encoding. Thus, TrueType font names may contain multiple-byte character codes, each of which requires multiple characters to represent in a PDF name object (using the # notation to quote special characters as needed).

5.5.3 Font Subsets

PDF 1.1 permits documents to include subsets of Type 1 and TrueType fonts. The font and font descriptor that describe a font subset are slightly different from those of ordinary fonts. These differences allow an application to recognize font subsets and to merge documents containing different subsets of the same font. (For more information on font descriptors, see Section 5.7, "Font Descriptors.")

For a font subset, the PostScript name of the font—the value of the font's **BaseFont** entry and the font descriptor's **FontName** entry—begins with a *tag* followed by a plus sign (+). The tag consists of exactly six uppercase letters; the choice of letters is arbitrary, but different subsets in the same PDF file must have different tags. For example, EOODIA+Poetica is the name of a subset of Poetica®, a Type 1 font. (See implementation note 55 in Appendix H.)

5.5.4 Type 3 Fonts

Type 3 fonts differ from the other fonts supported by PDF. A Type 3 font dictionary defines the font itself, while the other font dictionaries simply contain information *about* the font and refer to a separate font program for the actual glyph descriptions. In Type 3 fonts, glyphs are defined by streams of PDF graphics operators. These streams are associated with character names. A separate encoding entry maps character codes to the appropriate character names for the glyphs.

Type 3 fonts are more flexible than Type 1 fonts, because the glyph descriptions may contain arbitrary PDF graphics operators. However, Type 3 fonts have no hinting mechanism for improving output at small sizes or low resolutions. A Type 3 font dictionary contains the entries listed in Table 5.9.

TABLE 5.9 Entries in a Type 3 font dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Font** for a font dictionary.
Subtype	name	*(Required)* The type of font; must be **Type3** for a Type 3 font.
Name	name	*(Required in PDF 1.0; optional otherwise)* See Table 5.8 on page 375.
FontBBox	rectangle	*(Required)* A rectangle (see Section 3.8.4, “Rectangles”), expressed in the glyph coordinate system, specifying the *font bounding box*. This 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 filled. If all four elements of the rectangle are zero, no assumptions are made based on the font bounding box. If any element 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.
FontMatrix	array	*(Required)* An array of six numbers specifying the *font matrix*, mapping glyph space to text space (see Section 5.1.3, “Glyph Positioning and Metrics”). A common practice is to define glyphs in terms of a 1000-unit glyph coordinate system, in which case the font matrix is [0.001 0 0 0.001 0 0].

KEY	**TYPE**	**VALUE**
CharProcs	dictionary	*(Required)* A dictionary in which each key is a character name and the value associated with that key is a content stream that constructs and paints the glyph for that character. The stream must include as its first operator either **d0** or **d1**. This is followed by operators describing one or more graphics objects, which may include path, text, or image objects. See below for more details about Type 3 glyph descriptions.
Encoding	name or dictionary	*(Required)* An encoding dictionary whose **Differences** array specifies the complete character encoding for this font (see Section 5.5.5, “Character Encoding”; also see implementation note 56 in Appendix H).
FirstChar	integer	*(Required)* The first character code defined in the font’s **Widths** array.
LastChar	integer	*(Required)* The last character code defined in the font’s **Widths** array.
Widths	array	*(Required; indirect reference preferred)* An array of (**LastChar** − **FirstChar** + 1) widths, each element being the glyph width for the character code that equals **FirstChar** plus the array index. For character codes outside the range **FirstChar** to **LastChar**, the width is 0. These widths are interpreted in glyph space as specified by **FontMatrix** (unlike the widths of a Type 1 font, which are in thousandths of a unit of text space). ***Note:** If **FontMatrix** specifies a rotation, only the horizontal component of the transformed width is used. That is, the resulting displacement is always horizontal in text space, as is the case for all simple fonts.*
FontDescriptor	dictionary	*(Required in Tagged PDF documents; must be an indirect reference)* A font descriptor describing the font’s default metrics other than its glyph widths (see Section 5.7, “Font Descriptors”).
Resources	dictionary	*(Optional but strongly recommended; PDF 1.2)* A list of the named resources, such as fonts and images, required by the glyph descriptions in this font (see Section 3.7.2, “Resource Dictionaries”). If any glyph descriptions refer to named resources but this dictionary is absent, the names are looked up in the resource dictionary of the page on which the font is used. (See implementation note 57 in Appendix H.)
ToUnicode	stream	*(Optional; PDF 1.2)* A stream containing a CMap file that maps character codes to Unicode values (see Section 5.9, “Extraction of Text Content”).

For each character code shown by a text-showing operator using a Type 3 font, the viewer application does the following:

1. Looks up the character code in the font's **Encoding** entry, as described in Section 5.5.5, "Character Encoding," to obtain a character name.
2. Looks up the character name in the font's **CharProcs** dictionary to obtain a stream object containing a glyph description. (If the name is not present as a key in **CharProcs**, no glyph is painted.)
3. Invokes the glyph description, as described below. The graphics state is saved before this invocation and restored afterward, so any changes the glyph description makes to the graphics state do not persist after it finishes.

When the glyph description begins execution, the current transformation matrix (CTM) is the concatenation of the font matrix (**FontMatrix** in the current font dictionary) and the text space that was in effect at the time the text-showing operator was invoked (see Section 5.3.3, "Text Space Details"). 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. The glyph description 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 text position.

Aside from the CTM, the graphics state is inherited from the environment of the text-showing operator that caused the glyph description to be invoked. To ensure predictable results, the glyph description must initialize any graphics state parameters on which it depends. In particular, if it invokes the **S** (stroke) operator, it 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 the text-showing operators are designed to paint glyphs with the current color.

The glyph description must execute one of the operators described in Table 5.10 to pass width and bounding box information to the font machinery. This must precede the execution of any path construction or path-painting operators describing the glyph.

Note: *Type 3 fonts in PDF are very similar to those in PostScript. Some of the information provided in Type 3 font dictionaries and glyph descriptions, while seemingly redundant or unnecessary, is nevertheless required for correct results when a*

PDF viewer application prints to a PostScript output device. This applies particularly to the operands of the ***d0*** *and* ***d1*** *operators, which in PostScript are named* ***setcharwidth*** *and* ***setcachedevice****. For further explanation, see Section 5.7 of the* PostScript Language Reference, *Third Edition.*

TABLE 5.10 Type 3 font operators

OPERANDS	OPERATOR	DESCRIPTION
w_x w_y	**d0**	Set width information for the glyph and declare that the glyph description specifies both its shape and its color. (Note that this operator name ends in the digit **0**.) w_x specifies the horizontal displacement in the glyph coordinate system; it must be consistent with the corresponding width in the font's **Widths** array. w_y must be 0 (see Section 5.1.3, "Glyph Positioning and Metrics"). This operator is permitted only in a content stream appearing in a Type 3 font's **CharProcs** dictionary. It is typically used only if the glyph description executes operators to set the color explicitly.
w_x w_y ll_x ll_y ur_x ur_y	**d1**	Set width and bounding box information for the glyph and declare that the glyph description specifies only shape, not color. (Note that this operator name ends in the digit **1**.) w_x specifies the horizontal displacement in the glyph coordinate system; it must be consistent with the corresponding width in the font's **Widths** array. w_y must be 0 (see Section 5.1.3, "Glyph Positioning and Metrics"). 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 axes of the glyph coordinate system, that completely encloses all marks placed on the page as a result of executing the glyph's description. 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. A glyph description that begins with the **d1** operator should not execute any operators that set the color (or other color-related parameters) in the graphics state; any use of such operators will be ignored. The glyph description is executed solely to determine the glyph's shape; its color is determined by the graphics state in effect each time this glyph is painted by a text-showing operator. For the same reason, the glyph description may not include an image; however, an image mask is acceptable, since it merely defines a region of the page to be painted with the current color. This operator is permitted only in a content stream appearing in a Type 3 font's **CharProcs** dictionary.

Example of a Type 3 Font

Example 5.9 shows the definition of a Type 3 font with only two glyphs—a filled square and a filled triangle, selected by the character codes a and b. Figure 5.12 shows the result of showing the string (ababab) using this font.

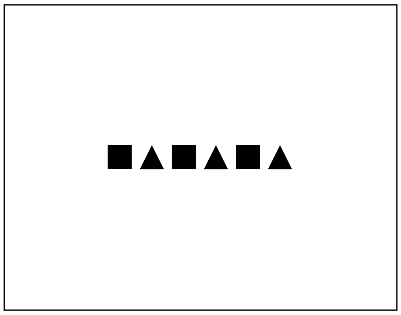


FIGURE 5.12 *Output from Example 5.9*

Example 5.9

```
4 0 obj
    << /Type /Font
       /Subtype /Type3
       /FontBBox [0 0 750 750]
       /FontMatrix [0.001 0 0 0.001 0 0]
       /CharProcs 10 0 R
       /Encoding 9 0 R
       /FirstChar 97
       /LastChar 98
       /Widths [1000 1000]
    >>
endobj

9 0 obj
    << /Type /Encoding
       /Differences [97 /square /triangle]
    >>
endobj
```

```
10 0 obj
   << /square 11 0 R
       /triangle 12 0 R
   >>
endobj

11 0 obj
   << /Length 39 >>
stream
   1000 0 0 0 750 750 d1
   0 0 750 750 re
   f
endstream
endobj

12 0 obj
   << /Length 48 >>
stream
   1000 0 0 0 750 750 d1
   0 0 m
   375 750 l
   750 0 l
   f
endstream
endobj
```

5.5.5 Character Encoding

A font's *encoding* is the association between character codes (obtained from text strings that are shown) and glyph descriptions. This section describes the character encoding scheme used with simple PDF fonts. Composite fonts (Type 0) use a different character mapping algorithm, as discussed in Section 5.6, "Composite Fonts."

Except for Type 3 fonts, every font program has a built-in encoding. Under certain circumstances, a PDF font dictionary can change a font's built-in encoding to match the requirements of the application generating the text being shown. 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*, often referred to as **StandardEncoding**. The name **StandardEncoding** has no special meaning in PDF, but this encoding does play a role as a default encoding (as shown in Table 5.11 below). The regular encodings used for Latin-text fonts on Mac OS and Windows systems are named **MacRomanEncoding** and **WinAnsiEncoding**, respectively. Additionally, an encoding named **MacExpertEncoding** is used with "expert" fonts that contain additional characters useful for sophisticated typography. Complete details of these encodings and of the characters present in typical fonts are provided in Appendix D.

In PDF, a font is classified as either *nonsymbolic* or *symbolic* according to whether or not all of its characters are members of the Adobe standard Latin character set. This is indicated by flags in the font descriptor; see Section 5.7.1, "Font Descriptor Flags." Symbolic fonts contain other character sets, to which the encodings mentioned above ordinarily do not apply. Such font programs have built-in encodings that are usually unique to each font. The standard 14 fonts include two symbolic fonts, Symbol and ZapfDingbats, whose encodings and character sets are documented in Appendix D.

A font program's built-in encoding can be overridden or altered by including an **Encoding** entry in the PDF font dictionary. The possible encoding modifications depend on the font type, as discussed below. The value of the **Encoding** entry is either a named encoding (the name of one of the predefined encodings **MacRomanEncoding**, **MacExpertEncoding**, or **WinAnsiEncoding**) or an *encoding dictionary.* An encoding dictionary contains the entries listed in Table 5.11.

TABLE 5.11 Entries in an encoding dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Encoding** for an encoding dictionary.
BaseEncoding	name	*(Optional)* The *base encoding*—that is, the encoding from which the **Differences** entry (if present) describes differences—specified as the name of a predefined encoding **MacRomanEncoding**, **MacExpertEncoding**, or **WinAnsiEncoding** (see Appendix D). If this entry is absent, the **Differences** entry describes differences from an implicit base encoding. For a font program that is embedded in the PDF file, the implicit base encoding is the font program's built-in encoding, as described above and further elaborated in the sections on specific font types below. Otherwise, for a nonsymbolic font, it is **StandardEncoding**, and for a symbolic font, it is the font's built-in encoding.
Differences	array	*(Optional; not recommended with TrueType fonts)* An array describing the differences from the encoding specified by **BaseEncoding** or, if **BaseEncoding** is absent, from an implicit base encoding. The **Differences** array is described below.

The value of the **Differences** entry is an array of character codes and character names organized as follows:

$code_1$ $name_{1,1}$ $name_{1,2}$ …
$code_2$ $name_{2,1}$ $name_{2,2}$ …
…
$code_n$ $name_{n,1}$ $name_{n,2}$ …

Each code is the first index in a sequence of character codes to be changed. The first character name after the code becomes the name corresponding to that code. Subsequent names replace consecutive code indices until the next code appears in the array or the array ends. These sequences may be specified in any order but should not overlap.

For example, in the encoding dictionary in Example 5.10, the name quotesingle (') is associated with character code 39, Adieresis (Ä) with code 128, Aring (Å) with 129, and trademark (™) with 170.

Example 5.10

```
25 0 obj
   <<  /Type /Encoding
       /Differences
         [    39 /quotesingle
              96 /grave
             128 /Adieresis /Aring /Ccedilla /Eacute /Ntilde /Odieresis /Udieresis
                 /aacute /agrave /acircumflex /adieresis /atilde /aring /ccedilla
                 /eacute /egrave /ecircumflex /edieresis /iacute /igrave /icircumflex
                 /idieresis /ntilde /oacute /ograve /ocircumflex /odieresis /otilde
                 /uacute /ugrave /ucircumflex /udieresis /dagger /degree /cent
                 /sterling /section /bullet /paragraph /germandbls /registered
                 /copyright /trademark /acute /dieresis
             174 /AE /Oslash
             177 /plusminus
             180 /yen /mu
             187 /ordfeminine /ordmasculine
             190 /ae /oslash /questiondown /exclamdown /logicalnot
             196 /florin
             199 /guillemotleft /guillemotright /ellipsis
             203 /Agrave /Atilde /Otilde /OE /oe /endash /emdash /quotedblleft
                 /quotedblright /quoteleft /quoteright /divide
             216 /ydieresis /Ydieresis /fraction /currency /guilsinglleft /guilsinglright
                 /fi /fl /daggerdbl /periodcentered /quotesinglbase /quotedblbase
                 /perthousand /Acircumflex /Ecircumflex /Aacute /Edieresis /Egrave
                 /Iacute /Icircumflex /Idieresis /Igrave /Oacute /Ocircumflex
             241 /Ograve /Uacute /Ucircumflex /Ugrave /dotlessi /circumflex /tilde
                 /macron /breve /dotaccent /ring /cedilla /hungarumlaut /ogonek
                 /caron
         ]
   >>
endobj
```

By convention, the name .notdef can be used to indicate that there is no character name associated with a given character code.

Encodings for Type 1 Fonts

A Type 1 font program's glyph descriptions are keyed by character *names*, not by character *codes*. Character names are ordinary PDF 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. A Type 1 font's built-in encoding is defined by an **Encoding** array that is part of the font program itself; this is not to be confused with the **Encoding** entry in the PDF font dictionary.

An **Encoding** entry can alter a Type 1 font's mapping from character codes to character names. The **Differences** array can map a code to the name of any glyph description that exists in the font program, whether or not that glyph is referenced by the font's built-in encoding or by the encoding specified in the **BaseEncoding** entry.

All Type 1 font programs contain an actual glyph named .notdef. The effect produced by showing the .notdef glyph is at the discretion of the font designer; in Type 1 font programs produced by Adobe, it is the same as the space character. If an encoding maps to a character name that does not exist in the Type 1 font program, the .notdef glyph is substituted.

Encodings for Type 3 Fonts

A Type 3 font, like Type 1, contains glyph descriptions that are keyed by character names; in this case, they appear as explicit keys in the font's **CharProcs** dictionary. A Type 3 font's mapping from character codes to character names is entirely defined by its **Encoding** entry, which is required in this case.

Encodings for TrueType Fonts

A TrueType font program's built-in encoding maps directly from character codes to glyph descriptions, using an internal data structure called a "cmap" (not to be confused with the CMap described in Section 5.6.4, "CMaps"). This section describes how the PDF font dictionary's **Encoding** entry is used in conjunction with a "cmap" to map from a character code in a string to a glyph description in a TrueType font program.

A "cmap" table may contain one or more subtables that represent multiple encodings intended for use on different platforms (such as Mac OS and Windows). Each subtable is identified by the two numbers, such as (3, 1), that represent a combination of a *platform ID* and a *platform-specific encoding ID*, respectively.

Glyph names are not mandatory in TrueType fonts, although some font programs have an optional "post" table listing glyph names for the glyphs. If the viewer application needs to select glyph descriptions by name, it translates from glyph names to codes in one of the encodings given in the font program's "cmap" table. When there is no character code in the "cmap" that corresponds to a glyph name, the "post" table is used to select a glyph description directly from the glyph name.

Because some aspects of TrueType glyph selection are dependent on the viewer implementation or the operating system, PDF files that use TrueType fonts should follow certain guidelines to ensure predictable behavior across all viewer applications:

- The font program should be embedded.
- A nonsymbolic font should specify **MacRomanEncoding** or **WinAnsiEncoding** as the value of its **Encoding** entry, with no **Differences** array.
- A font used to display glyphs that do not use **MacRomanEncoding** or **WinAnsi-Encoding** should not specify an **Encoding** entry. The font descriptor's Symbolic flag (see Table 5.20) should be set, and its font program's "cmap" table should contain a (1, 0) subtable. It may also contain a (3, 0) subtable; if present, this subtable should map from character codes in the range 0xF000 to 0xF0FF by prepending the single-byte codes in the (1, 0) subtable with 0xF0 and mapping to the corresponding glyph descriptions.

***Note:** Some popular TrueType font programs contain incorrect encoding information. Implementations of TrueType font interpreters have evolved heuristics for dealing with such problems; those heuristics are not described here. For maximum portability, only well-formed TrueType font programs should be used in PDF files. Therefore, a TrueType font program in a PDF file may need to be modified to conform to the guidelines described above.*

The following paragraphs describe the treatment of TrueType font encodings beginning with PDF 1.3, as implemented in Acrobat 5.0 and later viewers. This information does not necessarily apply to earlier versions or implementations.

If the font has a named **Encoding** entry of either **MacRomanEncoding** or **WinAnsiEncoding**, or if the font descriptor's Nonsymbolic flag (see Table 5.20) is set, the viewer creates a table that maps from character codes to glyph names, as follows:

- If the **Encoding** entry is one of the names **MacRomanEncoding** or **WinAnsiEncoding**, the table is initialized with the mappings described in Appendix D.
- If the **Encoding** entry is a dictionary, the table is initialized with the entries from the dictionary's **BaseEncoding** entry (see Table 5.11). Any entries in the **Differences** array are used to update the table. Finally, any undefined entries in the table are filled using **StandardEncoding**.

If a (3, 1) "cmap" subtable (Microsoft Unicode), is present:

- A character code is first mapped to a glyph name using the table described above.
- The glyph name is then mapped to a Unicode value by consulting the *Adobe Glyph List* (see the Bibliography).
- Finally, the Unicode value is mapped to a glyph description according to the (3, 1) subtable.

If there is no (3, 1) subtable, but a (1, 0) subtable (Macintosh® Roman) is present:

- A character code is first mapped to a glyph name using the table described above.
- The glyph name is then mapped back to a character code according to the standard Roman encoding used on Mac OS (see note below).
- Finally, the code is mapped to a glyph description according to the (1, 0) subtable.

In either of the cases above, if the glyph name cannot be mapped as specified, the glyph name is looked up in the font program's "post" table (if one is present) and the associated glyph description is used.

Note: *The standard Roman encoding used on Mac OS is the same as the* ***MacRomanEncoding*** *described in Appendix D, with the addition of following 15 entries and the replacement of the currency glyph with the Euro glyph, as shown in Table 5.12.*

TABLE 5.12 Differences between MacRomanEncoding and Mac OS Roman encoding

NAME	CODE (OCTAL)	CODE (DECIMAL)
notequal	255	173
infinity	260	176
lessequal	262	178
greaterequal	263	179
partialdiff	266	182
summation	267	183
product	270	184
pi	271	185
integral	272	186
Omega	275	189
radical	303	195
approxequal	305	197
Delta	306	198
lozenge	327	215
Euro	333	219
apple	360	240

When the font has no **Encoding** entry, or the font descriptor's Symbolic flag is set (in which case the **Encoding** entry is ignored), the following occurs:

- If the font contains a (3, 0) subtable, the range of character codes must be one of the following: 0x0000 - 0x00FF, 0xF000 - 0xF0FF, 0xF100 - 0xF1FF, or 0xF200 - 0xF2FF. Depending on the range of codes, each byte from the string is prepended with the high byte of the range, to form a two-byte character, which is used to select the associated glyph description from the subtable.
- Otherwise, if the font contains a (1, 0) subtable, single bytes from the string are used to look up the associated glyph descriptions from the subtable.

If a character cannot be mapped in any of the ways described above, the results are implementation-dependent.

5.6 Composite Fonts

A *composite font* is one whose glyphs are obtained from a fontlike object called a *CIDFont*. A composite font is represented by a font dictionary whose **Subtype** value is **Type0**; this is also called a Type 0 font. The Type 0 font is known as the *root font*, while its associated CIDFont is called its *descendant*.

***Note:** Composite fonts in PDF are analogous to composite fonts in PostScript, but with some limitations. In particular, PDF requires that the character encoding be defined by a CMap (described below), which is only one of several encoding methods available in PostScript.Also, PostScript allows a Type 0 font to have multiple descendants, which might themselves be Type 0 fonts. PDF supports only a single descendant, which must be a CIDFont.*

When the current font is composite, the text-showing operators behave differently than with simple fonts. Whereas for simple fonts each byte of a string to be shown selects one glyph, for composite fonts a sequence of one or more bytes can be decoded to select a glyph from the descendant CIDFont. This facility supports the use of very large character sets, such as those for the Chinese, Japanese, and Korean languages. It also simplifies the organization of fonts that have complex encoding requirements.

This section first introduces the architecture of *CID-keyed fonts*, which are the only kind of composite font supported in PDF. Then it describes the *CIDFont* and *CMap* dictionaries, which are the PDF objects that represent the correspondingly named components of a CID-keyed font. Finally, it describes the Type 0 font dictionary, which combines a CIDFont and a CMap to produce a font whose glyphs can be accessed by means of variable-length character codes in a string to be shown.

5.6.1 CID-Keyed Fonts Overview

CID-keyed fonts provide a convenient and efficient method for defining multiple-byte character encodings, fonts with a large number of glyphs, and fonts that incorporate glyphs obtained from other fonts. These capabilities provide great flexibility for representing text in writing systems for languages with large character sets, such as Chinese, Japanese, and Korean (CJK).

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. As mentioned earlier, PDF does not support the entire CID-keyed font architecture, which is independent of PDF; CID-keyed fonts can be used in other environments. 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 only the PDF objects that represent these font programs.

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 some simple fonts. CIDs range from 0 to a maximum value that is subject to an implementation limit (see Table C.1 on page 864).

A *character collection* is an ordered set of all glyphs needed to support one or more popular character sets for a particular language. The order of the glyphs in the character collection determines the CID number for each glyph. Each CID-keyed font must explicitly reference the character collection on which its CID numbers are based; see Section 5.6.2, "CIDSystemInfo Dictionaries."

A *CMap* (character map) file specifies the correspondence between character codes and the CID numbers used to identify glyphs. It is equivalent to the concept of an encoding in simple fonts. Whereas a simple font allows a maximum of 256 glyphs to be encoded and accessible at one time, a CMap can describe a mapping from multiple-byte codes to thousands of glyphs in a large CID-keyed font. For example, it can describe Shift-JIS, one of several widely used encodings for Japanese.

A CMap can reference an entire character collection, a subset, or multiple character collections. It can also reference characters in other fonts by character code or character name. The CMap mapping yields a *font number* (which in PDF is always 0) and a *character selector* (which in PDF is always a CID). 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 text-showing 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 simple fonts, such as Type 1. However, they are identified by CIDs rather than by names, and they are organized differently.

In PDF, the CMap and CIDFont are represented by PDF objects, which are described below. The CMap and CIDFont programs themselves can be either referenced by name or embedded as stream objects in the PDF file. As stated earlier, the external file formats are not documented here, but in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*.

A CID-keyed font, then, is the combination of a CMap with a CIDFont containing glyph descriptions. It is represented as a Type 0 font. It contains an **Encoding** entry whose value is a CMap dictionary, and its **DescendantFonts** entry references the CIDFont dictionary with which the CMap has been combined.

5.6.2 CIDSystemInfo Dictionaries

CIDFont and CMap dictionaries contain a **CIDSystemInfo** entry specifying the character collection assumed by the CIDFont associated with the CMap—that is, the interpretation of the CID numbers used by the CIDFont. A character collection is uniquely identified by the **Registry**, **Ordering**, and **Supplement** entries in the **CIDSystemInfo** dictionary, as described in Table 5.13. Character collections whose **Registry** and **Ordering** values are the same are compatible.

TABLE 5.13 Entries in a CIDSystemInfo dictionary

KEY	TYPE	VALUE
Registry	string	*(Required)* A string identifying the issuer of the character collection—for example, Adobe. For information about assigning a registry identifier, contact the Adobe Solutions Network or consult the ASN Web site (see the Bibliography).
Ordering	string	*(Required)* A string that uniquely names the character collection within the specified registry—for example, Japan1.

KEY	TYPE	VALUE
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 5.6.4, "CMaps."

For proper behavior, the **CIDSystemInfo** entry of a CMap should be compatible with that of the CIDFont or CIDFonts with which it is used. If they are incompatible, the effects produced will be unpredictable.

5.6.3 CIDFonts

A CIDFont program contains glyph descriptions that are accessed using a CID as the character selector. There are two types of CIDFont. A Type 0 CIDFont contains glyph descriptions based on Adobe's Type 1 font format, whereas those in a Type 2 CIDFont are based on the TrueType font format.

A CIDFont dictionary is a PDF object that contains information about a CIDFont program. Although its **Type** value is **Font**, a CIDFont is not actually a font. It does not have an **Encoding** entry, it cannot be listed in the **Font** subdictionary of a resource dictionary, and it cannot be used as the operand of the **Tf** operator. It is used only as a descendant of a Type 0 font. The CMap in the Type 0 font is what defines the encoding that maps character codes to CIDs in the CIDFont. Table 5.14 lists the entries in a CIDFont dictionary.

TABLE 5.14 Entries in a CIDFont dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Font** for a CIDFont dictionary.
Subtype	name	*(Required)* The type of CIDFont; **CIDFontType0** or **CIDFontType2**.

KEY	TYPE	VALUE
BaseFont	name	*(Required)* The PostScript name of the CIDFont. For Type 0 CIDFonts, this is usually the value of the **CIDFontName** entry in the CIDFont program. For Type 2 CIDFonts, it is derived the same way as for a simple TrueType font; see Section 5.5.2, "TrueType Fonts." In either case, the name can have a subset prefix if appropriate; see Section 5.5.3, "Font Subsets."
CIDSystemInfo	dictionary	*(Required)* A dictionary containing entries that define the character collection of the CIDFont. See Table 5.13 on page 397.
FontDescriptor	dictionary	*(Required; must be an indirect reference)* A font descriptor describing the CIDFont's default metrics other than its glyph widths (see Section 5.7, "Font Descriptors").
DW	integer	*(Optional)* The default width for glyphs in the CIDFont (see "Glyph Metrics in CIDFonts" on page 401). Default value: 1000.
W	array	*(Optional)* A description of the widths for the glyphs in the CIDFont. The array's elements have a variable format that can specify individual widths for consecutive CIDs or one width for a range of CIDs (see "Glyph Metrics in CIDFonts" on page 401). Default value: none (the **DW** value is used for all glyphs).
DW2	array	*(Optional; applies only to CIDFonts used for vertical writing)* An array of two numbers specifying the default metrics for vertical writing (see "Glyph Metrics in CIDFonts" on page 401). Default value: [880 −1000].
W2	array	*(Optional; applies only to CIDFonts used for vertical writing)* A description of the metrics for vertical writing for the glyphs in the CIDFont (see "Glyph Metrics in CIDFonts" on page 401). Default value: none (the **DW2** value is used for all glyphs).
CIDToGIDMap	stream or name	*(Optional; Type 2 CIDFonts only)* A specification of the mapping from CIDs to glyph indices. If the value is a stream, the bytes in the stream contain the mapping from CIDs to glyph indices: the glyph index for a particular CID value c is a 2-byte value stored in bytes $2 \times c$ and $2 \times c + 1$, where the first byte is the high-order byte. If the value of **CIDToGIDMap** is a name, it must be **Identity**, indicating that the mapping between CIDs and glyph indices is the identity mapping. Default value: **Identity**. This entry may appear only in a Type 2 CIDFont whose associated TrueType font program is embedded in the PDF file (see the next section).

Glyph Selection in CIDFonts

Type 0 and Type 2 CIDFonts handle the mapping from CIDs to glyph descriptions in somewhat different ways.

For Type 0, the CIDFont program itself contains glyph descriptions that are identified by CIDs. The CIDFont program identifies the character collection by a **CIDSystemInfo** dictionary, which should simply be copied into the PDF CIDFont dictionary. CIDs are interpreted uniformly in all CIDFont programs supporting a given character collection, whether the program is embedded in the PDF file or obtained from an external source.

For Type 2, the CIDFont program is actually a TrueType font program, which has no native notion of CIDs. In a TrueType font program, glyph descriptions are identified by *glyph index* values. Glyph indices are internal to the font and are not defined consistently from one font to another. Instead, a TrueType font program contains a "cmap" table that provides mappings directly from character codes to glyph indices for one or more predefined encodings.

TrueType font programs are integrated with the CID-keyed font architecture in one of two ways, depending on whether the font program is embedded in the PDF file.

- If the TrueType font program is embedded, the Type 2 CIDFont dictionary must contain a **CIDToGIDMap** entry that maps CIDs to the glyph indices for the appropriate glyph descriptions in that font program.
- If the TrueType font program is not embedded but is referenced by name, the Type 2 CIDFont dictionary must *not* contain a **CIDToGIDMap** entry, since it is not meaningful to refer to glyph indices in an external font program. In this case, CIDs do not participate in glyph selection, and only predefined CMaps may be used with this CIDFont (see Section 5.6.4, "CMaps"). The viewer application selects glyphs by translating characters from the encoding specified by the predefined CMap to one of the encodings given in the TrueType font's "cmap" table. The means by which this is accomplished are implementation-dependent.

Even though the CIDs are sometimes not used to select glyphs in a Type 2 CIDFont, they are always used to determine the glyph metrics, as described in the next section.

Every CIDFont must contain a glyph description for CID 0, which is analogous to the .notdef character name in simple fonts (see "Handling Undefined Characters" on page 416).

Glyph Metrics in CIDFonts

As discussed in Section 5.1.3, "Glyph Positioning and Metrics," the *width* of a glyph refers to the horizontal displacement between the origin of the glyph and the origin of the next glyph when writing in horizontal mode. In this mode, the vertical displacement between origins is always 0. Widths for a CIDFont are defined using the **DW** and **W** entries in the CIDFont dictionary. These widths must be consistent with the actual widths given in the CIDFont program itself. (See implementation note 53 in Appendix H.)

The **DW** entry defines the default width, which is used for all glyphs whose widths are not specified individually. This entry is particularly useful for Chinese, Japanese, and Korean fonts, in which many of the glyphs have the same width.

The **W** array allows the definition of widths for individual CIDs. The elements of the array are organized in groups of two or three, where each group is in one of the following two formats:

$c\ [w_1\ w_2\ \ldots\ w_n]$
$c_{first}\ c_{last}\ w$

In the first format, c is an integer specifying a starting CID value; it is followed by an array of n numbers that specify the widths for n consecutive CIDs, starting with c. The second format defines the same width, w, for all CIDs in the range c_{first} to c_{last}.

The following is an example of a **W** entry:

```
/W [ 120 [400 325 500]
     7080 8032 1000
   ]
```

In this example, the glyphs having CIDs 120, 121, and 122 are 400, 325, and 500 units wide, respectively. CIDs in the range 7080 through 8032 all have a width of 1000 units.

Glyphs from a CIDFont can be shown in vertical writing mode. (This is selected by the **WMode** entry in the associated CMap dictionary; see Section 5.6.4, "CMaps.") To be used in this way, the CIDFont must define the vertical displacement for each glyph and the position vector that relates the horizontal and vertical writing origins.

The default position vector and vertical displacement vector are specified by the **DW2** entry in the CIDFont dictionary. **DW2** is an array of two values: the vertical component of the position vector *v* and the vertical component of the displacement vector *w1* (see Figure 5.5 on page 358). The horizontal component of the position vector is always half the glyph width, and that of the displacement vector is always 0. For example, if the **DW2** entry is

```
/DW2 [880 −1000]
```

then a glyph's position vector and vertical displacement vector are

$$\begin{aligned} v &= (w0 \div 2, 880) \\ w1 &= (0, -1000) \end{aligned}$$

where *w0* is the width (horizontal displacement) for the same glyph. Note that a negative value for the vertical component will place the origin of the next glyph *below* the current glyph, because vertical coordinates in a standard coordinate system increase from bottom to top.

The **W2** array allows the definition of vertical metrics for individual CIDs. The elements of the array are organized in groups of two or five, where each group is in one of the following two formats:

$c\ [w1_{1y}\ v_{1x}\ v_{1y}\ \ w1_{2y}\ v_{2x}\ v_{2y}\ \ \ldots]$
$c_{first}\ c_{last}\ \ w1_{1y}\ v_{1x}\ v_{1y}$

In the first format, *c* is a starting CID and is followed by an array containing numbers interpreted in groups of three. Each group consists of the vertical component of the vertical displacement vector *w*1 (whose horizontal component is always 0) followed by the horizontal and vertical components for the position vector *v*. Successive groups define the vertical metrics for consecutive CIDs starting with *c*. The second format defines a range of CIDs from c_{first} to c_{last}, followed by three numbers that define the vertical metrics for all CIDs in this range. For example:

```
/W2 [ 120 [−1000 250 772]
        7080 8032 −1000 500 900
    ]
```

This **W2** entry defines the vertical displacement vector for the glyph with CID 120 as (0, −1000) and the position vector as (250, 772). It also defines the displacement vector for CIDs in the range 7080 through 8032 as (0, −1000) and the position vector as (500, 900).

5.6.4 CMaps

A CMap specifies the mapping from character codes to character selectors. In PDF, the character selectors are always CIDs in a CIDFont (as mentioned earlier, PostScript CMaps may use names or codes as well). A CMap serves a function analogous to the **Encoding** dictionary for a simple font. The CMap does not refer directly to a specific CIDFont; instead, it is combined with it as part of a CID-keyed font, represented in PDF as a Type 0 font dictionary (see Section 5.6.5, "Type 0 Font Dictionaries"). Within the CMap, the character mappings refer to the associated CIDFont by *font number*, which in PDF is always 0.

Note: *PDF also uses a special type of CMap to map character codes to Unicode values (see Section 5.9.2, "ToUnicode CMaps").*

A CMap also specifies the writing mode—horizontal or vertical—for any CIDFont with which the CMap is combined. This determines which metrics are to be used when glyphs are painted from that font. (Writing mode is specified as part of the CMap because, in some cases, different shapes are used when writing horizontally and vertically. In that case, the horizontal and vertical variants of a CMap specify different CIDs for a given character code.)

A CMap may be specified in two ways:

- As a name object identifying a predefined CMap, whose definition is known to the viewer application.
- As a stream object whose contents are a CMap file. (See implementation note 58 in Appendix H.)

Predefined CMaps

Table 5.15 lists the names of the predefined CMaps. These CMaps map character codes to CIDs in a single descendant CIDFont. CMaps whose names end in H specify horizontal writing mode; those ending in V specify vertical writing mode.

***Note:** Several of the CMaps define mappings from Unicode encodings to character collections. Unicode values appearing in a text string are represented in "big-endian" order (high-order byte first). CMap names containing "UCS2" use UCS-2 encoding; names containing "UTF16" use UTF-16BE ("big-endian") encoding.*

TABLE 5.15 Predefined CJK CMap names

NAME	DESCRIPTION
Chinese (Simplified)	
GB–EUC–H	Microsoft Code Page 936 (lfCharSet 0x86), GB 2312-80 character set, EUC-CN encoding
GB–EUC–V	Vertical version of GB–EUC–H
GBpc–EUC–H	Mac OS, GB 2312-80 character set, EUC-CN encoding, Script Manager code 19
GBpc–EUC–V	Vertical version of GBpc–EUC–H
GBK–EUC–H	Microsoft Code Page 936 (lfCharSet 0x86), GBK character set, GBK encoding
GBK–EUC–V	Vertical version of GBK–EUC–H
GBKp–EUC–H	Same as GBK–EUC–H, but replaces half-width Latin characters with proportional forms and maps character code 0x24 to a dollar sign ($) instead of a yuan symbol (¥)
GBKp–EUC–V	Vertical version of GBKp–EUC–H
GBK2K–H	GB 18030-2000 character set, mixed 1-, 2-, and 4-byte encoding
GBK2K–V	Vertical version of GBK2K–H
UniGB–UCS2–H	Unicode (UCS-2) encoding for the Adobe-GB1 character collection
UniGB–UCS2–V	Vertical version of UniGB–UCS2–H
UniGB–UTF16–H	Unicode (UTF-16BE) encoding for the Adobe-GB1 character collection. Contains mappings for all characters in the GB18030-2000 character set.
UniGB–UTF16–V	Vertical version of UniGB–UTF16–H.

NAME	DESCRIPTION
Chinese (Traditional)	
B5pc–H	Mac OS, Big Five character set, Big Five encoding, Script Manager code 2
B5pc–V	Vertical version of B5pc–H
HKscs–B5–H	Hong Kong SCS, an extension to the Big Five character set and encoding
HKscs–B5–V	Vertical version of HKscs–B5–H
ETen–B5–H	Microsoft Code Page 950 (lfCharSet 0x88), Big Five character set with ETen extensions
ETen–B5–V	Vertical version of ETen–B5–H
ETenms–B5–H	Same as ETen–B5–H, but replaces half-width Latin characters with proportional forms
ETenms–B5–V	Vertical version of ETenms–B5–H
CNS–EUC–H	CNS 11643-1992 character set, EUC-TW encoding
CNS–EUC–V	Vertical version of CNS–EUC–H
UniCNS–UCS2–H	Unicode (UCS-2) encoding for the Adobe-CNS1 character collection
UniCNS–UCS2–V	Vertical version of UniCNS–UCS2–H
UniCNS–UTF16–H	Unicode (UTF-16BE) encoding for the Adobe-CNS1 character collection. Contains mappings for all the characters in the HKSCS-2001 character set. Contains both 2- and 4-byte character codes.
UniCNS–UTF16–V	Vertical version of UniCNS–UTF16–H.
Japanese	
83pv–RKSJ–H	Mac OS, JIS X 0208 character set with KanjiTalk6 extensions, Shift-JIS encoding, Script Manager code 1
90ms–RKSJ–H	Microsoft Code Page 932 (lfCharSet 0x80), JIS X 0208 character set with NEC and IBM® extensions
90ms–RKSJ–V	Vertical version of 90ms–RKSJ–H
90msp–RKSJ–H	Same as 90ms–RKSJ–H, but replaces half-width Latin characters with proportional forms
90msp–RKSJ–V	Vertical version of 90msp–RKSJ–H

NAME	DESCRIPTION
90pv–RKSJ–H	Mac OS, JIS X 0208 character set with KanjiTalk7 extensions, Shift-JIS encoding, Script Manager code 1
Add–RKSJ–H	JIS X 0208 character set with Fujitsu FMR extensions, Shift-JIS encoding
Add–RKSJ–V	Vertical version of Add–RKSJ–H
EUC–H	JIS X 0208 character set, EUC-JP encoding
EUC–V	Vertical version of EUC–H
Ext–RKSJ–H	JIS C 6226 (JIS78) character set with NEC extensions, Shift-JIS encoding
Ext–RKSJ–V	Vertical version of Ext–RKSJ–H
H	JIS X 0208 character set, ISO-2022-JP encoding
V	Vertical version of H
UniJIS–UCS2–H	Unicode (UCS-2) encoding for the Adobe-Japan1 character collection
UniJIS–UCS2–V	Vertical version of UniJIS–UCS2–H
UniJIS–UCS2–HW–H	Same as UniJIS–UCS2–H, but replaces proportional Latin characters with half-width forms
UniJIS–UCS2–HW–V	Vertical version of UniJIS–UCS2–HW–H
UniJIS–UTF16-H	Unicode (UTF-16BE) encoding for the Adobe-Japan1 character collection. Contains mappings for all characters in the JIS X 0213:1000 character set.
UniJIS–UTF16-V	Vertical version of UniJIS–UTF16-H.
Korean	
KSC–EUC–H	KS X 1001:1992 character set, EUC-KR encoding
KSC–EUC–V	Vertical version of KSC–EUC–H
KSCms–UHC–H	Microsoft Code Page 949 (lfCharSet 0x81), KS X 1001:1992 character set plus 8822 additional hangul, Unified Hangul Code (UHC) encoding
KSCms–UHC–V	Vertical version of KSCms–UHC–H
KSCms–UHC–HW–H	Same as KSCms–UHC–H, but replaces proportional Latin characters with half-width forms

NAME	DESCRIPTION
KSCms–UHC–HW–V	Vertical version of KSCms–UHC–HW–H
KSCpc–EUC–H	Mac OS, KS X 1001:1992 character set with Mac OS KH extensions, Script Manager Code 3
UniKS–UCS2–H	Unicode (UCS-2) encoding for the Adobe-Korea1 character collection
UniKS–UCS2–V	Vertical version of UniKS–UCS2–H
UniKS–UTF16–H	Unicode (UTF-16BE) encoding for the Adobe-Korea1 character collection.
UniKS–UTF16–V	Vertical version of UniKS–UTF16–H.
Generic	
Identity–H	The horizontal identity mapping for 2-byte CIDs; may be used with CIDFonts using any **Registry**, **Ordering**, and **Supplement** values. It maps 2-byte character codes ranging from 0 to 65,535 to the same 2-byte CID value, interpreted high-order byte first (see below).
Identity–V	Vertical version of Identity–H. The mapping is the same as for Identity–H.

The Identity–H and Identity–V CMaps can be used to refer to glyphs directly by their CIDs when showing a text string. When the current font is a Type 0 font whose **Encoding** entry is Identity–H or Identity–V, the string to be shown is interpreted as pairs of bytes representing CIDs, high-order byte first. This works with any CIDFont, independently of its character collection. Additionally, when used in conjunction with a Type 2 CIDFont whose **CIDToGIDMap** entry is **Identity**, the 2-byte CIDs values in fact represent glyph indices for the glyph descriptions in the TrueType font program. This works only if the TrueType font program is embedded in the PDF file.

Table 5.16 lists the character collections referenced by the predefined CMaps for the different versions of PDF. A dash (—) indicates that the CMap is not predefined in that PDF version.

TABLE 5.16 Character collections for predefined CMaps, by PDF version

CMAP	PDF 1.2	PDF 1.3	PDF 1.4	PDF 1.5
Chinese (Simplified)				
GB–EUC–H/V	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0

CMAP	PDF 1.2	PDF 1.3	PDF 1.4	PDF 1.5
GBpc–EUC–H	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0
GBpc–EUC–V	—	Adobe-GB1-0	Adobe-GB1-0	Adobe-GB1-0
GBK–EUC–H/V	—	Adobe-GB1-2	Adobe-GB1-2	Adobe-GB1-2
GBKp–EUC–H/V	—	—	Adobe-GB1-2	Adobe-GB1-2
GBK2K–H/V	—	—	Adobe-GB1-4	Adobe-GB1-4
UniGB–UCS2–H/V	—	Adobe-GB1-2	Adobe-GB1-4	Adobe-GB1-4
UniGB–UTF16-H/V	—	—	—	Adobe-GB1-4
Chinese (Traditional)				
B5pc–H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
HKscs–B5–H/V	—	—	Adobe-CNS1-3	Adobe-CNS1-3
ETen–B5–H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
ETenms–B5–H/V	—	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
CNS–EUC–H/V	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0	Adobe-CNS1-0
UniCNS–UCS2–H/V	—	Adobe-CNS1-0	Adobe-CNS1-3	Adobe-CNS1-3
UniCNS–UTF16-H/V	—	—	—	Adobe-CNS1-4
Japanese				
83pv–RKSJ–H	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
90ms–RKSJ–H/V	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
90msp–RKSJ–H/V	—	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
90pv–RKSJ–H	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
Add–RKSJ–H/V	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
EUC–H/V	—	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1
Ext–RKSJ–H/V	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2	Adobe-Japan1-2
H/V	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1	Adobe-Japan1-1

CMAP	PDF 1.2	PDF 1.3	PDF 1.4	PDF 1.5
UniJIS–UCS2–H/V	—	Adobe-Japan1-2	Adobe-Japan1-4	Adobe-Japan1-4
UniJIS–UCS2–HW–H/V	—	Adobe-Japan1-2	Adobe-Japan1-4	Adobe-Japan1-4
UniJIS–UTF16–H/V	—	—	—	Adobe-Japan1-5
Korean				
KSC–EUC–H/V	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0
KSCms–UHC–H/V	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
KSCms–UHC–HW–H/V	—	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
KSCpc–EUC–H	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0	Adobe-Korea1-0
UniKS–UCS2–H/V	—	Adobe-Korea1-1	Adobe-Korea1-1	Adobe-Korea1-1
UniKS–UTF16–H/V	—	—	—	Adobe-Korea1-2
Generic				
Identity–H/V	Adobe-Identity-0	Adobe-Identity-0	Adobe-Identity-0	Adobe-Identity-0

As noted in Section 5.6.2, "CIDSystemInfo Dictionaries," a character collection is identified by registry, ordering, and supplement number, and supplements are cumulative; that is, a higher-numbered supplement includes the CIDs contained in lower-numbered supplements, as well as some additional CIDs. Consequently, text encoded according to the predefined CMaps for a given PDF version will be valid when interpreted by a viewer application supporting the same or a later PDF version. When interpreted by a viewer supporting an earlier PDF version, such text will cause an error if a CMap is encountered that is not predefined for that PDF version. If character codes are encountered that were added in a higher-numbered supplement than the one corresponding to the supported PDF version, no characters will be displayed for those codes; see "Handling Undefined Characters" on page 416. See also implementation note 59 in Appendix H.

Note: *If an application producing a PDF file encounters text to be included that uses CIDs from a higher-numbered supplement than the one corresponding to the PDF version being generated, the application should embed the CMap for the higher-numbered supplement rather than refer to the predefined CMap (see the next section).*

The CMap programs that define the predefined CMaps are available through the ASN Web site and are also provided in conjunction with the book *CJKV Information Processing*, by Ken Lunde. Details on the character collections, including sample glyphs for all the CIDs, can be found in a number of Adobe Technical Notes; for more information about these Notes and the aforementioned book, see the Bibliography.

Embedded CMap Files

For character encodings that are not predefined, the PDF file must contain a stream that defines the CMap. In addition to the standard entries for streams (listed in Table 3.4 on page 38), the CMap stream dictionary contains the entries listed in Table 5.17. The data in the stream defines the mapping from character codes to a font number and a character selector. The data must follow the syntax defined in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*.

TABLE 5.17 Additional entries in a CMap dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **CMap** for a CMap dictionary. (Note that although this object is the value of an entry named **Encoding** in a Type 0 font, its type is **CMap**.)
CMapName	name	*(Required)* The PostScript name of the CMap. This should be the same as the value of **CMapName** in the CMap file itself.
CIDSystemInfo	dictionary	*(Required)* A dictionary (see Section 5.6.2, "CIDSystemInfo Dictionaries") containing entries that define the character collection for the CIDFont or CIDFonts associated with the CMap. The value of this entry should be the same as the value of **CIDSystemInfo** in the CMap file itself. (However, it does not need to match the values of **CIDSystemInfo** for the Identity-H or Identity-V CMaps.
WMode	integer	*(Optional)* A code that determines the writing mode for any CIDFont with which this CMap is combined. The possible values are 0 for horizontal and 1 for vertical. Default value: 0. The value of this entry should be the same as the value of **WMode** in the CMap file itself.

KEY	TYPE	VALUE
UseCMap	name or stream	*(Optional)* The name of a predefined CMap, or a stream containing a CMap, that is to be used as the base for this CMap. This allows the CMap to be defined differentially, specifying only the character mappings that differ from the base CMap.

CMap Example and Operator Summary

CMap files are fully documented in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*. The following example of a CMap stream object illustrates and partially explains the contents of a CMap file. There are several reasons for including this material here:

- It documents some restrictions on the contents of a CMap file that can be embedded in a PDF file.
- It provides background to aid in understanding subsequent material, particularly "CMap Mapping" on page 415.
- It is the basis for a PDF feature, the **ToUnicode** CMap, which is a minor extension of the CMap file format. This extension is described in Section 5.9, "Extraction of Text Content."

Example 5.11 is a sample CMap for a Japanese Shift-JIS encoding. Character codes in this encoding can be either 1 or 2 bytes in length. This CMap could be used with a CIDFont that uses the same CID ordering as specified in the **CIDSystemInfo** entry. Note that several of the entries in the stream dictionary are also replicated in the stream data itself.

Example 5.11

```
22 0 obj
   << /Type /CMap
      /CMapName /90ms−RKSJ−H
      /CIDSystemInfo << /Registry (Adobe)
                        /Ordering (Japan1)
                        /Supplement 2
                     >>
      /WMode 0
      /Length 23 0 R
   >>
```

```
stream
%!PS–Adobe–3.0 Resource–CMap
%%DocumentNeededResources: ProcSet (CIDInit)
%%IncludeResource: ProcSet (CIDInit)
%%BeginResource: CMap (90ms–RKSJ–H)
%%Title: (90ms–RKSJ–H Adobe Japan1 2)
%%Version: 10.001
%%Copyright: Copyright 1990–2001 Adobe Systems Inc.
%%Copyright: All Rights Reserved.
%%EndComments

/CIDInit /ProcSet findresource begin
12 dict begin
begincmap
/CIDSystemInfo
3 dict dup begin
/Registry (Adobe) def
/Ordering (Japan1) def
/Supplement 2 def
end def

/CMapName /90ms–RKSJ–H def
/CMapVersion 10.001 def
/CMapType 1 def
/UIDOffset 950 def
/XUID [1 10 25343] def
/WMode 0 def

4 begincodespacerange
<00>     <80>
<8140>   <9FFC>
<A0>     <DF>
<E040>   <FCFC>
endcodespacerange

1 beginnotdefrange
<00>     <1F>     231
endnotdefrange
```

```
100 begincidrange
<20>    <7D>    231
<7E>    <7E>    631
<8140>  <817E>  633
<8180>  <81AC>  696
<81B8>  <81BF>  741
<81C8>  <81CE>  749
… Additional ranges …
<FB40>  <FB7E>  8518
<FB80>  <FBFC>  8581
<FC40>  <FC4B>  8706
endcidrange
endcmap
CMapName currentdict /CMap defineresource pop
end
end
%%EndResource
%%EOF
endstream
endobj
```

As can be seen from this example, a CMap file conforms to PostScript language syntax; however, a full PostScript interpreter is not needed to interpret it. Aside from some required boilerplate, the CMap file consists of one or more occurrences of several special CMap construction operators, invoked in a specific order. The following is a summary of these operators.

- **begincmap** and **endcmap** enclose the CMap definition.
- **usecmap** incorporates the code mappings from another CMap file. In PDF, the other CMap must also be identified in the **UseCMap** entry in the CMap dictionary (see Table 5.17 on page 410).
- **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 415.
- **usefont** specifies a font number that is an implicit operand of all the character code mapping operations that follow. In PDF, the font number must be 0; therefore, **usefont** typically does not actually appear.
- **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. In PDF, these operators may not appear in a CMap that is used as the **Encoding** entry of a Type 0 font; however, they may appear in the definition of a **ToUnicode** CMap (see Section 5.9, "Extraction of Text Content").

- **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 416, a notdef mapping is used if the normal mapping produces a CID for which no glyph is present in the associated CIDFont.

The **beginrearrangedfont**, **endrearrangedfont**, **beginusematrix**, and **endusematrix** operators, described in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*, cannot be used in CMap files embedded in a PDF file.

5.6.5 Type 0 Font Dictionaries

A Type 0 font dictionary contains the entries listed in Table 5.18.

Example 5.12 shows a Type 0 font that refers to a single CIDFont. The CMap used is one of the predefined CMaps listed in Table 5.15 on page 404, and is referenced by name.

TABLE 5.18 Entries in a Type 0 font dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Font** for a font dictionary.
Subtype	name	*(Required)* The type of font; must be **Type0** for a Type 0 font.

KEY	TYPE	VALUE
BaseFont	name	*(Required)* The PostScript name of the font. In principle, this is an arbitrary name, since there is no font program associated directly with a Type 0 font dictionary. The conventions described here ensure maximum compatibility with existing Acrobat products. If the descendant is a Type 0 CIDFont, this name should be the concatenation of the CIDFont's **BaseFont** name, a hyphen, and the CMap name given in the **Encoding** entry (or the **CMapName** entry in the CMap program itself). If the descendant is a Type 2 CIDFont, this name should be the same as the CIDFont's **BaseFont** name.
Encoding	name or stream	*(Required)* The name of a predefined CMap, or a stream containing a CMap program, that maps character codes to font numbers and CIDs. If the descendant is a Type 2 CIDFont whose associated TrueType font program is not embedded in the PDF file, the **Encoding** entry must be a predefined CMap name (see "Glyph Selection in CIDFonts" on page 400).
DescendantFonts	array	*(Required)* A one-element array specifying the CIDFont that is the descendant of this Type 0 font.
ToUnicode	stream	*(Optional)* A stream containing a CMap file that maps character codes to Unicode values (see Section 5.9, "Extraction of Text Content").

Example 5.12

```
14 0 obj
   << /Type /Font
      /Subtype /Type0
      /BaseFont /HeiseiMin–W5–90ms–RKSJ–H
      /Encoding /90ms–RKSJ–H
      /DescendantFonts [15 0 R]
   >>
endobj
```

CMap Mapping

The **Encoding** entry of a Type 0 font dictionary specifies a CMap that determines how text-showing operators (such as **Tj**) interpret the bytes in the string to be shown, when the current font is the Type 0 font. The following paragraphs describe how the characters in the string are decoded and mapped into character selectors (which in PDF must always be CIDs).

The codespace ranges in the CMap (delimited by **begincodespacerange** and **end-codespacerange**) determine how many bytes are extracted from the string for each successive character code. 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 C).

A sequence of one or more bytes is extracted from the string and matched against the codespace ranges in the CMap. That is, the first byte is matched against 1-byte codespace ranges; if no match is found, a second byte is extracted, and the 2-byte code is matched against 2-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 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. The font number is used as an index into the Type 0 font's **Descendant-Fonts** array, selecting a CIDFont. In PDF, the font number is always 0 and the character selector is always a CID; this is the only case described here. The CID is then used to select a glyph in the CIDFont. If the CIDFont contains no glyph for that CID, the notdef mappings are consulted, as described in the next section.

Handling Undefined Characters

A CMap mapping operation can fail to select a glyph for a variety of reasons. This section describes those reasons and what happens when they occur.

If a code maps to a CID for which 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 simple fonts) are delimited by the operators **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. If there is no glyph for that CID, the glyph for CID 0 (which is required to be present) is substituted.

If the CMap does not contain either a character mapping or a notdef mapping for the code, descendant 0 is selected and the glyph for CID 0 is substituted from the associated CIDFont.

If the code is invalid—that is, the bytes extracted from the string to be shown 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 string, and a modified mapping algorithm chooses the best partially matching codespace range, as follows:

1. If the first byte extracted from the string to be shown 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), 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 string for the current mapping operation.

5.7 Font Descriptors

A *font descriptor* specifies metrics and other attributes of a simple font or a CIDFont as a whole, as distinct from the metrics of individual glyphs. These font metrics provide information that enables a viewer application to synthesize a substitute font or select a similar font when the font program is unavailable. The font descriptor may also be used to embed the font program in the PDF file.

Font descriptors are not used with Type 0 fonts. Beginning with PDF 1.5, font descriptors may be used with Type 3 fonts in Tagged PDF documents (see Section 10.7, “Tagged PDF”).

A font descriptor is a dictionary whose entries specify various font attributes. The entries common to all font descriptors—for both simple fonts and CIDFonts—are listed in Table 5.19; additional entries in the font descriptor for a CIDFont are described in Section 5.7.2, "Font Descriptors for CIDFonts." All integer values are units in glyph space. The conversion from glyph space to text space is described in Section 5.1.3, "Glyph Positioning and Metrics."

TABLE 5.19 Entries common to all font descriptors

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **FontDescriptor** for a font descriptor.
FontName	name	*(Required)* The PostScript name of the font. This should be the same as the value of **BaseFont** in the font or CIDFont dictionary that refers to this font descriptor.
FontFamily	string	*(Optional; PDF 1.5; strongly recommended for Type 3 fonts in Tagged PDF documents)* A string specifying the preferred font family name. For example, for the font Times Bold Italic, the **FontFamily** is Times.
FontStretch	name	*(Optional; PDF 1.5; strongly recommended for Type 3 fonts in Tagged PDF documents)* The font stretch value; it must be one of the following (ordered from narrowest to widest): UltraCondensed, ExtraCondensed, Condensed, SemiCondensed, Normal, SemiExpanded, Expanded, ExtraExpanded or UltraExpanded. ***Note:** The specific interpretation of these values varies from font to font. For example, **Condensed** in one font may appear most similar to **Normal** in another.*
FontWeight	number	*(Optional; PDF 1.5; strongly recommended for Type 3 fonts in Tagged PDF documents)* The weight (thickness) component of the fully-qualified font name or font specifier. The possible values are 100, 200, 300, 400, 500, 600, 700, 800 or 900, where each number indicates a weight that is at least as dark as its predecessor. A value of 400 indicates a normal weight; 700 indicates bold. ***Note:** The specific interpretation of these values varies from font to font. For example, 300 in one font may appear most similar to 500 in another.*
Flags	integer	*(Required)* A collection of flags defining various characteristics of the font (see Section 5.7.1, "Font Descriptor Flags").
FontBBox	rectangle	*(Required, except for Type 3 fonts)* A rectangle (see Section 3.8.4, "Rectangles"), expressed in the glyph coordinate system, specifying the *font bounding box*. This 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 filled.

KEY	TYPE	VALUE
ItalicAngle	number	*(Required)* The angle, expressed in degrees counterclockwise from the vertical, of the dominant vertical strokes of the font. (For example, the 9-o'clock position is 90 degrees, and the 3-o'clock position is −90 degrees.) The value is negative for fonts that slope to the right, as almost all italic fonts do.
Ascent	number	*(Required, except for Type 3 fonts)* The maximum height above the baseline reached by glyphs in this font, excluding the height of glyphs for accented characters.
Descent	number	*(Required, except for Type 3 fonts)* The maximum depth below the baseline reached by glyphs in this font. The value is a negative number.
Leading	number	*(Optional)* The desired spacing between baselines of consecutive lines of text. Default value: 0.
CapHeight	number	*(Required for fonts that have Latin characters, except for Type 3 fonts)* The vertical coordinate of the top of flat capital letters, measured from the baseline.
XHeight	number	*(Optional)* The font's *x height*: the vertical coordinate of the top of flat nonascending lowercase letters (like the letter *x*), measured from the baseline, in fonts that have Latin characters. Default value: 0.
StemV	number	*(Required, except for Type 3 fonts)* The thickness, measured horizontally, of the dominant vertical stems of glyphs in the font.
StemH	number	*(Optional)* The thickness, measured vertically, of the dominant horizontal stems of glyphs in the font. Default value: 0.
AvgWidth	number	*(Optional)* The average width of glyphs in the font. Default value: 0.
MaxWidth	number	*(Optional)* The maximum width of glyphs in the font. Default value: 0.
MissingWidth	number	*(Optional)* The width to use for character codes whose widths are not specified in a font dictionary's **Widths** array. This has a predictable effect only if all such codes map to glyphs whose actual widths are the same as the **MissingWidth** value. Default value: 0.
FontFile	stream	*(Optional)* A stream containing a Type 1 font program (see Section 5.8, "Embedded Font Programs").
FontFile2	stream	*(Optional; PDF 1.1)* A stream containing a TrueType font program (see Section 5.8, "Embedded Font Programs").

KEY	TYPE	VALUE
FontFile3	stream	*(Optional; PDF 1.2)* A stream containing a font program other than Type 1 or TrueType. The format of the font program is specified by the **Subtype** entry in the stream dictionary (see Section 5.8, "Embedded Font Programs," and implementation note 60 in Appendix H). At most, only one of the **FontFile**, **FontFile2**, and **FontFile3** entries may be present.
CharSet	string	*(Optional; meaningful only in Type 1 fonts; PDF 1.1)* A string listing the character names defined in a font subset. The names in this string must be in PDF syntax—that is, each name preceded by a slash (/). The names can appear in any order. The name .notdef should be omitted; it is assumed to exist in the font subset. If this entry is absent, the only indication of a font subset is the subset tag in the **FontName** entry (see Section 5.5.3, "Font Subsets").

5.7.1 Font Descriptor Flags

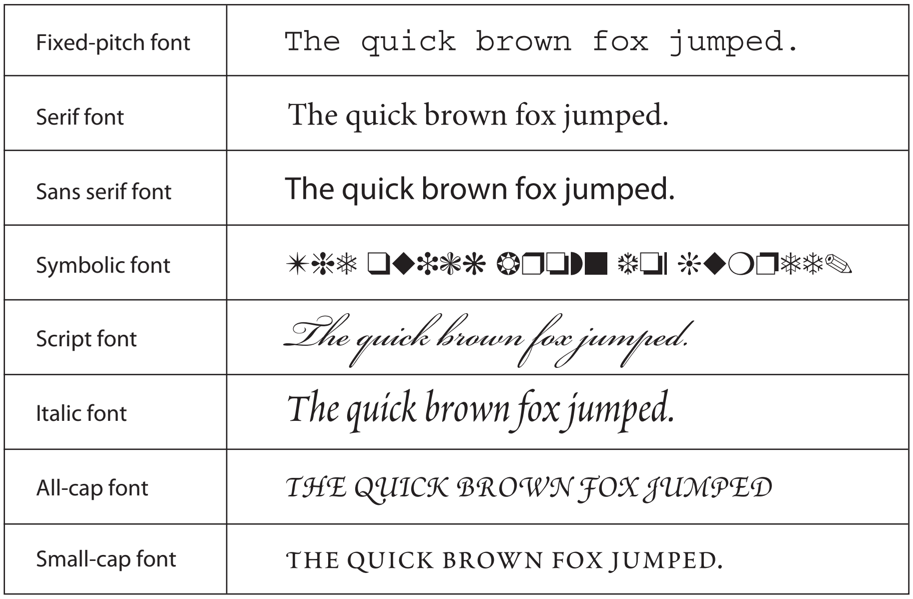
The value of the **Flags** entry in a font descriptor is an unsigned 32-bit integer containing flags specifying various characteristics of the font. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table 5.20 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0. Figure 5.13 shows examples of fonts with these characteristics.

TABLE 5.20 Font flags

BIT POSITION	NAME	MEANING
1	FixedPitch	All glyphs have the same width (as opposed to proportional or variable-pitch fonts, which have different widths).
2	Serif	Glyphs have serifs, which are short strokes drawn at an angle on the top and bottom of glyph stems (as opposed to *sans serif* fonts, which do not).
3	Symbolic	Font contains glyphs outside the Adobe standard Latin character set. This flag and the Nonsymbolic flag cannot both be set or both be clear (see below).
4	Script	Glyphs resemble cursive handwriting.
6	Nonsymbolic	Font uses the Adobe standard Latin character set or a subset of it (see below).
7	Italic	Glyphs have dominant vertical strokes that are slanted.

BIT POSITION	NAME	MEANING
17	AllCap	Font contains no lowercase letters; typically used for display purposes such as titles or headlines.
18	SmallCap	Font contains both uppercase and lowercase letters. The uppercase letters are similar to ones in the regular version of the same typeface family. The glyphs for the lowercase letters have the same shapes as the corresponding uppercase letters, but they are sized and their proportions adjusted so that they have the same size and stroke weight as lowercase glyphs in the same typeface family.
19	ForceBold	See below.

The Nonsymbolic flag (bit 6 in the **Flags** entry) indicates that the font's character set is the Adobe standard Latin character set (or a subset of it) and that it uses the standard names for those glyphs. This character set is shown in Section D.1, "Latin Character Set and Encodings." If the font contains any glyphs outside this set, the Symbolic flag should be set and the Nonsymbolic flag clear; in other words, any font whose character set is not a subset of the Adobe standard character set is considered to be symbolic. This influences the font's implicit base encoding and may affect a viewer application's font substitution strategies.



Fixed-pitch font	The quick brown fox jumped.
Serif font	The quick brown fox jumped.
Sans serif font	The quick brown fox jumped.
Symbolic font	✳❉❈ ❑◆❇❊❋ ✪❒❑◗■ ❄❑ ✲◆❍❒❊❄✎
Script font	The quick brown fox jumped.
Italic font	*The quick brown fox jumped.*
All-cap font	THE QUICK BROWN FOX JUMPED
Small-cap font	THE QUICK BROWN FOX JUMPED.

FIGURE 5.13 *Characteristics represented in the* ***Flags*** *entry of a font descriptor*

***Note:** This classification of nonsymbolic and symbolic fonts is peculiar to PDF. A font may contain additional characters that are used in Latin writing systems but are outside the Adobe standard Latin character set; PDF considers such a font to be symbolic. The use of two flags to represent a single binary choice is a historical accident.*

The ForceBold flag (bit 19) determines whether bold glyphs are painted with extra pixels even at very small text sizes. Typically, when glyphs are painted at small sizes on very low-resolution devices such as display screens, features of bold glyphs may appear only 1 pixel wide. Because this is the minimum feature width on a pixel-based device, ordinary (nonbold) glyphs also appear with 1-pixel-wide features, and so cannot be distinguished from bold glyphs. If the ForceBold flag is set, features of bold glyphs may be thickened at small text sizes.

Example 5.13 illustrates a font descriptor whose **Flags** entry has the Serif, Nonsymbolic, and ForceBold flags (bits 2, 6, and 19) set.

Example 5.13

```
7 0 obj
   << /Type /FontDescriptor
      /FontName /AGaramond–Semibold
      /Flags 262178                                % Bits 2, 6, and 19
      /FontBBox [–177 –269 1123 866]
      /MissingWidth 255
      /StemV 105
      /StemH 45
      /CapHeight 660
      /XHeight 394
      /Ascent 720
      /Descent –270
      /Leading 83
      /MaxWidth 1212
      /AvgWidth 478
      /ItalicAngle 0
   >>
endobj
```

5.7.2 Font Descriptors for CIDFonts

In addition to the entries in Table 5.19 on page 418, the **FontDescriptor** dictionaries of CIDFonts may contain the entries listed in Table 5.21.

TABLE 5.21 Additional font descriptor entries for CIDFonts

KEY	TYPE	VALUE
Style	dictionary	*(Optional)* A dictionary containing entries that describe the style of the glyphs in the font (see "Style," above).
Lang	name	*(Optional)* A name specifying the language of the font, used for encodings where the language is not implied by the encoding itself. The possible values are the codes defined by Internet RFC 3066, *Tags for the Identification of Languages* (see the Bibliography). If this entry is absent, the language is considered to be unknown. ***Note:** This specification for the allowable language codes is introduced in PDF 1.5. Prior versions supported a subset: the 2-character language codes defined by ISO 639 (see the Bibliography).*
FD	dictionary	*(Optional)* A dictionary whose keys identify a class of glyphs in a CIDFont. Each value is a dictionary containing entries that override the corresponding values in the main font descriptor dictionary for that class of glyphs (see "FD," below).
CIDSet	stream	*(Optional)* A stream identifying which CIDs are present in the CIDFont file. If this entry is present, the CIDFont contains only a subset of the glyphs in the character collection defined by the **CIDSystemInfo** dictionary. If it is absent, the only indication of a CIDFont subset is the subset tag in the **FontName** entry (see Section 5.5.3, "Font Subsets"). The stream's data is organized as a table of bits indexed by CID. The bits should be stored in bytes with the high-order bit first. Each bit corresponds to a CID. The most significant bit of the first byte corresponds to CID 0, the next bit to CID 1, and so on.

Style

The **Style** dictionary contains entries that define style attributes and values for the CIDFont. Currently, only the **Panose** entry is defined. The value of **Panose** is a 12-byte string consisting of the following:

- The font family class and subclass ID bytes, given in the sFamilyClass field of the "OS/2" table in a TrueType font. This field is documented in Microsoft's *TrueType 1.0 Font Files Technical Specification*.
- Ten bytes for the PANOSE classification number for the font. The PANOSE classification system is documented in Hewlett-Packard Company's *PANOSE Classification Metrics Guide*.

See the Bibliography for more information about these documents.

The following is an example of a **Style** entry in the font descriptor:

```
/Style << /Panose <01 05 02 02 03 00 00 00 00 00 00 00> >>
```

FD

A CIDFont may be made up of different classes of glyphs, each class requiring different sets of the fontwide attributes that appear in font descriptors. Latin glyphs, for example, may require different attributes than kanji glyphs. The font descriptor defines a set of default attributes that apply to all glyphs in the CIDFont; the **FD** entry in the font descriptor contains exceptions to these defaults.

The key for each entry in an **FD** dictionary is the name of a class of glyphs—that is, a particular subset of the CIDFont's character collection. The entry's value is a font descriptor whose contents are to override the fontwide attributes for that class only. This font descriptor should contain entries for metric information only; it should not include **FontFile**, **FontFile2**, **FontFile3**, or any of the entries listed in Table 5.21.

It is strongly recommended that the **FD** dictionary contain at least the metrics for the proportional Latin glyphs. With the information for these glyphs, a more accurate substitution font can be created.

The names of the glyph classes depend on the character collection, as identified by the **Registry**, **Ordering**, and **Supplement** entries in the **CIDSystemInfo** dictionary. Table 5.22 lists the valid keys for the Adobe-GB1, Adobe-CNS1, Adobe-Japan1, Adobe-Japan2, and Adobe-Korea1 character collections.

TABLE 5.22 Glyph classes in CJK fonts

CHARACTER COLLECTION	CLASS	GLYPHS IN CLASS
Adobe-GB1	**Alphabetic**	Full-width Latin, Greek, and Cyrillic glyphs
	Dingbats	Special symbols
	Generic	Typeface-independent glyphs, such as line-drawing
	Hanzi	Full-width hanzi (Chinese) glyphs
	HRoman	Half-width Latin glyphs
	HRomanRot	Same as **HRoman**, but rotated for use in vertical writing
	Kana	Japanese kana (katakana and hiragana) glyphs
	Proportional	Proportional Latin glyphs
	ProportionalRot	Same as **Proportional**, but rotated for use in vertical writing
Adobe-CNS1	**Alphabetic**	Full-width Latin, Greek, and Cyrillic glyphs
	Dingbats	Special symbols
	Generic	Typeface-independent glyphs, such as line-drawing
	Hanzi	Full-width hanzi (Chinese) glyphs
	HRoman	Half-width Latin glyphs
	HRomanRot	Same as **HRoman**, but rotated for use in vertical writing
	Kana	Japanese kana (katakana and hiragana) glyphs
	Proportional	Proportional Latin glyphs
	ProportionalRot	Same as **Proportional**, but rotated for use in vertical writing
Adobe-Japan1	**Alphabetic**	Full-width Latin, Greek, and Cyrillic glyphs
	AlphaNum	Numeric glyphs
	Dingbats	Special symbols
	DingbatsRot	Same as **Dingbats**, but rotated for use in vertical writing
	Generic	Typeface-independent glyphs, such as line-drawing
	GenericRot	Same as **Generic**, but rotated for use in vertical writing
	HKana	Half-width kana (katakana and hiragana) glyphs
	HKanaRot	Same as **HKana**, but rotated for use in vertical writing
	HRoman	Half-width Latin glyphs
	HRomanRot	Same as **HRoman**, but rotated for use in vertical writing
	Kana	Full-width kana (katakana and hiragana) glyphs
	Kanji	Full-width kanji (Chinese) glyphs
	Proportional	Proportional Latin glyphs
	ProportionalRot	Same as **Proportional**, but rotated for use in vertical writing
	Ruby	Glyphs used for setting ruby (small glyphs that serve to annotate other glyphs with meanings or readings)
Adobe-Japan2	**Alphabetic**	Full-width Latin, Greek, and Cyrillic glyphs
	Dingbats	Special symbols
	HojoKanji	Full-width kanji glyphs

CHARACTER COLLECTION	**CLASS**	**GLYPHS IN CLASS**
Adobe-Korea1	**Alphabetic**	Full-width Latin, Greek, and Cyrillic glyphs
	Dingbats	Special symbols
	Generic	Typeface-independent glyphs, such as line-drawing
	Hangul	Hangul and jamo glyphs
	Hanja	Full-width hanja (Chinese) glyphs
	HRoman	Half-width Latin glyphs
	HRomanRot	Same as **HRoman**, but rotated for use in vertical writing
	Kana	Japanese kana (katakana and hiragana) glyphs
	Proportional	Proportional Latin glyphs
	ProportionalRot	Same as **Proportional**, but rotated for use in vertical writing

Example 5.14 illustrates an **FD** dictionary containing two entries.

Example 5.14

```
/FD   <<  /Proportional  25 0 R
          /HKana  26 0 R
     >>

25  0  obj
   <<  /Type  /FontDescriptor
       /FontName  /HeiseiMin–W3–Proportional
       /Flags  2
       /AvgWidth  478
       /MaxWidth  1212
       /MissingWidth  250
       /StemV  105
       /StemH  45
       /CapHeight  660
       /XHeight  394
       /Ascent  720
       /Descent  –270
       /Leading  83
   >>
endobj
```

```
26 0 obj
   << /Type /FontDescriptor
       /FontName /HeiseiMin–W3–HKana
       /Flags 3
       /Style << /Panose <01 05 02 02 03 00 00 00 00 00 00 00> >>
       /AvgWidth 500
       /MaxWidth 500
       /MissingWidth 500
       /StemV 50
       /StemH 75
       /Ascent 720
       /Descent 0
       /Leading 83
   >>
endobj
```

5.8 Embedded Font Programs

A font program can be embedded in a PDF file as data contained in a PDF stream object. Such a stream object is also called a *font file*, by analogy with font programs that are available from sources external to the viewer application. (See also implementation note 61 in Appendix H.)

Font programs are subject to copyright, and the copyright owner may impose conditions under which a font program can be used. These permissions are recorded either in the font program itself or as part of a separate license. One of the conditions may be that the font program cannot be embedded, in which case it should not be incorporated into a PDF file. A font program may allow embedding for the sole purpose of viewing and printing the document, but not for creating new or modified text using the font (in either the same document or other documents); the latter operation would require the user performing the operation to have a licensed copy of the font program, not a copy extracted from the PDF file. In the absence of explicit information to the contrary, a PDF consumer should assume that any embedded font programs are to be used only to view and print the document and not for any other purposes.

Table 5.23 summarizes the ways in which font programs are embedded in a PDF file, depending on the representation of the font program. The key is the name used in the font descriptor to refer to the font file stream; the subtype is the value of the **Subtype** key, if present, in the font file stream dictionary. Further details of specific font program representations are given below.

TABLE 5.23 Embedded font organization for various font types

KEY	SUBTYPE	DESCRIPTION
FontFile	—	Type 1 font program, in the original (noncompact) format described in *Adobe Type 1 Font Format*. This entry can appear in the font descriptor for a **Type1** or **MMType1** font dictionary.
FontFile2	—	*(PDF 1.1)* TrueType font program, as described in the *TrueType Reference Manual*. This entry can appear in the font descriptor for a **TrueType** font dictionary or (in PDF 1.3) for a **CIDFontType2** CIDFont dictionary.
FontFile3	**Type1C**	*(PDF 1.2)* Type 1–equivalent font program represented in the Compact Font Format (CFF), as described in Adobe Technical Note #5176, *The Compact Font Format Specification*. This entry can appear in the font descriptor for a **Type1** or **MMType1** font dictionary.
	CIDFontType0C	*(PDF 1.3)* Type 0 CIDFont program represented in the Compact Font Format (CFF), as described in Adobe Technical Note #5176, *The Compact Font Format Specification*. This entry can appear in the font descriptor for a **CIDFontType0** CIDFont dictionary.

The stream dictionary for a font file contains the normal entries for a stream, such as **Length** and **Filter** (listed in Table 3.4 on page 38), plus the additional entries listed in Table 5.24.

TABLE 5.24 Additional entries in an embedded font stream dictionary

KEY	TYPE	VALUE
Length1	integer	*(Required for Type 1 and TrueType fonts)* The length in bytes of the clear-text portion of the Type 1 font program (see below), or the entire TrueType font program, after it has been decoded using the filters specified by the stream's **Filter** entry, if any.
Length2	integer	*(Required for Type 1 fonts)* The length in bytes of the encrypted portion of the Type 1 font program (see below) after it has been decoded using the filters specified by the stream's **Filter** entry.
Length3	integer	*(Required for Type 1 fonts)* The length in bytes of the fixed-content portion of the Type 1 font program (see below), after it has been decoded using the filters specified by the stream's **Filter** entry. If **Length3** is 0, it indicates that the 512 zeros and **cleartomark** have not been included in the **FontFile** font program and must be added.

KEY	TYPE	VALUE
Subtype	name	*(Required if referenced from* ***FontFile3****; PDF 1.2)* A name specifying the format of the embedded font program. The name must be **Type1C** for Type 1 compact fonts or **CIDFontType0C** for Type 0 compact CIDFonts. When additional font formats are added to PDF, more values will be defined for **Subtype**.
Metadata	stream	*(Optional; PDF 1.4)* A *metadata stream* containing metadata for the embedded font program (see Section 10.2.2, "Metadata Streams").

A standard Type 1 font program, as described in the *Adobe Type 1 Font Format* specification, consists of three parts: a clear-text portion (written using PostScript syntax), an encrypted portion, and a fixed-content portion. The fixed-content portion contains 512 ASCII zeros followed by a **cleartomark** operator, and perhaps followed by additional data. While the encrypted portion of a standard Type 1 font may be in binary or ASCII hexadecimal format, PDF supports only the binary format; however, the entire font program may be encoded using any filters.

Example 5.15 shows the structure of an embedded standard Type 1 font.

Example 5.15

```
12 0 obj
   << /Filter /ASCII85Decode
      /Length 41116
      /Length1 2526
      /Length2 32393
      /Length3 570
   >>
stream
,p>`rDKJj'E+LaU0eP.@+AH9dBOu$hFD55nC
…Omitted data…
JJQ&Nt')<=^p&mGf(%:%h1%9c//K(/*o=.C>UXkbVGTrr~>
endstream
endobj
```

As noted in Table 5.23, a Type 1–equivalent font program or a Type 0 CIDFont program can be represented in the Compact Font Format (CFF). The **Length1**, **Length2**, and **Length3** entries are not needed in that case. Although CFF enables multiple font or CIDFont programs to be bundled together in a single file, an embedded CFF font file in PDF must consist of exactly one font or CIDFont (as appropriate for the associated font dictionary).

***Note:** According to the* Adobe Type 1 Font Format *specification, a Type 1 font program may contain a **PaintType** entry specifying whether the glyphs' outlines are to be filled or stroked. For fonts embedded in a PDF file, this entry is ignored; the decision whether to fill or stroke glyph outlines is entirely determined by the PDF text rendering mode parameter (see Section 5.2.5, "Text Rendering Mode"). This also applies to Type 1 compact fonts and Type 0 compact CIDFonts.*

A TrueType font program may be used as part of either a font or a CIDFont. Although the basic font file format is the same in both cases, there are different requirements for what information must be present in the font program. The following TrueType tables are always required: "head," "hhea," "loca," "maxp," "cvt_," "prep," "glyf," "hmtx," and "fpgm." If used with a simple font dictionary, the font program must additionally contain a "cmap" table defining one or more encodings, as discussed in "Encodings for TrueType Fonts" on page 391. If used with a CIDFont dictionary, the "cmap" table is not needed, since the mapping from character codes to glyph descriptions is provided separately.

***Note:** The "vhea" and "vmtx" tables that specify vertical metrics are never used by a PDF viewer application. The only way to specify vertical metrics in PDF is by means of the **DW2** and **W2** entries in a CIDFont dictionary.*

As discussed in Section 5.5.3, "Font Subsets," an embedded font program may contain only the subset of glyphs that are used in the PDF document. This may be indicated by the presence of a **CharSet** or **CIDSet** entry in the font descriptor that refers to the font file, although subset fonts are not always so identified.

5.9 Extraction of Text Content

The preceding sections describe all the facilities for showing text and causing glyphs to be painted on the page. In addition to displaying text, viewer applications sometimes need to determine the information content of text—that is, its meaning according to some standard character identification, as opposed to its rendered appearance. This need arises during operations such as searching, indexing, and exporting of text to other applications.

The Unicode standard defines a system for numbering all of the common characters used in a large number of languages. It is a suitable scheme for representing the information content of text, but not its appearance, since Unicode values identify characters, not glyphs. For information about Unicode, see the *Unicode Standard* by the Unicode Consortium (see the Bibliography).

When extracting character content, a viewer application can easily convert text to Unicode values if a font's characters are identified according to a standard character set that is known to the viewer. This character identification can occur if either the font uses a standard named encoding or the characters in the font are identified by standard character names or CIDs. in a well-known collection. Section 5.9.1, "Mapping Character Codes to Unicode Values," describes in detail the overall algorithm for mapping character codes to Unicode values.

If a font is not defined in one of these ways, the glyphs can still be shown, but the characters cannot be converted to Unicode values without additional information:

- This information can be provided as an optional **ToUnicode** entry in the font dictionary *(PDF 1.2*; see Section 5.9.2, "ToUnicode CMaps"*)*, whose value is a stream object containing a special kind of CMap file that maps character codes to Unicode values.
- An **ActualText** entry for a structure element or marked-content sequence (see Section 10.8.3, "Replacement Text") can be used to specify the text content directly.

5.9.1 Mapping Character Codes to Unicode Values

A viewer application can use the following methods, in the priority given, to map a character code to a Unicode value. Tagged PDF documents, in particular, must provide at least one of these methods (see "Unicode Mapping in Tagged PDF" on page 760).

- If the font dictionary contains a **ToUnicode** CMap (see Section 5.9.2, "ToUnicode CMaps"), use that CMap to convert the character code to Unicode.
- If the font is a simple font that uses one of the predefined encodings **MacRomanEncoding**, **MacExpertEncoding**, or **WinAnsiEncoding**, or that has an encoding whose **Differences** array includes only character names taken from the Adobe standard Latin character set and the set of named characters in the Symbol font (see Appendix D):
  1. Map the character code to a character name according to Table D.1 on page 868 and the font's **Differences** array.
  2. Look up the character name in the *Adobe Glyph List* (see the Bibliography) to obtain the corresponding Unicode value.

- If the font is a composite font that uses one of the predefined CMaps listed in Table 5.15 on page 404 (except Identity–H and Identity–V), or whose descendant CIDFont uses the Adobe-GB1, Adobe-CNS1, Adobe-Japan1, or Adobe-Korea1 character collection:

    1. Map the character code to a character identifier (CID) according to the font's CMap.
    2. Obtain the registry and ordering of the character collection used by the font's CMap (for example, Adobe and Japan1) from its **CIDSystemInfo** dictionary.
    3. Construct a second CMap name by concatenating the registry and ordering obtained in step 2 in the format *registry*–*ordering*–UCS2 (for example, Adobe–Japan1–UCS2).
    4. Obtain the CMap with the name constructed in step 3 (available from the ASN Web site; see the Bibliography).
    5. Map the CID obtained in step 1 according to the CMap obtained in step 4, producing a Unicode value.

Note: *Type 0 fonts whose descendant CIDFonts use the Adobe-GB1, Adobe-CNS1, Adobe-Japan1, or Adobe-Korea1 character collection (as specified in the* **CIDSystemInfo** *dictionary) must have a supplement number corresponding to the version of PDF supported by the viewer. See Table 5.16 on page 407 for a list of the character collections corresponding to a given PDF version. (Other supplements of these character collections can be used, but if the supplement is higher-numbered than the one corresponding to the supported PDF version, only the CIDs in the latter supplement are considered to be standard CIDs.)*

If these methods fail to produce a Unicode value, there is no way to determine what the character code represents.

5.9.2 ToUnicode CMaps

The CMap defined in the **ToUnicode** entry of the font dictionary must follow the syntax for CMaps introduced in Section 5.6.4, "CMaps" and fully documented in Adobe Technical Note #5014, *Adobe CMap and CIDFont Files Specification*. Additional guidance regarding the CMap defined in this entry is provided in Adobe

Technical Note #5411, *ToUnicode Mapping File Tutorial*. This CMap differs from an ordinary one in the following ways:

- The only pertinent entry in the CMap stream dictionary (see Table 5.17 on page 410) is **UseCMap**, which may be used if the CMap is based on another **ToUnicode** CMap.
- The CMap file must contain **begincodespacerange** and **endcodespacerange** operators that are consistent with the encoding that the font uses. In particular, for a simple font, the codespace must be one byte long.
- It must use the **beginbfchar**, **endbfchar**, **beginbfrange**, and **endbfrange** operators to define the mapping from character codes to Unicode character sequences expressed in UTF-16BE encoding.

Example 5.16 illustrates a Type 0 font that uses the Identity–H CMap to map from character codes to CIDs, and whose descendant CIDFont uses the **Identity** mapping from CIDs to TrueType glyph indices. Text strings shown using this font simply use a 2-byte glyph index for each glyph. In the absence of a **ToUnicode** entry, there would be no information available about what the glyphs mean.

Example 5.16

```
14 0 obj
   << /Type /Font
      /Subtype /Type0
      /BaseFont /Ryumin–Light
      /Encoding /Identity–H
      /DescendantFonts [15 0 R]
      /ToUnicode 16 0 R
   >>
endobj

15 0 obj
   << /Type /Font
      /Subtype /CIDFontType2
      /BaseFont /Ryumin–Light
      /CIDSystemInfo 17 0 R
      /FontDescriptor 18 0 R
      /CIDToGIDMap /Identity
   >>
endobj
```

The value of the **ToUnicode** entry is a stream object that contains the definition of the CMap, as shown in Example 5.17.

Example 5.17

```
16 0 obj
   << /Length 433 >>
stream
/CIDInit /ProcSet findresource begin
12 dict begin
begincmap
/CIDSystemInfo
<< /Registry (Adobe)
/Ordering (UCS)
/Supplement 0
>> def
/CMapName /Adobe–Identity–UCS def
/CMapType 2 def
1 begincodespacerange
<0000> <FFFF>
endcodespacerange
2 beginbfrange
<0000> <005E> <0020>
<005F> <0061> [<00660066> <00660069> <00660066006C>]
endbfrange
1 beginbfchar
<3A51> <D840DC3E>
endbfchar
endcmap
CMapName currentdict /CMap defineresource pop
end
end
endstream
endobj
```

The **begincodespacerange** and **endcodespacerange** operators in Example 5.17 define the source character code range to be the 2-byte character codes from <00 00> to <FF FF>. The specific mappings for several of the character codes are shown. For example, <00 00> to <00 5E> are mapped to the Unicode values U+0020 to U+007E (where Unicode values are conventionally written as U+ followed by four to six hexadecimal digits). This is followed by the definition of a mapping used where each character code represents more than one Unicode value:

```
<005F> <0061> [<00660066> <00660069> <00660066006C>]
```

In this case, the original character codes are the glyph indices for the ligatures ﬀ, ﬁ, and ﬄ. The entry defines the mapping from the character codes <00 5F>, <00 60>, and <00 61> to the strings of Unicode values with a Unicode scalar value for each character in the ligature: U+0066 U+0066 are the Unicode values for the character sequence f f, U+0066 U+0069 for f i, and U+0066 U+0066 U+006c for f f l.

Finally, the character code <3A 51> is mapped to the Unicode value U+2003E, which is expressed by the byte sequence <D840DC3E> in UTF-16BE encoding.

Example 5.17 illustrates several extensions to the way destination values can be defined. To support mappings from a source code to a string of destination codes, the following extension has been made to the ranges defined after a **beginbfchar** operator:

n **beginbfchar**
srcCode dstString
endbfchar

where *dstString* can be a string of up to 512 bytes. Likewise, mappings after the **beginbfrange** operator may be defined as:

n **beginbfrange**
$srcCode_1$ $srcCode_2$ *dstString*
endbfrange

In this case, the last byte of the string will be incremented for each consecutive code in the source code range. When defining ranges of this type, care must be taken to ensure that the value of the last byte in the string is less than or equal to $255 - (srcCode_2 - srcCode_1)$. This ensures that the last byte of the string will not be incremented past 255; otherwise the result of mapping is undefined and an error occurs.

To support more compact representations of mappings from a range of source character codes to a discontiguous range of destination codes, the CMaps used for the **ToUnicode** entry may use the following syntax for the mappings following a **beginbfrange** definition:

> *n* **beginbfrange**
> *srcCode*$_1$ *srcCode*$_n$ [*dstString*$_1$ *dstString*$_2$ … *dstString*$_n$]
> **endbfrange**

Consecutive codes starting with *srcCode*$_1$ and ending with *srcCode*$_n$ are mapped to the destination strings in the array starting with *dstString*$_1$ and ending with *dstString*$_n$.

CHAPTER 6

Rendering

THE ADOBE IMAGING MODEL separates *graphics* (the specification of shapes and colors) from *rendering* (controlling a raster output device). Figures 4.12 and 4.13 on pages 207 and 208 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; PDF documents 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 the control of a PDF document have to do with the reproduction of color. Colors are rendered 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 (see Section 4.5.4, "CIE-Based Color Spaces"), it must first be transformed to the *native color space* of the raster output device (also called its *process color model*).
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), a *color conversion function* is invoked.
3. The device color values are now mapped 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, a *halftone function* is invoked, which approximates the desired colors by means of patterns of pixels.

5. Finally, *scan conversion* is performed to mark the appropriate pixels of the raster output device with the requested colors.

Once these operations have been performed for all graphics objects on the page, the resulting raster data is used to mark the physical output medium, such as pixels on a display or ink on a printed page. A PDF document specifies very little about the properties of the physical medium on which the output will be produced; that information is obtained from the following sources:

- The media box and a few other entries in the page dictionary (see Section 10.10.1, "Page Boundaries").
- An interactive dialog conducted when the user requests viewing or printing.
- A *job ticket*, either embedded in the PDF file or provided separately, specifying detailed instructions for imposing PDF pages onto media and for controlling special features of the output device. Various standards exist for the format of job tickets; two of them, JDF (Job Definition Format) and PJTF (Portable Job Ticket Format), are described in the CIP4 document *JDF Specification* and in Adobe Technical Note #5620, *Portable Job Ticket Format* (see the Bibliography).

Some of the rendering facilities described in this chapter are controlled by device-dependent graphics state parameters, listed in Table 4.3 on page 182. These parameters can be changed by invoking the **gs** operator with a parameter dictionary containing entries shown in Table 4.8 on page 190.

6.1 CIE-Based Color to Device Color

To render CIE-based colors on an output device, the viewer application must convert from the specified CIE-based color space to the device's native color space (typically **DeviceGray**, **DeviceRGB**, or **DeviceCMYK**), taking into account the known properties of the device. As discussed in Section 4.5.4, "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. CIE-based color specification and rendering are a feature of PDF 1.1 (**CalGray**, **CalRGB**, and **Lab**) and PDF 1.3 (**ICCBased**).

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. The colors that a device can produce are characterized by a *device profile*, which is usually specified by an ICC profile associated with the device (and entirely separate from the profile that is specified in an **ICCBased** color space).

Note: *PDF has no equivalent of the PostScript color rendering dictionary. The means by which a device profile is associated with a viewer application's output device are implementation-dependent and cannot be specified in a PDF file. Typically, this is done through a color management system (CMS) that is provided by the operating system. Beginning with PDF 1.4, a PDF document can also specify one or more* output intents *providing possible profiles that might be used to process the document (see Section 10.10.4, "Output Intents").*

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. An output 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 "Rendering Intents" on page 230).
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 **DefaultGray**, **DefaultRGB**, and **DefaultCMYK** color space resources; see "Default Color Spaces" on page 227).

The source gamut is specified by a page description when it selects a CIE-based color space. This specification is device-independent. The corresponding properties of the output device are given in the device profile associated with the device. The gamut mapping and color mapping functions are part of the implementation of the viewer application.

6.2 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), *RGB* (red-green-blue), or *CMYK* (cyan-magenta-yellow-black) model. If the device supports continuous-tone output, reproduction occurs directly. Otherwise, it is accomplished by means of halftoning.

A device's native color space is also called its *process color model*. Process colors are ones that are produced by combinations of one or more standard *process colorants*. Colors specified in any device or CIE-based color space are rendered as process colors. (A device can also support additional *spot colorants*, which can be painted only by means of **Separation** or **DeviceN** color spaces. They are not involved in the rendering of device or CIE-based color spaces, nor are they subject to the conversions described below.)

***Note:** Some devices provide a native color space that is not one of the three named above but consists of a different combination of colorants. In that case, conversion from the standard device color spaces to the device's native color space is performed by device-dependent means.*

Knowing the native color space and other output capabilities of the device, the viewer application can automatically convert the color values specified in a document to those appropriate for the device's native color space. For example, if a document 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 viewer application performs the necessary conversions. If the document specifies colors directly in the device's native color space, no conversions are necessary.

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 "Default Color Spaces" on page 227).

6.2.1 Conversion between DeviceGray and DeviceRGB

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

$$\begin{aligned} \mathit{red} &= \mathit{gray} \\ \mathit{green} &= \mathit{gray} \\ \mathit{blue} &= \mathit{gray} \end{aligned}$$

The gray value for a given *RGB* value is computed according to the NTSC video standard, which determines how a color television signal is rendered on a black-and-white television set:

$$\mathit{gray} = 0.3 \times \mathit{red} + 0.59 \times \mathit{green} + 0.11 \times \mathit{blue}$$

6.2.2 Conversion between DeviceGray and DeviceCMYK

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

$$\begin{aligned} \mathit{cyan} &= 0.0 \\ \mathit{magenta} &= 0.0 \\ \mathit{yellow} &= 0.0 \\ \mathit{black} &= 1.0 - \mathit{gray} \end{aligned}$$

To obtain the equivalent gray level for a given *CMYK* value, the contributions of all components must be taken into account:

$$\mathit{gray} = 1.0 - \min(1.0,\ 0.3 \times \mathit{cyan} + 0.59 \times \mathit{magenta} + 0.11 \times \mathit{yellow} + \mathit{black})$$

The interactions between the black component and the other three are elaborated below.

6.2.3 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:

$$\begin{aligned} \mathit{cyan} &= 1.0 - \mathit{red} \\ \mathit{magenta} &= 1.0 - \mathit{green} \\ \mathit{yellow} &= 1.0 - \mathit{blue} \end{aligned}$$

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, however, 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:

$$\begin{aligned} c &= 1.0 - \mathit{red} \\ m &= 1.0 - \mathit{green} \\ y &= 1.0 - \mathit{blue} \\ k &= \min(c, m, y) \end{aligned}$$

$$\begin{aligned} \mathit{cyan} &= \min(1.0, \max(0.0, c - UCR(k))) \\ \mathit{magenta} &= \min(1.0, \max(0.0, m - UCR(k))) \\ \mathit{yellow} &= \min(1.0, \max(0.0, y - UCR(k))) \\ \mathit{black} &= \min(1.0, \max(0.0, BG(k))) \end{aligned}$$

In PDF 1.2, the black-generation and undercolor-removal functions are defined as PDF function dictionaries (see Section 3.9, "Functions") that are parameters in the graphics state. They are specified as the values of the **BG** and **UCR** (or **BG2** and **UCR2**) entries in a graphics state parameter dictionary (see Table 4.8 on page 190). Each function is called with a single numeric operand and is expected to return a single numeric result.

The input of both the black-generation and undercolor-removal functions 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 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 colorant.

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 the *min* and *max* operations in the formulas 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.

See Section 7.6.4, "Rendering Parameters and Transparency," and in particular "Rendering Intent and Color Conversions" on page 533, for further discussion of the role of black-generation and undercolor-removal functions in the transparent imaging model.

6.2.4 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:

$$\begin{aligned}
\textit{red} &= 1.0 - \min(1.0,\ \textit{cyan} + \textit{black}) \\
\textit{green} &= 1.0 - \min(1.0,\ \textit{magenta} + \textit{black}) \\
\textit{blue} &= 1.0 - \min(1.0,\ \textit{yellow} + \textit{black})
\end{aligned}$$

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.

6.3 Transfer Functions

In PDF 1.2, 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 viewer application 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, a **DeviceRGB** or **CalRGB** color space.) The transfer function is called with a numeric operand in the range 0.0 to 1.0 and must return a number in the same range. The input is the value of a color component in the device's native color space, either specified directly or produced by conversion from some other color space. The output is the transformed component value to be transmitted to the device (after halftoning, if necessary).

Both the input and the output of a transfer function are always interpreted as if the corresponding 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), it is converted to additive form by subtracting it from 1.0 before it is passed to the transfer function. The output of the function is always in additive form, and is passed on to the halftone function in that form.

In PDF 1.2, transfer functions are defined as PDF function objects (see Section 3.9, "Functions"). There are two ways to specify transfer functions:

- The *current transfer function* parameter in the graphics state consists of either a single transfer function or an array of four separate transfer functions, one each for red, green, blue, and gray or their complements cyan, magenta, yellow, and black. (If only a single function is specified, it applies to all components.) An *RGB* device uses the first three; a monochrome device uses the gray transfer function only; and a *CMYK* device uses all four. The current transfer function can be specified as the value of the **TR** or **TR2** entry in a graphics state parameter dictionary; see Table 4.8 on page 190.
- The *current halftone* parameter in the graphics state can specify transfer functions as optional entries in *halftone dictionaries* (see Section 6.4.4, "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 uses components other than the ones listed above. A transfer function specified in a halftone dictionary overrides the corresponding one specified by the current transfer function parameter in the graphics state.

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 PostScript calculator 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 a transfer function to obtain special effects on monochrome devices, and applies only to colors specified in the* **DeviceGray** *color space.*

See Section 7.6.4, "Rendering Parameters and Transparency," and in particular "Halftone and Transfer Function" on page 532, for further discussion of the role of transfer functions in the transparent imaging model.

6.4 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.

PDF provides a high degree of control over details of the halftoning process. For example, in color printing, independent halftone screens can 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 a PDF document provides its own halftone specifications, it sacrifices portability. Associated with every output device is a default halftone definition that is appropriate for most purposes. Only relatively sophisticated documents 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 PDF document that defines a new halftone must make assumptions about 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 the effects of electronic or mechanical noise.

6.4.1 Halftone Screens

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. A screen is defined by conceptually laying a uniform rectangular grid over the device pixel array. Each pixel belongs to one cell of the grid; a single cell typically contains many pixels. The screen grid is defined entirely in device space, and is unaffected by modifications to the current transformation matrix. This property is essential to ensure that adjacent areas colored by halftones are properly stitched together without visible "seams."

On a bilevel (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 white pixels to the total number of pixels in the cell. A cell containing n pixels can render $n + 1$ different gray levels, ranging from all pixels black to all 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 additively (*RGB* or gray) or subtractively (*CMYK* or tint). Larger values of a color component represent lighter colors—greater intensity in an additive device such as a display, or less ink in a subtractive device such as a printer. Transfer functions produce color values in additive form; see Section 6.3, "Transfer Functions."

6.4.2 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 along with some additional ones. The order in which pixels change from black to white for increasing gray levels is determined by a *spot function*, which specifies that order in an indirect way that minimizes interactions with the screen frequency and angle.

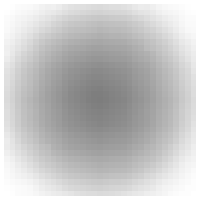
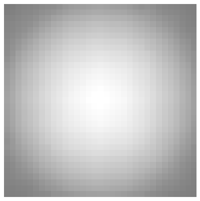
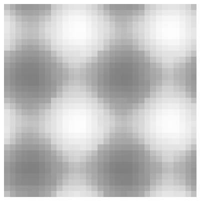
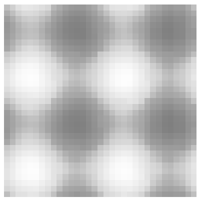
Consider a halftone cell to have its own coordinate system: the center of the cell is the origin and the corners are at coordinates ±1.0 horizontally and vertically. Each pixel in the cell is centered at horizontal and vertical coordinates that both lie in the range −1.0 to +1.0. For each pixel, the spot function is invoked with the pixel's coordinates as input and must return a single number in the range −1.0 to +1.0, defining the pixel's position in the whitening order.

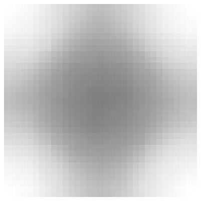
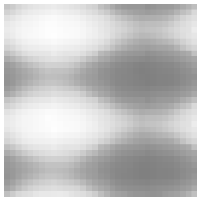
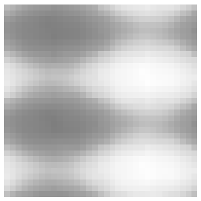
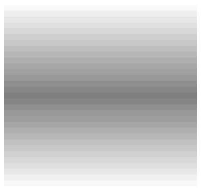
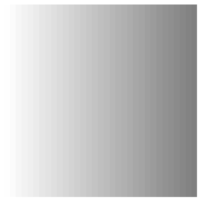
The specific values the spot function returns are not significant; all that matters are the *relative* values returned for different pixels. As a cell's gray level varies from black to white, the first pixel whitened is the one for which the spot function returns the lowest value, the next pixel is the 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.

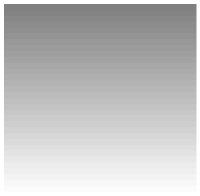
PDF provides built-in definitions for many of the most commonly used spot functions. A halftone can simply specify any of these predefined spot functions by name instead of giving an explicit function definition. For example, the name **SimpleDot** designates a spot function whose value is inversely related to a pixel's distance from the center of the halftone cell. This produces a "dot screen" in which the black pixels are clustered within a circle whose area is inversely proportional to the gray level. The predefined function **Line** is a spot function whose value is the distance from a given pixel to a line through the center of the cell, producing a "line screen" in which the white pixels grow away from that line.

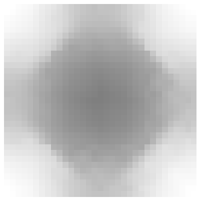
Table 6.1 shows the predefined spot functions. The table gives the mathematical definition of each function along with the corresponding PostScript language code as it would be defined in a PostScript calculator function (see Section 3.9.4, “Type 4 (PostScript Calculator) Functions”). The image accompanying each function shows how the relative values of the function are distributed over the halftone cell, indicating the approximate order in which pixels are whitened; pixels corresponding to darker points in the image are whitened later than those corresponding to lighter points. (See implementation note 62 in Appendix H.)

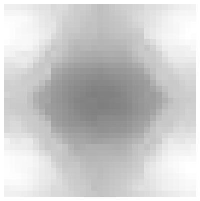
TABLE 6.1 Predefined spot functions

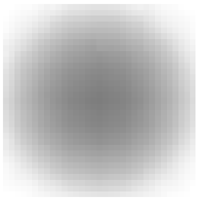
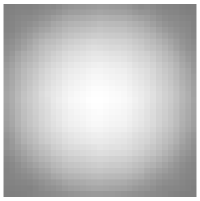
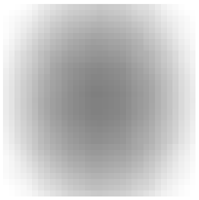
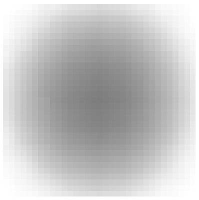
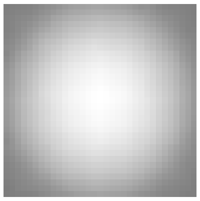
NAME	APPEARANCE	DEFINITION
SimpleDot		$1-(x^2+y^2)$ `{ dup mul exch dup mul add 1 exch sub }`
InvertedSimpleDot		x^2+y^2-1 `{ dup mul exch dup mul add 1 sub }`
DoubleDot		$\frac{\sin(360\times x)}{2}+\frac{\sin(360\times y)}{2}$ `{ 360 mul sin 2 div exch 360 mul sin 2 div add }`
InvertedDoubleDot		$-\left(\frac{\sin(360\times x)}{2}+\frac{\sin(360\times y)}{2}\right)$ `{ 360 mul sin 2 div exch 360 mul sin 2 div add neg }`

NAME	APPEARANCE	DEFINITION
CosineDot		$\frac{\cos(180 \times x)}{2} + \frac{\cos(180 \times y)}{2}$ { 180 mul cos exch 180 mul cos add 2 div }
Double		$\frac{\sin\left(360 \times \frac{x}{2}\right)}{2} + \frac{\sin(360 \times y)}{2}$ { 360 mul sin 2 div exch 2 div 360 mul sin 2 div add }
InvertedDouble		$-\left(\frac{\sin\left(360 \times \frac{x}{2}\right)}{2} + \frac{\sin(360 \times y)}{2}\right)$ { 360 mul sin 2 div exch 2 div 360 mul sin 2 div add neg }
Line		$-\lvert y \rvert$ { exch pop abs neg }
LineX		x { pop }

NAME	APPEARANCE	DEFINITION
LineY		y `{ exch pop }`

Round		*if* $\lvert x\rvert + \lvert y\rvert \le 1$ *then* $1 - (x^2 + y^2)$
		else $(\lvert x\rvert - 1)^2 + (\lvert y\rvert - 1)^2 - 1$
		`{ abs exch abs`
		`  2 copy add 1 le`
		`    { dup mul exch dup mul add 1 exch sub }`
		`    { 1 sub dup mul exch 1 sub dup mul add 1 sub }`
		`  ifelse }`

Ellipse		*let* $w = (3 \times \lvert x\rvert) + (4 \times \lvert y\rvert) - 3$
		if $w < 0$ *then* $1 - \dfrac{x^2 + \left(\dfrac{\lvert y\rvert}{0.75}\right)^2}{4}$
		else if $w > 1$ *then* $\dfrac{(1 - \lvert x\rvert)^2 + \left(\dfrac{1 - \lvert y\rvert}{0.75}\right)^2}{4} - 1$
		else $0.5 - w$
		`{ abs exch abs 2 copy 3 mul exch 4 mul add 3 sub dup 0 lt`
		`    { pop dup mul exch 0.75 div dup mul add`
		`      4 div 1 exch sub }`
		`    { dup 1 gt`
		`        { pop 1 exch sub dup mul`
		`          exch 1 exch sub 0.75 div dup mul add`
		`          4 div 1 sub }`
		`        { 0.5 exch sub exch pop exch pop }`
		`      ifelse }`
		`  ifelse }`

NAME	APPEARANCE	DEFINITION
EllipseA		$1 - (x^2 + 0.9 \times y^2)$ { dup mul 0.9 mul exch dup mul add 1 exch sub }
InvertedEllipseA		$x^2 + 0.9 \times y^2 - 1$ { dup mul 0.9 mul exch dup mul add 1 sub }
EllipseB		$1 - \sqrt{x^2 + \frac{5}{8} \times y^2}$ { dup 5 mul 8 div mul exch dup mul exch add sqrt 1 exch sub }
EllipseC		$1 - (0.9 \times x^2 + y^2)$ { dup mul exch dup mul 0.9 mul add 1 exch sub }
InvertedEllipseC		$0.9 \times x^2 + y^2 - 1$ { dup mul exch dup mul 0.9 mul add 1 sub }

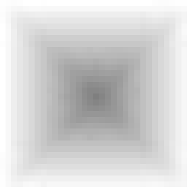
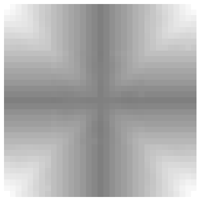
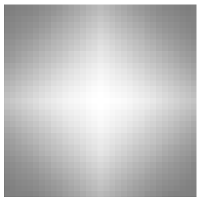
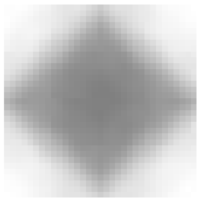
NAME	APPEARANCE	DEFINITION
Square		$-\max(\lvert x\rvert, \lvert y\rvert)$ `{ abs exch abs 2 copy lt { exch } if pop neg }`
Cross		$-\min(\lvert x\rvert, \lvert y\rvert)$ `{ abs exch abs 2 copy gt { exch } if pop neg }`
Rhomboid		$\dfrac{0.9 \times \lvert x\rvert + \lvert y\rvert}{2}$ `{ abs exch abs 0.9 mul add 2 div }`
Diamond		$\textit{if}\ \lvert x\rvert + \lvert y\rvert \le 0.75\ \textit{then}\ 1 - (x^2 + y^2)$ $\textit{else if}\ \lvert x\rvert + \lvert y\rvert \le 1.23\ \textit{then}\ 1 - (0.85 \times \lvert x\rvert + \lvert y\rvert)$ $\textit{else}\ (\lvert x\rvert - 1)^2 + (\lvert y\rvert - 1)^2 - 1$ `{ abs exch abs 2 copy add 0.75 le { dup mul exch dup mul add 1 exch sub } { 2 copy add 1.23 le { 0.85 mul add 1 exch sub } { 1 sub dup mul exch 1 sub dup mul add 1 sub } ifelse } ifelse }`

Figure 6.1 illustrates the effects of some of the predefined spot functions.

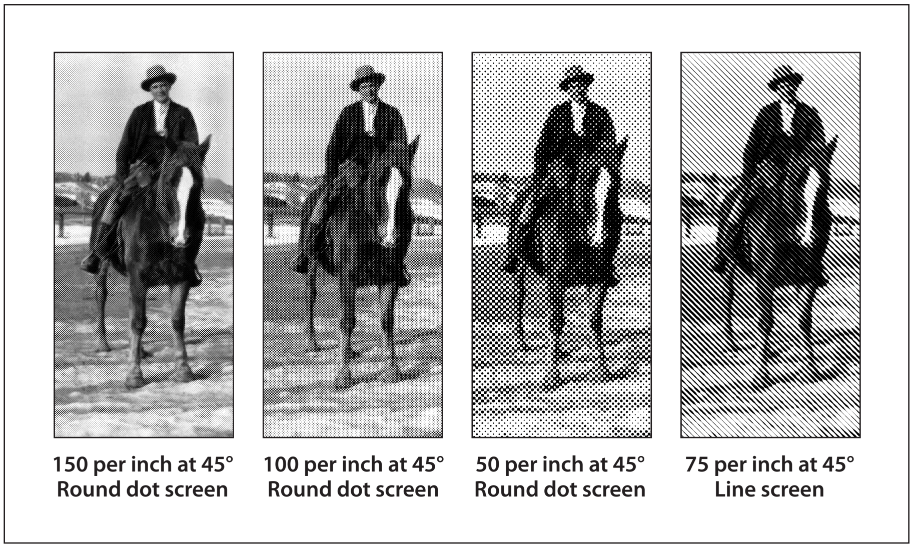


FIGURE 6.1 *Various halftoning effects*

6.4.3 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 arbitrary rectangles, 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, or 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 spot function is used to compute a threshold array internally.)

6.4.4 Halftone Dictionaries

In PDF 1.2, the graphics state includes a *current halftone* parameter, which determines the halftoning process to be used by the painting operators. The current halftone can be specified as the value of the **HT** entry in a graphics state parameter dictionary; see Table 4.8 on page 190. It may be defined by either a dictionary or a stream, depending on the type of halftone; the term *halftone dictionary* is used generically throughout this section to refer to either a dictionary object or the dictionary portion of a stream object. (Those halftones that are defined by streams are specifically identified as such in the descriptions of particular halftone types; unless otherwise stated, they are understood to be defined by simple dictionaries instead.)

Every halftone dictionary must have a **HalftoneType** entry whose value is an integer specifying the overall type of halftone definition. The remaining entries in the dictionary are interpreted according to this type. PDF supports the halftone types listed in Table 6.2.

TABLE 6.2 PDF halftone types

TYPE	MEANING
1	Defines a single halftone screen by a *frequency, angle,* and *spot function.*
5	Defines an arbitrary number of halftone screens, one for each colorant or color component (including both primary and spot colorants). The keys in this dictionary are names of colorants; the values are halftone dictionaries of other types, each defining the halftone screen for a single colorant.
6	Defines a single halftone screen by a threshold array containing 8-bit sample values.
10	Defines a single halftone screen by a threshold array containing 8-bit sample values, representing a halftone cell that may have a nonzero screen angle.
16	*(PDF 1.3)* Defines a single halftone screen by a threshold array containing 16-bit sample values, representing a halftone cell that may have a nonzero screen angle.

The dictionaries representing these halftone types contain the same entries as the corresponding PostScript language halftone dictionaries (as described in Section 7.4 of the *PostScript Language Reference*, Third Edition), with the following exceptions:

- The PDF dictionaries may contain a **Type** entry with the value **Halftone**, identifying the type of PDF object that the dictionary describes.
- Spot functions and transfer functions are represented by function objects instead of PostScript procedures.
- Threshold arrays are specified as streams instead of files.
- In type 5 halftone dictionaries, the keys for colorants must be name objects; they may not be strings as they may in PostScript.

Halftone dictionaries have an optional entry, **HalftoneName**, that identifies the desired halftone by name. In PDF 1.3, if this entry is present, all other entries, including **HalftoneType**, are optional. At rendering time, if the output device has a halftone with the specified name, that halftone will be used, overriding any other halftone parameters specified in the dictionary. This provides a way for PDF documents to select the proprietary halftones supplied by some device manufacturers, which would not otherwise be accessible because they are not explicitly defined in PDF. If there is no **HalftoneName** entry, or if the requested halftone name does not exist on the device, the halftone's parameters are defined by the other entries in the dictionary, if any. If no other entries are present, the default halftone is used.

See Section 7.6.4, "Rendering Parameters and Transparency," and in particular "Halftone and Transfer Function" on page 532, for further discussion of the role of halftones in the transparent imaging model.

Type 1 Halftones

Table 6.3 describes the contents of a halftone dictionary of type 1, which defines a halftone screen in terms of its frequency, angle, and spot function.

TABLE 6.3 Entries in a type 1 halftone dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Halftone** for a halftone dictionary.
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 1 for this type of halftone.
HalftoneName	string	*(Optional)* The name of the halftone dictionary.
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	function or name	*(Required)* A function object defining the order in which device pixels within a screen cell are adjusted for different gray levels, or the name of one of the predefined spot functions (see Table 6.1 on page 449).
AccurateScreens	boolean	*(Optional)* A flag specifying whether to invoke a special halftone algorithm that is extremely precise, but computationally expensive; see below for further discussion. Default value: **false**.

KEY	TYPE	VALUE
TransferFunction	function or name	*(Optional)* A transfer function, which overrides the current transfer function in the graphics state for the same component. This entry is required if the dictionary is a component of a type 5 halftone (see "Type 5 Halftones" on page 465) and represents either a nonprimary or nonstandard primary color component (see Section 6.3, "Transfer Functions"). The name **Identity** may be used to specify the identity function.

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.

In principle, PDF 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 happens, an error will occur. The **AccurateScreens** feature often requires very large amounts of memory to achieve the highest accuracy.

Example 6.1 shows a halftone dictionary for a type 1 halftone.

Example 6.1

```
28 0 obj
    << /Type /Halftone
        /HalftoneType 1
        /Frequency 120
        /Angle 30
        /SpotFunction /CosineDot
        /TransferFunction /Identity
    >>
endobj
```

Type 6 Halftones

A type 6 halftone defines a halftone screen with a threshold array. The halftone is represented as a stream containing the threshold values; the parameters defining the halftone are specified by entries in the stream dictionary. Table 6.4 shows the contents of this dictionary, in addition to the usual entries common to all streams (see Table 3.4 on page 38). The **Width** and **Height** entries specify the dimensions of the threshold array in device pixels; the stream must contain **Width** × **Height** bytes, each representing a single threshold value. Threshold values are defined in device space in the same order as image samples in image space (see Figure 4.26 on page 301), with the first value at device coordinates (0, 0) and horizontal coordinates changing faster than vertical.

Type 10 Halftones

Although type 6 halftones can be used to specify a threshold array with a zero screen angle, they make no provision for other angles. The type 10 halftone removes this restriction and allows the use of threshold arrays for halftones with nonzero screen angles as well.

TABLE 6.4 Additional entries specific to a type 6 halftone dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Halftone** for a halftone dictionary.
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 6 for this type of halftone.
HalftoneName	string	*(Optional)* The name of the halftone dictionary.
Width	integer	*(Required)* The width of the threshold array, in device pixels.
Height	integer	*(Required)* The height of the threshold array, in device pixels.
TransferFunction	function or name	*(Optional)* A transfer function, which overrides the current transfer function in the graphics state for the same component. This entry is required if the dictionary is a component of a type 5 halftone (see "Type 5 Halftones" on page 465) and represents either a nonprimary or nonstandard primary color component (see Section 6.3, "Transfer Functions"). The name **Identity** may be used to specify the identity function.

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 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 6.2 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.

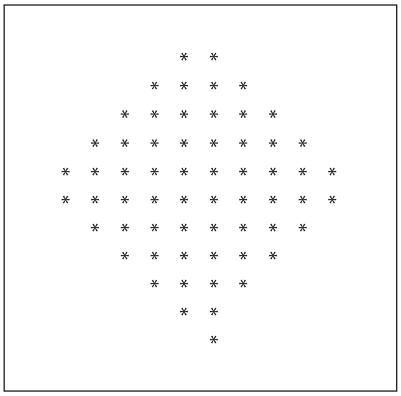


FIGURE 6.2 *Halftone cell with a nonzero angle*

Figure 6.3 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 6.4). The last row in the first square is immediately adjacent to the first row in the second and starts in the same column.

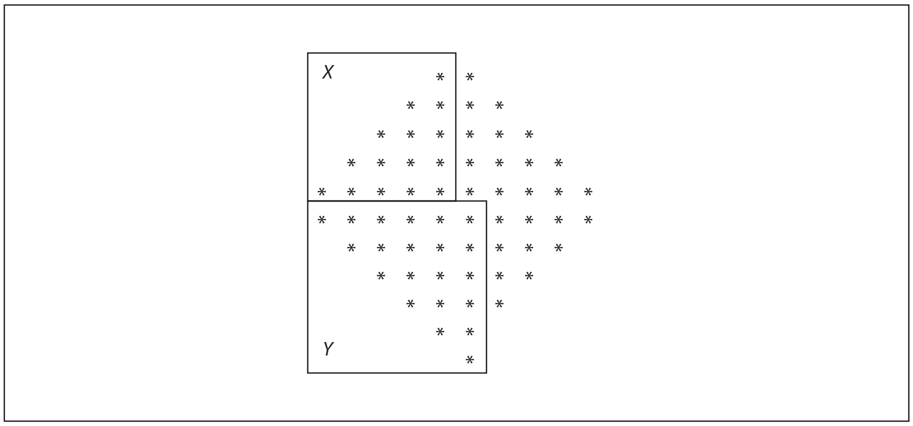


FIGURE 6.3 *Angled halftone cell divided into two squares*

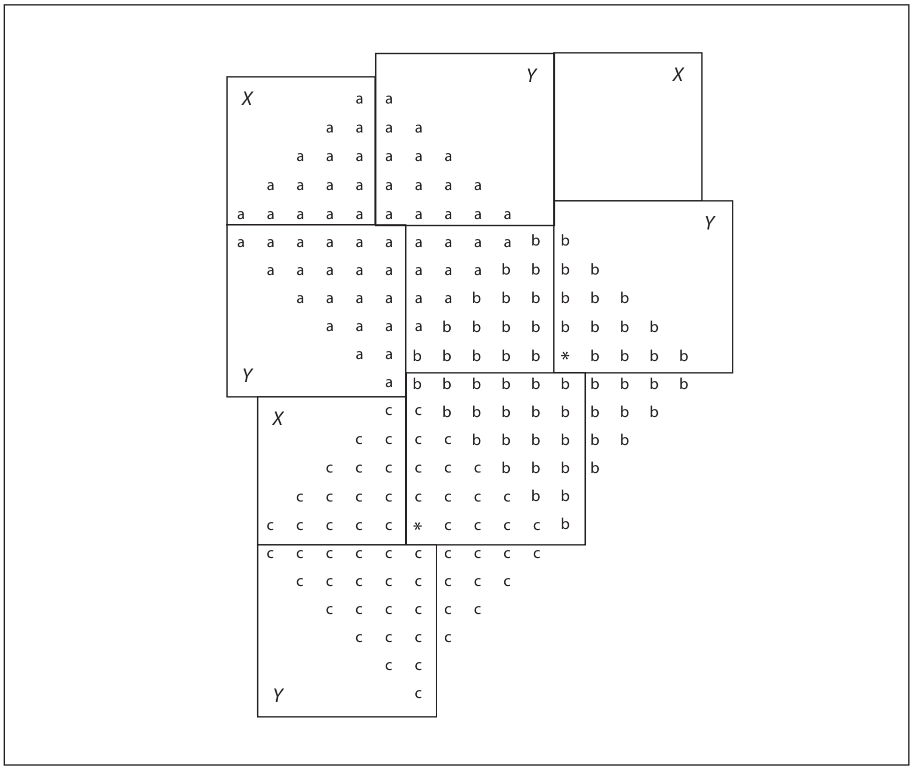


FIGURE 6.4 *Halftone cell and two squares tiled across device space*

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 6.4. 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 with sides X and Y is thus given by

$$\mathit{frequency} = \frac{\mathit{resolution}}{\sqrt{X^2 + Y^2}}$$

and the angle by

$$\mathit{angle} = \operatorname{atan}\left(\frac{Y}{X}\right)$$

Like a type 6 halftone, a type 10 halftone is represented as a stream containing the threshold values, with the parameters defining the halftone specified by entries in the stream dictionary. Table 6.5 shows the contents of this dictionary, in addition to the usual entries common to all streams (see Table 3.4 on page 38); the **Xsquare** and **Ysquare** entries replace the type 6 halftone's **Width** and **Height** entries.

TABLE 6.5 Additional entries specific to a type 10 halftone dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Halftone** for a halftone dictionary.
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 10 for this type of halftone.
HalftoneName	string	*(Optional)* The name of the halftone dictionary.
Xsquare	integer	*(Required)* The side of square *X,* in device pixels; see below.
Ysquare	integer	*(Required)* The side of square *Y,* in device pixels; see below.
TransferFunction	function or name	*(Optional)* A transfer function, which overrides the current transfer function in the graphics state for the same component. This entry is required if the dictionary is a component of a type 5 halftone (see "Type 5 Halftones" on page 465) and represents either a nonprimary or nonstandard primary color component (see Section 6.3, "Transfer Functions"). The name **Identity** may be used to specify the identity function.

The **Xsquare** and **Ysquare** entries specify the dimensions of the two squares in device pixels; the stream must contain $\textbf{Xsquare}^2 + \textbf{Ysquare}^2$ bytes, each representing a single threshold value. The contents of square *X* are specified first, followed by those of square *Y*. Threshold values within each square are defined in device space in the same order as image samples in image space (see Figure 4.26 on page 301), with the first value at device coordinates (0, 0) and horizontal coordinates changing faster than vertical.

Type 16 Halftones

Like type 10, a type 16 halftone *(PDF 1.3)* defines a halftone screen with a threshold array and 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. The threshold array can consist of either one or two rectangles. If two rectangles are specified, they will tile the device space as shown in Figure 6.5. The last row in the first rectangle is immediately adjacent to the first row in the second and starts in the same column.

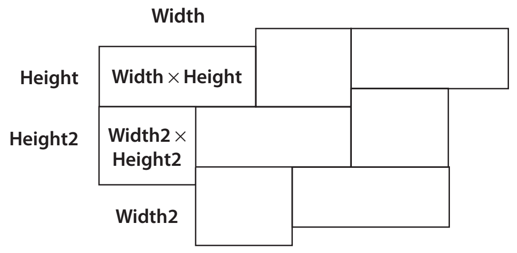


FIGURE 6.5 *Tiling of device space in a type 16 halftone*

A type 16 halftone, like type 6 and type 10, is represented as a stream containing the threshold values, with the parameters defining the halftone specified by entries in the stream dictionary. Table 6.6 shows the contents of this dictionary, in addition to the usual entries common to all streams (see Table 3.4 on page 38). The dictionary's **Width** and **Height** entries define the dimensions of the first (or only) rectangle; those of the second, optional rectangle are defined by the optional entries **Width2** and **Height2**. Each threshold value is represented as 2 bytes, with the high-order byte first. The stream must thus contain 2 × **Width** × **Height** bytes if there is only one rectangle, or 2 × (**Width** × **Height** + **Width2** × **Height2**) bytes if there are two. The contents of the first rectangle are specified first, followed by those of the second rectangle. Threshold values within each rectangle are defined in device space in the same order as image samples in image space (see Figure 4.26 on page 301), with the first value at device coordinates (0, 0) and horizontal coordinates changing faster than vertical.

TABLE 6.6 Additional entries specific to a type 16 halftone dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Halftone** for a halftone dictionary.
HalftoneType	integer	*(Required)* A code identifying the halftone type that this dictionary describes; must be 16 for this type of halftone.
HalftoneName	string	*(Optional)* The name of the halftone dictionary.
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 this entry is present, the **Height2** entry must be present as well; if this entry is absent, 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.
TransferFunction	function or name	*(Optional)* A transfer function, which overrides the current transfer function in the graphics state for the same component. This entry is required if the dictionary is a component of a type 5 halftone (see "Type 5 Halftones," below) and represents either a nonprimary or nonstandard primary color component (see Section 6.3, "Transfer Functions"). The name **Identity** may be used to specify the identity function.

Type 5 Halftones

Some devices, particularly color printers, require separate halftones for each individual colorant. Also, devices that can produce named separations may require individual halftones for each separation. Halftone dictionaries of type 5 allow individual halftones to be specified for an arbitrary number of colorants or color components.

A type 5 halftone dictionary (Table 6.7) is a composite dictionary containing independent halftone definitions for multiple colorants. Its keys are name objects representing the names of individual colorants or color components. 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 supported type except 5.

TABLE 6.7 Entries in a type 5 halftone dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Halftone** for a halftone dictionary.
HalftoneType	number	*(Required)* A code identifying the halftone type that this dictionary describes; must be 5 for this type of halftone.
HalftoneName	string	*(Optional)* The name of the halftone dictionary.
any colorant name	dictionary or stream	*(Required, one per colorant)* The halftone corresponding to the colorant or color component named by the key. The halftone may be of any type other than 5. Note that the key must be a name object; strings are not permitted, as they are in type 5 PostScript halftone dictionaries.
Default	dictionary or stream	*(Required)* A halftone to be used for any colorant or color component that does not have an entry of its own. The value may not be a type 5 halftone. If there are any nonprimary colorants, the default halftone must have a transfer function.

The colorants or color components represented in a type 5 halftone dictionary fall into two categories:

- Primary color components for the standard native device color spaces (**Red**, **Green**, and **Blue** for **DeviceRGB**; **Cyan**, **Magenta**, **Yellow**, and **Black** for **DeviceCMYK**; **Gray** for **DeviceGray**).
- Nonstandard color components for use as spot colorants in **Separation** and **DeviceN** color spaces. Some of these may also be used as process colorants if the native color space is nonstandard.

The dictionary must also contain an entry whose key is **Default**; the value of this entry is a halftone dictionary to be used for any color component that does not have an entry of its own.

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 colorant or color component named by that entry's key. This is in contrast to such a dictionary's being used as the current halftone parameter in the graphics state, which applies to all color components. If nonprimary colorants 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 colorants.

Example 6.2 shows a type 5 halftone dictionary with the primary color components for a *CMYK* device. In this example, the halftone dictionaries for the color components and for the default all use the same spot function.

Example 6.2

```
27 0 obj
   << /Type /Halftone
      /HalftoneType 5
      /Cyan 31 0 R
      /Magenta 32 0 R
      /Yellow 33 0 R
      /Black 34 0 R
      /Default 35 0 R
   >>
endobj
```

```
31 0 obj
   << /Type /Halftone
      /HalftoneType 1
      /Frequency 89.827
      /Angle 15
      /SpotFunction /Round
      /AccurateScreens true
   >>
endobj

32 0 obj
   << /Type /Halftone
      /HalftoneType 1
      /Frequency 89.827
      /Angle 75
      /SpotFunction /Round
      /AccurateScreens true
   >>
endobj

33 0 obj
   << /Type /Halftone
      /HalftoneType 1
      /Frequency 90.714
      /Angle 0
      /SpotFunction /Round
      /AccurateScreens true
   >>
endobj

34 0 obj
   << /Type /Halftone
      /HalftoneType 1
      /Frequency 89.803
      /Angle 45
      /SpotFunction /Round
      /AccurateScreens true
   >>
endobj
```

```
35 0 obj
    << /Type /Halftone
        /HalftoneType 1
        /Frequency 90.000
        /Angle 45
        /SpotFunction /Round
        /AccurateScreens true
    >>
endobj
```

6.5 Scan Conversion Details

The final step of rendering is *scan conversion.* As discussed in Section 2.1.4, "Scan Conversion," the viewer application 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 PDF. Different implementations can perform scan conversion in different ways; techniques that are appropriate for one device may be inappropriate for another. Still, it is useful to have a general understanding of how scan conversion works, particularly when creating PDF documents intended for viewing on a display. 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 Adobe Acrobat products. (These details also apply to Adobe PostScript products, yielding consistent results when a viewer application prints a document on a PostScript printer.) Most scan conversion details are not under program control, but a few are; the parameters for controlling them are described here.

6.5.1 Flatness Tolerance

The *flatness tolerance* controls the maximum permitted distance in device pixels between the mathematically correct path and an approximation constructed from straight line segments, as shown in Figure 6.6. Flatness can be specified as the operand of the **i** operator (see Table 4.7 on page 189) or as the value of the **FL** entry in a graphics state parameter dictionary (see Table 4.8 on page 190). It must be a positive number; smaller values yield greater precision at the cost of more computation.

***Note:** Although the figure exaggerates the difference between the curved and flattened paths for the sake of clarity, the purpose of the flatness tolerance is to control the precision of curve rendering, not to draw inscribed polygons. If the parameter's value is large enough to cause visible straight line segments to appear, the result is unpredictable.*

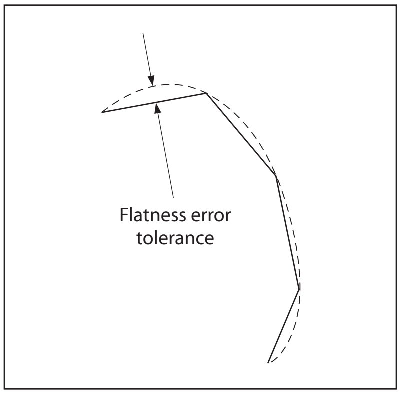


FIGURE 6.6 *Flatness tolerance*

6.5.2 Smoothness Tolerance

The *smoothness tolerance (PDF 1.3)* controls the quality of smooth shading (type 2 patterns and the **sh** operator), and thus indirectly controls the rendering performance. Smoothness is the allowable color error between 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 allowable error (or tolerance) is expressed as a fraction of the range of the color component, from 0.0 to 1.0. Thus, a smoothness tolerance of 0.1 represents a tolerance of 10 percent in each color component. Smoothness can be specified as the value of the **SM** entry in a graphics state parameter dictionary (see Table 4.8 on page 190).

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 internal smoothness of 0.5 on a high-quality color device, while setting it to 0.0 on the same device may result in an internal smoothness of 0.01 if an error of that magnitude is imperceptible on the device.

The smoothness tolerance may also interact with the accuracy of color conversion. In the case of a color conversion defined by a sampled function, the conversion function is unknown. Thus the error may be sampled at too low a frequency, in which case the accuracy defined by the smoothness tolerance cannot be guaranteed. In most cases, however, where the conversion function is smooth and continuous, the accuracy should be within the specified tolerance.

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

6.5.3 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 $i \leq x' < i + 1$ and $j \leq y' < j + 1$. Like pixels, shapes to be painted by filling and stroking operations 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 implies.

***Note:** Normally, the intersection of two regions is defined as the intersection of their interiors. However, for purposes of scan conversion, a filling region is considered to intersect every pixel through which its boundary passes, even if the interior of the filling region is empty. Thus, for example, a zero-width or zero-height rectangle will paint a line 1 pixel wide.*

The region of device space to be painted by a sampled image is determined similarly to that of a filled shape, though not identically. The viewer application 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 glyphs is performed by a different algorithm from the one above. That font rendering algorithm uses hints in the glyph descriptions and techniques that are specialized to glyph rasterization.

6.5.4 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 6.7 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, PDF 1.2 provides an optional *automatic 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.

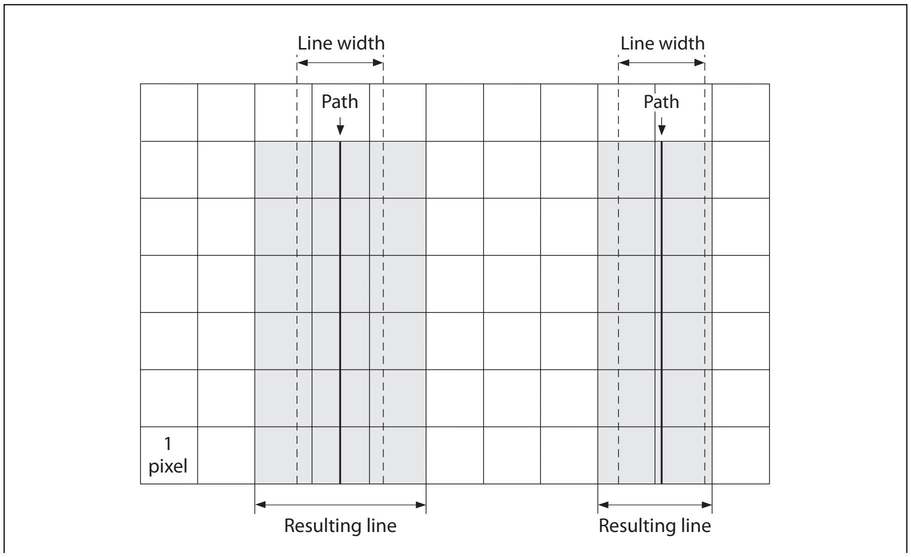


FIGURE 6.7 *Rasterization without stroke adjustment*

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 6.5.3, "Scan Conversion Rules").*

Because automatic stroke adjustment can have a substantial effect on the appearance of lines, a PDF document must be able to control whether the adjustment is to be performed. This can be specified with the stroke adjustment parameter in the graphics state, set via the **SA** entry in a graphics state parameter dictionary (see Section 4.3.4, "Graphics State Parameter Dictionaries"); see implementation note 63 in Appendix H.

CHAPTER 7

Transparency

PDF 1.4 EXTENDS the Adobe imaging model to include the notion of *transparency*. Transparent objects do not necessarily obey a strict opaque painting model, but can blend *(composite)* in interesting ways with other overlapping objects. This chapter describes the general transparency model, but does not attempt to cover how it is to be implemented. Although implementation-like descriptions are used at various points to describe how things work, this is only for the purpose of elucidating the behavior of the model; the actual implementation will almost certainly be different from what these descriptions might imply.

The chapter is organized as follows:

- Section 7.1, "Overview of Transparency," introduces the basic concepts of the transparency model and its associated terminology.
- Section 7.2, "Basic Compositing Computations," describes the mathematics involved in compositing a single object with its backdrop.
- Section 7.3, "Transparency Groups," introduces the concept of *transparency groups* and describes their properties and behavior.
- Section 7.4, "Soft Masks," covers the creation and use of masks to specify position-dependent shape and opacity.
- Section 7.5, "Specifying Transparency in PDF," describes how transparency properties are represented in a PDF document.
- Section 7.6, "Color Space and Rendering Issues," deals with some specific interactions between transparency and other aspects of color specification and rendering.

7.1 Overview of Transparency

The original Adobe imaging model paints objects (fills, strokes, text, and images), possibly clipped by a path, opaquely onto a page. The color of the page at any point is that of the topmost enclosing object, disregarding any previous objects it may overlap. This effect can be—and often is—realized simply by rendering objects directly to the page in the order in which they are specified, with each object completely overwriting any others that it overlaps.

Under the transparent imaging model, all of the objects on a page can potentially contribute to the result. Objects at a given point can be thought of as forming a *transparency stack* (or just *stack* for short), arranged from bottom to top in the order in which they are specified. The color of the page at each point is determined by combining the colors of all enclosing objects in the stack according to *compositing* rules defined by the transparency model.

Note: *The order in which objects are specified determines the stacking order, but not necessarily the order in which the objects are actually painted onto the page. In particular, the transparency model does not require a viewer application to rasterize objects immediately or to commit to a raster representation at any time before rendering the entire stack onto the page. This is important, since rasterization often causes significant loss of information and precision that is best avoided during intermediate stages of the transparency computation.*

A given object is composited with a *backdrop*. Ordinarily, the backdrop consists of the stack of all objects that have been specified previously; the result of compositing is then treated as the backdrop for the next object. However, within certain kinds of transparency group (see below), a different backdrop is chosen.

When an object is composited with its backdrop, the color at each point is computed using a specified *blend mode*, which is a function of both the object's color and the backdrop color. The blend mode determines how colors interact; different blend modes can be used to achieve a variety of useful effects. A single blend mode is in effect for compositing all of a given object, but different blend modes can be applied to different objects.

Compositing of an object with its backdrop is mediated by two scalar quantities called *shape* and *opacity*. Conceptually, for each object, these quantities are defined at every point in the plane, just as if they were additional color components.

(In actual practice, they are often obtained from auxiliary sources rather than being intrinsic to the object itself.)

Both shape and opacity vary from 0.0 (no contribution) to 1.0 (maximum contribution). At any point where either the shape or the opacity of an object is 0.0, its color is undefined. At points where the shape is 0.0, the opacity is also undefined. The shape and opacity are themselves subject to compositing rules, so that the stack as a whole also has a shape and opacity at each point.

An object's opacity, in combination with the backdrop's opacity, determines the relative contributions of the backdrop color, the object's color, and the blended color to the resulting composite color. The object's shape then determines the degree to which the composite color replaces the backdrop color. Shape values of 0.0 and 1.0 identify points that lie "outside" and "inside" a conventional sharp-edged object; intermediate values are useful in defining soft-edged objects.

Shape and opacity are conceptually very similar. In fact, they can usually be combined into a single value, called *alpha*, which controls both the color compositing computation and the fading between an object and its backdrop. However, there are a few situations in which they must be treated separately; see Section 7.3.5, "Knockout Groups." Moreover, raster-based implementations must maintain a separate shape parameter in order to do anti-aliasing properly; it is therefore convenient to have it be an explicit part of the model.

One or more consecutive objects in a stack can be collected together into a *transparency group* (often referred to hereafter simply as a *group*). The group as a whole can have various properties that modify the compositing behavior of objects within the group and their interactions with its backdrop. An additional blend mode, shape, and opacity can also be associated with the group as a whole and used when compositing it with its backdrop. Groups can be nested within other groups, forming a tree-structured hierarchy.

***Note:** The concept of a transparency group is independent of existing notions of "group" or "layer" in applications such as Adobe Illustrator®. Those groupings reflect logical relationships among objects that are meaningful when editing those objects, but they are not part of the imaging model.*

Plate 16 illustrates the effects of transparency grouping. In the upper two figures, three colored circles are painted as independent objects, with no grouping. At the upper left, the three objects are painted opaquely (opacity = 1.0); each object completely replaces its backdrop (including previously painted objects) with its own color. At the upper right, the same three independent objects are painted with an opacity of 0.5, causing them to composite with each other and with the gray and white backdrop. In the lower two figures, the three objects are combined as a transparency group. At the lower left, the individual objects have an opacity of 1.0 within the group, but the group as a whole is painted in the **Normal** blend mode with an opacity of 0.5. The objects thus completely overwrite each other within the group, but the resulting group then composites transparently with the gray and white backdrop. At the lower right, the objects have an opacity of 0.5 within the group and thus composite with each other; the group as a whole is painted against the backdrop with an opacity of 1.0, but in a different blend mode (**HardLight**), producing a different visual effect.

The color result of compositing a group can be converted to a single-component luminosity value and treated as a *soft mask*. Such a mask can then be used as an additional source of shape or opacity values for subsequent compositing operations. When the mask is used as a shape, this technique is known as *soft clipping*; it is a generalization of the current clipping path in the opaque imaging model (see Section 4.4.3, "Clipping Path Operators").

The notion of *current page* is generalized to refer to a transparency group consisting of the entire stack of objects placed on the page, composited with a backdrop that is pure white and fully opaque. Logically, this entire stack is then rasterized to determine the actual pixel values to be transmitted to the output device.

Note: *In contexts where a PDF page is treated as a piece of artwork to be placed on some other page—such as an Illustrator artboard or an Encapsulated PostScript (EPS) file—it is treated not as a page but as a group, whose backdrop may be defined differently from that of a page.*

7.2 Basic Compositing Computations

This section describes the basic computations for compositing a single object with its backdrop. These computations will be extended in Section 7.3, "Transparency Groups," to cover groups consisting of multiple objects.

7.2.1 Basic Notation for Compositing Computations

In general, variable names in this chapter consisting of a lowercase letter denote a scalar quantity, such as an opacity; uppercase letters denote a value with multiple scalar components, such as a color. In the descriptions of the basic color compositing computations, color values are generally denoted by the letter C, with a mnemonic subscript indicating which of several color values is being referred to; for instance, C_s stands for "source color." Shape and opacity values are denoted respectively by the letters f (for "*f*orm factor") and q (for "opa*q*ueness")—again with a mnemonic subscript, such as q_s for "source opacity." The symbol α (alpha) stands for a product of shape and opacity values.

In certain computations, one or more variables may have undefined values; for instance, when opacity is zero, the corresponding color is undefined. A quantity can also be undefined if it results from division by zero. In any formula that uses such an undefined quantity, the quantity has no effect on the ultimate result, because it is subsequently multiplied by zero or otherwise canceled out. The significant point is that while any arbitrary value can be chosen for such an undefined quantity, the computation must not malfunction because of exceptions caused by overflow or division by zero. It is convenient to adopt the further convention that $0 \div 0 = 0$.

7.2.2 Basic Compositing Formula

The primary change in the imaging model to accommodate transparency is in how colors are painted. In the transparent model, the result of painting (the *result color*) is a function of both the color being painted (the *source color*) and the color it is painted over (the *backdrop color*). Both of these colors may vary as a function of position on the page, but for the purposes of this section we will focus our attention on some fixed point on the page and assume a fixed backdrop and source color.

Other parameters in this computation are the *alpha*, which controls the relative contributions of the backdrop and source colors, and the *blend function*, which specifies how they are combined in the painting operation. The resulting *basic* *color compositing formula* (or just *basic compositing formula* for short) determines the result color produced by the painting operation:

$$C_r = \left(1 - \frac{\alpha_s}{\alpha_r}\right) \times C_b + \frac{\alpha_s}{\alpha_r} \times [(1 - \alpha_b) \times C_s + \alpha_b \times B(C_b, C_s)]$$

where the variables have the meanings shown in Table 7.1.

TABLE 7.1 Variables used in the basic compositing formula

VARIABLE	MEANING
C_b	Backdrop color
C_s	Source color
C_r	Result color
α_b	Backdrop alpha
α_s	Source alpha
α_r	Result alpha
$B(C_b, C_s)$	Blend function

This is actually a simplified form of the compositing formula in which the shape and opacity values are combined and represented as a single alpha value; the more general form is presented later. This function is based on the **over** operation defined in the article “Compositing Digital Images,” by Porter and Duff (see the Bibliography), extended to include a blend mode in the region of overlapping coverage. The following sections elaborate on the meaning and implications of this formula.

7.2.3 Blending Color Space

The compositing formula shown above is actually a vector function: the colors it operates on are represented in the form of n-element vectors, where n is the number of components required by the color space in which compositing is performed. The ith component of the result color C_r is obtained by applying the compositing formula to the ith components of the constituent colors C_b, C_s, and

$B(C_b, C_s)$. The result of the computation thus depends on the color space in which the colors are represented. For this reason, the color space used for compositing, called the *blending color space*, is explicitly made part of the transparent imaging model. When necessary, backdrop and source colors are converted to the blending color space prior to the compositing computation.

Of the PDF color spaces described in Section 4.5, "Color Spaces," the following are supported as blending color spaces:

- **DeviceGray**
- **DeviceRGB**
- **DeviceCMYK**
- **CalGray**
- **CalRGB**
- **ICCBased** color spaces equivalent to those above (including calibrated *CMYK*)

The **Lab** space and **ICCBased** spaces that represent lightness and chromaticity separately (such as *L*a*b**, *L*u*v**, and *HSV*) are not allowed as blending color spaces, because the compositing computations in such spaces do not give meaningful results when applied separately to each component. In addition, an **ICCBased** space used as a blending color space must be bidirectional; that is, the ICC profile must contain both *AToB* and *BToA* transformations.

The blending color space is consulted only for process colors. Although blending can also be done on individual spot colors specified in a **Separation** or **DeviceN** color space, such colors are never converted to a blending color space (except in the case where they first revert to their alternate color space, as described under "Separation Color Spaces" on page 234 and "DeviceN Color Spaces" on page 238). Instead, the specified color components are blended individually with the corresponding components of the backdrop.

The blend functions for the various blend modes assume that the range for each color component is 0.0 to 1.0 and that the color space is additive. The former condition is true for all of the allowed blending color spaces, but the latter is not. In particular, the **DeviceCMYK**, **Separation**, and **DeviceN** spaces are subtractive. When performing blending operations in subtractive color spaces, it is assumed that the color component values are complemented (subtracted from 1.0) before the blend function is applied and that the results of the function are then complemented back before being used. This adjustment makes the effects of the various blend modes numerically consistent across all color spaces. However, the actual visual effect produced by a given blend mode still depends on the color space. Blending in a device color space produces device-dependent results, whereas in a CIE-based space it produces results that are consistent across all devices. See Section 7.6, "Color Space and Rendering Issues," for additional details concerning color spaces.

7.2.4 Blend Mode

In principle, the blend function $B(C_b, C_s)$, used in the compositing formula to customize the blending operation, could be any function of the backdrop and source colors that yields another color, C_r, for the result. PDF defines a standard set of named blend functions, or *blend modes*, listed in Tables 7.2 and 7.3. Plates 18 and 19 illustrate the resulting visual effects for *RGB* and *CMYK* colors, respectively.

A blend mode is termed *separable* if each component of the result color is completely determined by the corresponding components of the constituent backdrop and source colors—that is, if the blend mode function B is applied separately to each set of corresponding components:

$$c_r = B(c_b, c_s)$$

where the lowercase variables c_r, c_b, and c_s denote corresponding components of the colors C_r, C_b, and C_s, expressed in additive form. (Theoretically, a blend mode could have a different function for each color component and still be separable; however, none of the standard PDF blend modes have this property.) A separable blend mode can be used with any color space, since it applies independently to any number of components. Only separable blend modes can be used for blending spot colors.

Table 7.2 lists the standard separable blend modes available in PDF. Some of them are defined by actual mathematical formulas; the rest are characterized only by a general description of their intended effects.

TABLE 7.2 Standard separable blend modes

NAME	RESULT
Normal	Selects the source color, ignoring the backdrop: $B(c_b, c_s) = c_s$
Multiply	Multiplies the backdrop and source color values: $B(c_b, c_s) = c_b \times c_s$ The result color is always at least as dark as either of the two constituent colors. Multiplying any color with black produces black; multiplying with white leaves the original color unchanged. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.
Screen	Multiplies the complements of the backdrop and source color values, then complements the result: $B(c_b, c_s) = 1 - [(1 - c_b) \times (1 - c_s)] = c_b + c_s - (c_b \times c_s)$ The result color is always at least as light as either of the two constituent colors. Screening any color with white produces white; screening with black leaves the original color unchanged. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.
Overlay	Multiplies or screens the colors, depending on the backdrop color. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced, but is mixed with the source color to reflect the lightness or darkness of the backdrop.
Darken	Selects the darker of the backdrop and source colors: $B(c_b, c_s) = \min(c_b, c_s)$ The backdrop is replaced with the source where the source is darker; otherwise it is left unchanged.
Lighten	Selects the lighter of the backdrop and source colors: $B(c_b, c_s) = \max(c_b, c_s)$ The backdrop is replaced with the source where the source is lighter; otherwise it is left unchanged.

NAME	**RESULT**
ColorDodge	Brightens the backdrop color to reflect the source color. Painting with black produces no change.
ColorBurn	Darkens the backdrop color to reflect the source color. Painting with white produces no change.
HardLight	Multiplies or screens the colors, depending on the source color value. If the source color is lighter than 0.5, the backdrop is lightened, as if it were screened; this is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it were multiplied; this is useful for adding shadows to a scene. The degree of lightening or darkening is proportional to the difference between the source color and 0.5; if it is equal to 0.5, the backdrop is unchanged. Painting with pure black or white produces pure black or white. The effect is similar to shining a harsh spotlight on the backdrop.
SoftLight	Darkens or lightens the colors, depending on the source color value. If the source color is lighter than 0.5, the backdrop is lightened, as if it were dodged; this is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it were burned in. The degree of lightening or darkening is proportional to the difference between the source color and 0.5; if it is equal to 0.5, the backdrop is unchanged. Painting with pure black or white produces a distinctly darker or lighter area, but does not result in pure black or white. The effect is similar to shining a diffused spotlight on the backdrop.
Difference	Subtracts the darker of the two constituent colors from the lighter: $B(c_b, c_s) = \lvert c_b - c_s \rvert$ Painting with white inverts the backdrop color; painting with black produces no change.
Exclusion	Produces an effect similar to that of the **Difference** mode, but lower in contrast. Painting with white inverts the backdrop color; painting with black produces no change.

Table 7.3 lists the standard nonseparable blend modes. Their effects are described, but no mathematical formulas are given. These modes all entail conversion to and from an intermediate *HSL* (hue-saturation-luminance) representation. Since the nonseparable blend modes consider all color components in

combination, their computation depends on the blending color space in which the components are interpreted.

TABLE 7.3 Standard nonseparable blend modes

NAME	RESULT
Hue	Creates a color with the hue of the source color and the saturation and luminance of the backdrop color.
Saturation	Creates a color with the saturation of the source color and the hue and luminance of the backdrop color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.
Color	Creates a color with the hue and saturation of the source color and the luminance of the backdrop color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images or tinting color images.
Luminosity	Creates a color with the luminance of the source color and the hue and saturation of the backdrop color. This produces an inverse effect to that of the **Color** mode.

***Note:** An additional standard blend mode, **Compatible**, is a vestige of an earlier design and is no longer needed, but is still recognized for the sake of compatibility; its effect is equivalent to that of the **Normal** blend mode. See "Compatibility with Opaque Overprinting" on page 526 for further discussion.*

7.2.5 Interpretation of Alpha

The color compositing formula

$$C_r = \left(1 - \frac{\alpha_s}{\alpha_r}\right) \times C_b + \frac{\alpha_s}{\alpha_r} \times [(1 - \alpha_b) \times C_s + \alpha_b \times B(C_b, C_s)]$$

produces a result color that is a weighted average of the backdrop color, the source color, and the blended $B(C_b, C_s)$ term, with the weighting determined by the backdrop and source alphas α_b and α_s. For the simplest blend mode, **Normal**, defined by

$$B(c_b, c_s) = c_s$$

the compositing formula collapses to a simple weighted average of the backdrop and source colors, controlled by the backdrop and source alpha values. For more interesting blend functions, the backdrop and source alphas control whether the effect of the blend mode is fully realized or is toned down by mixing the result with the backdrop and source colors.

The result alpha, α_r, is actually a computed result, described below in Section 7.2.6, "Shape and Opacity Computations." The result color is normalized by the result alpha, ensuring that when this color and alpha are subsequently used together in another compositing operation, the color's contribution will be correctly represented. Note that if α_r is zero, the result color is undefined.

The formula shown above is a simplification of the following one, which presents the relative contributions of backdrop, source, and blended colors in a more straightforward way:

$$\alpha_r \times C_r = [(1 - \alpha_s) \times \alpha_b \times C_b] + [(1 - \alpha_b) \times \alpha_s \times C_s] + [\alpha_b \times \alpha_s \times B(C_b, C_s)]$$

(The simplification requires a substitution based on the alpha compositing formula, which is presented in the next section.) Thus, mathematically, the backdrop and source alphas control the influence of the backdrop and source colors, respectively, while their product controls the influence of the blend function. An alpha value of $\alpha_s = 0.0$ or $\alpha_b = 0.0$ results in no blend mode effect; setting $\alpha_s = 1.0$ and $\alpha_b = 1.0$ results in maximum blend mode effect.

7.2.6 Shape and Opacity Computations

As stated earlier, the alpha values that control the compositing process are defined as the product of shape and opacity:

$$\begin{aligned} \alpha_b &= f_b \times q_b \\ \alpha_r &= f_r \times q_r \\ \alpha_s &= f_s \times q_s \end{aligned}$$

This section examines the various shape and opacity values individually. Once again, keep in mind that conceptually these values are computed for every point on the page.

Source Shape and Opacity

Shape and opacity values can come from several sources. The transparency model provides for three independent sources for each; however, the PDF representation imposes some limitations on the ability to specify all of these sources independently (see Section 7.5.3, "Specifying Shape and Opacity").

- *Object shape*. Elementary objects such as strokes, fills, and text have an intrinsic shape, whose value is 1.0 for points inside the object and 0.0 outside. Similarly, an image with an explicit mask (see "Explicit Masking" on page 314) has a shape that is 1.0 in the unmasked portions and 0.0 in the masked portions. The shape of a group object is the union of the shapes of the objects it contains.

  ***Note:** Mathematically, elementary objects have "hard" edges, with a shape value of either 0.0 or 1.0 at every point. However, when such objects are rasterized to device pixels, the shape values along the boundaries may be* anti-aliased, *taking on fractional values representing fractional coverage of those pixels. When such anti-aliasing is performed, it is important to treat the fractional coverage as shape rather than opacity.*

- *Mask shape*. Shape values for compositing an object can be taken from an additional source, or *soft mask*, independent of the object itself. (See Section 7.4, "Soft Masks," for a discussion of how such a mask might be generated.) The use of a soft mask to modify the shape of an object or group, called *soft clipping*, can produce effects such as a gradual transition between an object and its backdrop, as in a vignette.

- *Constant shape*. The source shape can be modified at every point by a scalar *shape constant*. This is merely a convenience, since the same effect could be achieved with a shape mask whose value is the same everywhere.

- *Object opacity*. Elementary objects have an opacity of 1.0 everywhere. The opacity of a group object is the result of the opacity computations for all of the objects it contains.

- *Mask opacity*. Opacity values, like shape values, can be provided by a soft mask independent of the object being composited.

- *Constant opacity*. The source opacity can be modified at every point by a scalar *opacity constant*. It is useful to think of this value as the "current opacity," analogous to the current color used when painting elementary objects.

All of these shape and opacity inputs range in value from 0.0 to 1.0, with a default value of 1.0. The intent is that any of the inputs will make the painting operation more transparent as it goes toward 0.0. If more than one input goes toward 0.0, the effect is compounded. This is achieved mathematically by simply multiplying the three inputs of each type, producing intermediate values called the *source shape* and the *source opacity*:

$$\begin{aligned} f_s &= f_j \times f_m \times f_k \\ q_s &= q_j \times q_m \times q_k \end{aligned}$$

where the variables have the meanings shown in Table 7.4.

TABLE 7.4 Variables used in the source shape and opacity formulas

VARIABLE	MEANING
f_s	Source shape
f_j	Object shape
f_m	Mask shape
f_k	Constant shape
q_s	Source opacity
q_j	Object opacity
q_m	Mask opacity
q_k	Constant opacity

***Note:** When an object is painted with a tiling pattern, the object shape and object opacity for points in the object's interior are determined by those of corresponding points in the pattern, rather than being 1.0 everywhere (see Section 7.5.6, "Patterns and Transparency").*

Result Shape and Opacity

In addition to a result color, the painting operation also computes an associated *result shape* and *result opacity*. These computations are based on the *union function*

$$\begin{aligned}\text{Union}(b, s) &= 1 - [(1 - b) \times (1 - s)] \\ &= b + s - (b \times s)\end{aligned}$$

where b and s are the backdrop and source values to be composited. This is a generalization of the conventional concept of union for opaque shapes, and can be thought of as an "inverted multiplication"—a multiplication with the inputs and outputs complemented. The result tends toward 1.0: if either input is 1.0, the result will be 1.0.

The result shape and opacity are given by

$$\begin{aligned}f_r &= \text{Union}(f_b, f_s) \\ q_r &= \frac{\text{Union}(f_b \times q_b, f_s \times q_s)}{f_r}\end{aligned}$$

where the variables have the meanings shown in Table 7.5.

TABLE 7.5 Variables used in the result shape and opacity formulas

VARIABLE	MEANING
f_r	Result shape
f_b	Backdrop shape
f_s	Source shape
q_r	Result opacity
q_b	Backdrop opacity
q_s	Source opacity

These formulas can be interpreted as follows:

- The result shape is simply the union of the backdrop and source shapes.
- The result opacity is the union of the backdrop and source opacities, weighted by their respective shapes. The result is then normalized by the result shape, ensuring that when this shape and opacity are subsequently used together in another compositing operation, the opacity's contribution will be correctly represented.

Since alpha is just the product of shape and opacity, it can easily be shown that

$$\alpha_r = \text{Union}(\alpha_b, \alpha_s)$$

This formula can be used whenever the independent shape and opacity results are not needed.

7.2.7 Summary of Basic Compositing Computations

Below is a summary of all the computations presented in this section. They are given in an order such that no variable is used before it is computed; also, some of the formulas have been rearranged to simplify them. See Tables 7.1, 7.4, and 7.5 above for the meanings of the variables used in these formulas.

$$\begin{aligned}\text{Union}(b, s) &= 1 - [(1-b)\times(1-s)] \\ &= b + s - (b\times s)\end{aligned}$$

$$\begin{aligned}f_s &= f_j \times f_m \times f_k \\ q_s &= q_j \times q_m \times q_k \\ f_r &= \text{Union}(f_b, f_s)\end{aligned}$$

$$\begin{aligned}\alpha_b &= f_b \times q_b \\ \alpha_s &= f_s \times q_s \\ \alpha_r &= \text{Union}(\alpha_b, \alpha_s)\end{aligned}$$

$$q_r = \frac{\alpha_r}{f_r}$$

$$C_r = \left(1 - \frac{\alpha_s}{\alpha_r}\right) \times C_b + \frac{\alpha_s}{\alpha_r} \times [(1-\alpha_b)\times C_s + \alpha_b \times B(C_b, C_s)]$$

7.3 Transparency Groups

A *transparency group* is a sequence of consecutive objects in a transparency stack that are collected together and composited to produce a single color, shape, and opacity at each point. The result is then treated as if it were a single object for subsequent compositing operations. This facilitates creating independent pieces of artwork, each composed of multiple objects, and then combining them, possibly with additional transparency effects applied during the combination. Groups can be nested within other groups to form a tree-structured group hierarchy.

The objects contained within a group are treated as a separate transparency stack, called the *group stack*. The objects in the stack are composited against some initial backdrop (discussed later), producing a composite color, shape, and opacity for the group as a whole. The result is an object whose shape is the union of the shapes of its constituent objects and whose color and opacity are the result of the compositing operations. This object is then composited with the group's backdrop in the usual way.

In addition to its computed color, shape, and opacity, the group as a whole can have several further attributes:

- All of the input variables that affect the compositing computation for individual objects can also be applied when compositing the group with its backdrop. These include mask and constant shape, mask and constant opacity, and blend mode.
- The group can be *isolated* or *non-isolated*, determining the initial backdrop against which its stack is composited.
- The group can be *knockout* or *non-knockout*, determining whether the objects within its stack are composited with one another or only with the group's backdrop.
- An isolated group can specify its own blending color space, independent of that of the group's backdrop.
- Instead of being composited onto the current page, a group's results can be used as a source of shape or opacity values for creating a *soft mask* (see Section 7.4, "Soft Masks").

The next section introduces some new notation for dealing with group compositing. Subsequent sections describe the group compositing formulas for a non-isolated, non-knockout group and the special properties of isolated and knockout groups.

7.3.1 Notation for Group Compositing Computations

Since we are now dealing with multiple objects at a time, it is useful to have some notation for distinguishing among them. Accordingly, the variables introduced earlier are altered to include a second-level subscript denoting an object's position in the transparency stack. Thus, for example, C_{s_i} stands for "source color of the *i*th object in the stack." The subscript 0 represents the initial backdrop; subscripts 1 to n denote the bottommost to topmost objects in an n-element stack. In addition, the subscripts b and r are dropped from the variables $C_b, f_b, q_b, \alpha_b, C_r, f_r, q_r$, and α_r; other variables retain their mnemonic subscripts.

These conventions permit the compositing formulas to be restated as recurrence relations among the elements of a stack. For instance, the result of the color compositing computation for object i is denoted by C_i (formerly C_r). This computation takes as one of its inputs the immediate backdrop color, which is the result of the color compositing computation for object $i-1$; this is denoted by C_{i-1} (formerly C_b).

The revised formulas for a simple n-element stack (not including any groups) are, for $i = 1, \ldots, n$:

$$\begin{aligned} f_{s_i} &= f_{j_i} \times f_{m_i} \times f_{k_i} \\ q_{s_i} &= q_{j_i} \times q_{m_i} \times q_{k_i} \end{aligned}$$

$$\begin{aligned} \alpha_{s_i} &= f_{s_i} \times q_{s_i} \\ \alpha_i &= \text{Union}(\alpha_{i-1}, \alpha_{s_i}) \end{aligned}$$

$$\begin{aligned} f_i &= \text{Union}(f_{i-1}, f_{s_i}) \\ q_i &= \frac{\alpha_i}{f_i} \end{aligned}$$

$$C_i = \left(1 - \frac{\alpha_{s_i}}{\alpha_i}\right) \times C_{i-1} + \frac{\alpha_{s_i}}{\alpha_i} \times [(1 - \alpha_{i-1}) \times C_{s_i} + \alpha_{i-1} \times B_i(C_{i-1}, C_{s_i})]$$

where the variables have the meanings shown in Table 7.6. Compare these formulas with those shown in Section 7.2.7, “Summary of Basic Compositing Computations.”

TABLE 7.6 Revised variables for the basic compositing formulas

VARIABLE	MEANING
f_{s_i}	Source shape for object *i*
f_{j_i}	Object shape for object *i*
f_{m_i}	Mask shape for object *i*
f_{k_i}	Constant shape for object *i*
f_i	Result shape after compositing object *i*
q_{s_i}	Source opacity for object *i*
q_{j_i}	Object opacity for object *i*
q_{m_i}	Mask opacity for object *i*
q_{k_i}	Constant opacity for object *i*
q_i	Result opacity after compositing object *i*
α_{s_i}	Source alpha for object *i*
α_i	Result alpha after compositing object *i*
C_{s_i}	Source color for object *i*
C_i	Result color after compositing object *i*
$B_i(C_{i-1}, C_{s_i})$	Blend function for object *i*

7.3.2 Group Structure and Nomenclature

As stated earlier, the elements of a group are treated as a separate transparency stack, the group stack. These objects are composited against a selected initial backdrop (to be described) and the resulting color, shape, and opacity are then treated as if they belonged to a single object. The resulting object is in turn composited with the group's backdrop in the usual way.

This computation entails interpreting the stack as a tree. For an n-element group that begins at position i in the stack, it treats the next n objects as an n-element substack, whose elements are given an independent numbering of 1 to n. These objects are then removed from the object numbering in the parent (containing) stack and replaced by the group object, numbered i, followed by the remaining objects to be painted on top of the group, renumbered starting at $i + 1$. This operation applies recursively to any nested subgroups. Henceforth, the term *element* (denoted E_i) refers to a member of some group; it can itself be either an individual object or a contained subgroup.

From the perspective of a particular element in a nested group, there are three different backdrops of interest:

- *The group backdrop* is the result of compositing all elements up to but not including the first element in the group. (This definition is altered if the parent group is a knockout group; see Section 7.3.5, "Knockout Groups.")
- *The initial backdrop* is a backdrop that is selected for compositing the group's first element. This is either the same as the group backdrop (for a non-isolated group) or a fully transparent backdrop (for an isolated group).
- *The immediate backdrop* is the result of compositing all elements in the group up to but not including the current element.

When all elements in a group have been composited, the result is treated as if the group were a single object, which is then composited with the group backdrop. (Note that this operation occurs whether the initial backdrop chosen for compositing the elements of the group was the group backdrop or a transparent backdrop. There is a special correction to ensure that the backdrop's contribution to the overall result is applied only once.)

7.3.3 Group Compositing Computations

The color and opacity of a group are defined by the *group compositing function*:

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

where the variables have the meanings shown in Table 7.7.

TABLE 7.7 Arguments and results of the group compositing function

VARIABLE	MEANING
G	The transparency group: a compound object consisting of all elements $E_1, \ldots, E_n$ of the group—the n constituent objects' colors, shapes, opacities, and blend modes
C_0	Color of the group's backdrop
C	Computed color of the group, to be used as the source color when the group itself is treated as an object
f	Computed shape of the group, to be used as the object shape when the group itself is treated as an object
α_0	Alpha of the group's backdrop
α	Computed alpha of the group, to be used as the object alpha when the group itself is treated as an object

Note that the opacity is not given explicitly as an argument or result of this function. Almost all of the computations use the product of shape and opacity (alpha) rather than opacity by itself, so it is usually convenient to work directly with shape and alpha, rather than shape and opacity. When needed, the opacity can be computed by dividing the alpha by the associated shape.

The result of applying the group compositing function is then treated as if it were a single object, which in turn is composited with the group's backdrop according to the usual formulas. In those formulas, the color, shape, and alpha *(C, f, and α)* calculated by the group compositing function are used, respectively, as the source color C_s, the object shape f_j, and the object alpha α_j.

The group compositing formulas for a non-isolated, non-knockout group are defined as follows:

- Initialization:

$$f_{g_0} = \alpha_{g_0} = 0.0$$

- For each group element $E_i \in G$ $(i = 1, \dots, n)$:

$$\langle C_{s_i}, f_{j_i}, \alpha_{j_i} \rangle = \begin{cases} \text{Composite}(C_{i-1}, \alpha_{i-1}, E_i) & \text{if } E_i \text{ is a group} \\ \text{intrinsic color, shape, and (shape} \times \text{opacity) of } E_i & \text{otherwise} \end{cases}$$

$$\begin{aligned} f_{s_i} &= f_{j_i} \times f_{m_i} \times f_{k_i} \\ \alpha_{s_i} &= \alpha_{j_i} \times (f_{m_i} \times q_{m_i}) \times (f_{k_i} \times q_{k_i}) \\ f_{g_i} &= \text{Union}(f_{g_{i-1}}, f_{s_i}) \\ \alpha_{g_i} &= \text{Union}(\alpha_{g_{i-1}}, \alpha_{s_i}) \\ \alpha_i &= \text{Union}(\alpha_0, \alpha_{g_i}) \end{aligned}$$

$$C_i = \left(1 - \frac{\alpha_{s_i}}{\alpha_i}\right) \times C_{i-1} + \frac{\alpha_{s_i}}{\alpha_i} \times ((1 - \alpha_{i-1}) \times C_{s_i} + \alpha_{i-1} \times B_i(C_{i-1}, C_{s_i}))$$

- Result:

$$\begin{aligned} C &= C_n + (C_n - C_0) \times \left(\frac{\alpha_0}{\alpha_{g_n}} - \alpha_0\right) \\ f &= f_{g_n} \\ \alpha &= \alpha_{g_n} \end{aligned}$$

where the variables have the meanings shown in Table 7.8 (in addition to those in Table 7.7 above).

For an element E_i that is an elementary object, the color, shape, and alpha values C_{s_i}, f_{j_i}, and α_{j_i} are intrinsic attributes of the object itself. For an element that is itself a group, the group compositing function is applied recursively to the subgroup and the resulting C, f, and α values are used for its C_{s_i}, f_{j_i}, and α_{j_i} in the calculations for the parent group.

TABLE 7.8 Variables used in the group compositing formulas

VARIABLE	MEANING
E_i	Element i of the group: a compound variable representing the element's color, shape, opacity, and blend mode
f_{s_i}	Source shape for element E_i
f_{j_i}	Object shape for element E_i
f_{m_i}	Mask shape for element E_i
f_{k_i}	Constant shape for element E_i
f_{g_i}	Group shape: the accumulated source shapes of group elements E_1 to E_i, excluding the initial backdrop
q_{m_i}	Mask opacity for element E_i
q_{k_i}	Constant opacity for element E_i
α_{s_i}	Source alpha for element E_i
α_{j_i}	Object alpha for element E_i: the product of its object shape and object opacity
α_{g_i}	Group alpha: the accumulated source alphas of group elements E_1 to E_i, excluding the initial backdrop
α_i	Accumulated alpha after compositing element E_i, including the initial backdrop
C_{s_i}	Source color for element E_i
C_i	Accumulated color after compositing element E_i, including the initial backdrop
$B_i(C_{i-1}, C_{s_i})$	Blend function for element E_i

Note that the elements of a group are composited onto a backdrop that includes the group's initial backdrop. This is done to achieve the correct effects of the blend modes, most of which are dependent on both the backdrop and source colors being blended. (This feature is what distinguishes non-isolated groups from isolated groups, discussed in the next section.)

Special attention should be directed to the formulas at the end that compute the final results, C, f, and α, of the group compositing function. Essentially, these formulas remove the contribution of the group backdrop from the computed results. This ensures that when the group itself is subsequently composited with that backdrop (possibly with additional shape or opacity inputs or a different blend mode), the backdrop's contribution is included only once.

For color, the backdrop removal is accomplished by an explicit calculation, whose effect is essentially the reverse of compositing with the **Normal** blend mode. The formula is a simplification of the following formulas, which present this operation more intuitively:

$$\phi_b = \frac{(1 - \alpha_{g_n}) \times \alpha_0}{\text{Union}(\alpha_0, \alpha_{g_n})}$$

$$C = \frac{C_n - \phi_b \times C_0}{1 - \phi_b}$$

where ϕ_b is the *backdrop fraction*, the relative contribution of the backdrop color to the overall color.

For shape and alpha, backdrop removal is accomplished by maintaining two sets of variables to hold the accumulated values. The group shape and alpha, f_{g_i} and α_{g_i}, accumulate only the shape and alpha of the group elements, excluding the group backdrop; their final values become the group results returned by the group compositing function. The complete alpha, α_i, includes the backdrop contribution as well; its value is used in the color compositing computations. (There is never any need to compute the corresponding complete shape, f_i, that includes the backdrop contribution.)

As a result of these corrections, the effect of compositing objects as a group is the same as that of compositing them separately (without grouping) if the following conditions hold:

- The group is non-isolated and has the same knockout attribute as its parent group (see Sections 7.3.4, "Isolated Groups," and 7.3.5, "Knockout Groups").
- When compositing the group's results with the group backdrop, the **Normal** blend mode is used and the shape and opacity inputs are always 1.0.

7.3.4 Isolated Groups

An *isolated group* is one whose elements are composited onto a fully transparent initial backdrop rather than onto the group's backdrop. The resulting source color, object shape, and object alpha for the group are therefore independent of the group backdrop. The only interaction with the group backdrop occurs when the group's computed color, shape, and alpha are then composited with it.

In particular, the special effects produced by the blend modes of objects within the group take into account only the intrinsic colors and opacities of those objects; they are not influenced by the group's backdrop. For example, applying the **Multiply** blend mode to an object in the group will produce a darkening effect on other objects lower in the group's stack, but not on the group's backdrop.

Plate 17 illustrates this effect for a group consisting of four overlapping circles in a light gray color ($C = M = Y = 0.0$; $K = 0.15$). The circles are painted within the group with opacity 1.0 in the **Multiply** blend mode; the group itself is painted against its backdrop in **Normal** blend mode. In the top row, the group is isolated and thus does not interact with the rainbow backdrop; in the bottom row, it is non-isolated and composites with the backdrop. The plate also illustrates the difference between knockout and non-knockout groups (see Section 7.3.5, "Knockout Groups").

The effect of an isolated group can be represented by a simple object that directly specifies a color, shape, and opacity at each point. This so-called "flattening" of an isolated group is sometimes useful for importing and exporting fully composited artwork in applications. Furthermore, a group that specifies an explicit blending color space must be an isolated group.

For an isolated group, the group compositing formulas are altered by simply adding one statement to the initialization:

$$\alpha_0 = 0.0 \qquad \text{if the group is isolated}$$

That is, the initial backdrop on which the elements of the group are composited is transparent, rather than inherited from the group's backdrop. This substitution also makes C_0 undefined, but the normal compositing formulas take care of that. Also, the result computation for C automatically simplifies to $C = C_n$, since there is no backdrop contribution to be factored out.

7.3.5 Knockout Groups

In a knockout group, each individual element is composited with the group's initial backdrop, rather than with the stack of preceding elements in the group. When objects have binary shapes (1.0 for "inside," 0.0 for "outside"), each object overwrites ("knocks out") the effects of any earlier elements it overlaps within the same group. At any given point, only the topmost object enclosing the point contributes to the result color and opacity of the group as a whole.

Plate 17, already discussed above in Section 7.3.4, "Isolated Groups," illustrates the difference between knockout and non-knockout groups. In the left column, the four overlapping circles are defined as a knockout group and therefore do not composite with each other within the group; in the right column, they form a non-knockout group and thus do composite with each other. In each column, the upper and lower figures depict an isolated and a non-isolated group, respectively.

This model is similar to the opaque imaging model, except that the "topmost object wins" rule applies to both the color and the opacity. Knockout groups are useful in composing a piece of artwork from a collection of overlapping objects, where the topmost object in any overlap completely obscures those beneath. At the same time, the topmost object interacts with the group's initial backdrop in the usual way, with its opacity and blend mode applied as appropriate.

The concept of "knockout" is generalized to accommodate fractional shape values. In that case, the immediate backdrop is only partially knocked out and replaced by only a fraction of the result of compositing the object with the initial backdrop.

The restated group compositing formulas deal with knockout groups by introducing a new variable, *b*, which is a subscript that specifies which previous result to use as the backdrop in the compositing computations: 0 in a knockout group or $i - 1$ in a non-knockout group. When $b = i - 1$, the formulas simplify to the ones given in Section 7.3.3, "Group Compositing Computations."

In the general case, the computation proceeds in two stages:

1. Composite the object with the group's initial backdrop, but disregarding the object's shape and using a source shape value of 1.0 everywhere. This produces unnormalized temporary alpha and color results, α_t and C_t. (For color, this computation is essentially the same as the unsimplified color compositing formula given in Section 7.2.5, "Interpretation of Alpha," but using a source shape of 1.0.)

$$\alpha_t = \text{Union}(\alpha_{g_b}, q_{s_i})$$

$$C_t = (1 - q_{s_i}) \times \alpha_b \times C_b + q_{s_i} \times ((1 - \alpha_b) \times C_{s_i} + \alpha_b \times B_i(C_b, C_{s_i}))$$

2. Compute a weighted average of this result with the object's immediate backdrop, using the source shape as the weighting factor. Then normalize the result color by the result alpha:

$$\alpha_{g_i} = (1 - f_{s_i}) \times \alpha_{g_{i-1}} + f_{s_i} \times \alpha_t$$

$$\alpha_i = \text{Union}(\alpha_0, \alpha_{g_i})$$

$$C_i = \frac{(1 - f_{s_i}) \times \alpha_{i-1} \times C_{i-1} + f_{s_i} \times C_t}{\alpha_i}$$

This averaging computation is performed for both color and alpha. The formulas above show this averaging directly; those in Section 7.3.7, "Summary of Group Compositing Computations," are slightly altered to use source shape and alpha rather than source shape and opacity, avoiding the need to compute a source opacity value explicitly. (Note that C_t there is slightly different from C_t above: it is premultiplied by f_{s_i}.)

The extreme values of the source shape produce the straightforward knockout effect. That is, a shape value of 1.0 ("inside") yields the color and opacity that result from compositing the object with the initial backdrop. A shape value of 0.0 ("outside") leaves the previous group results unchanged. The existence of the knockout feature is the main reason for maintaining a separate shape value, rather than only a single alpha that combines shape and opacity. The separate shape value must be computed in any group that is subsequently used as an element of a knockout group.

A knockout group can be isolated or non-isolated; that is, *isolated* and *knockout* are independent attributes. A non-isolated knockout group composites its topmost enclosing element with the group's backdrop; an isolated knockout group composites the element with a transparent backdrop.

Note: *When a non-isolated group is nested within a knockout group, the initial backdrop of the inner group is the same as that of the outer group; it is not the immediate backdrop of the inner group. This behavior, although perhaps unexpected, is a consequence of the group compositing formulas when* $b = 0$.

7.3.6 Page Group

All of the elements painted directly onto a page—both top-level groups and top-level objects that are not part of any group—are treated as if they were contained in a transparency group *P*, which in turn is composited with a context-dependent backdrop. This group is called the *page group.*

The page group can be treated in two distinctly different ways:

- Ordinarily, the page is imposed directly on an output medium, such as paper or a display screen. The page group is treated as an isolated group, whose results are then composited with a backdrop color appropriate for the medium. The backdrop is nominally white, although varying according to the actual properties of the medium. However, some applications may choose to provide a different backdrop, such as a checkerboard or grid to aid in visualizing the effects of transparency in the artwork.
- A "page" of a PDF file can be treated as a graphics object to be used as an element of a page of some other document. This case arises, for example, when placing a PDF file containing a piece of artwork produced by Illustrator into a page layout produced by InDesign®. In this situation, the PDF "page" is not composited with the media color; instead, it is treated as an ordinary transparency group, which can be either isolated or non-isolated and is composited with its backdrop in the normal way.

The remainder of this section pertains only to the first use of the page group, where it is to be imposed directly on the medium.

The color C of the page at a given point is defined by a simplification of the general group compositing formula:

$$\begin{aligned}\langle C_g, f_g, \alpha_g \rangle &= \text{Composite}(\dot{U}, 0, P) \\ C &= (1 - \alpha_g) \times W + \alpha_g \times C_g\end{aligned}$$

where the variables have the meanings shown in Table 7.9. The first formula computes the color and alpha for the group given a transparent backdrop—in effect, treating P as an isolated group. The second formula composites the results with the context-dependent backdrop (using the equivalent of the **Normal** blend mode).

TABLE 7.9 Variables used in the page group compositing formulas

VARIABLE	MEANING
P	The page group, consisting of all elements $E_1, \ldots, E_n$ in the page's top-level stack
C_g	Computed color of the page group
f_g	Computed shape of the page group
α_g	Computed alpha of the page group
C	Computed color of the page
W	Initial color of the page (nominally white, but may vary depending on the properties of the medium or the needs of the application)
U	An undefined color (which is not used, since the α_0 argument of Composite is 0)

If not otherwise specified, the page group's color space is inherited from the native color space of the output device—that is, a device color space, such as **DeviceRGB** or **DeviceCMYK**. It is often preferable to specify an explicit color space, particularly a CIE-based space, to ensure more predictable results of the compositing computations within the page group. In this case, all page-level compositing is done in the specified color space, with the entire result then converted to the native color space of the output device before being composited with the context-dependent backdrop. This case also arises when the page is not actually being rendered but is converted to a "flattened" representation in an opaque imaging model, such as PostScript.

7.3.7 Summary of Group Compositing Computations

The following restatement of the group compositing formulas also takes isolated groups and knockout groups into account. See Tables 7.7 and 7.8 on pages 493 and 495 for the meanings of the variables.

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

- Initialization:

$$f_{g_0} = \alpha_{g_0} = 0$$

$$\alpha_0 = 0 \qquad \text{if the group is isolated}$$

- For each group element $E_i \in G$ $(i = 1, \ldots, n)$:

$$b = \begin{cases} 0 & \text{if the group is knockout} \\ i-1 & \text{otherwise} \end{cases}$$

$$\langle C_{s_i}, f_{j_i}, \alpha_{j_i} \rangle = \begin{cases} \text{Composite}(C_b, \alpha_b, E_i) & \text{if } E_i \text{ is a group} \\ \text{intrinsic color, shape, and (shape} \times \text{opacity) of } E_i & \text{otherwise} \end{cases}$$

$$f_{s_i} = f_{j_i} \times f_{m_i} \times f_{k_i}$$
$$\alpha_{s_i} = \alpha_{j_i} \times (f_{m_i} \times q_{m_i}) \times (f_{k_i} \times q_{k_i})$$

$$f_{g_i} = \text{Union}(f_{g_{i-1}}, f_{s_i})$$
$$\alpha_{g_i} = (1 - f_{s_i}) \times \alpha_{g_{i-1}} + (f_{s_i} - \alpha_{s_i}) \times \alpha_{g_b} + \alpha_{s_i}$$
$$\alpha_i = \text{Union}(\alpha_0, \alpha_{g_i})$$

$$C_t = (f_{s_i} - \alpha_{s_i}) \times \alpha_b \times C_b + \alpha_{s_i} \times ((1 - \alpha_b) \times C_{s_i} + \alpha_b \times B_i(C_b, C_{s_i}))$$

$$C_i = \frac{(1 - f_{s_i}) \times \alpha_{i-1} \times C_{i-1} + C_t}{\alpha_i}$$

- Result:

$$C = C_n + (C_n - C_0) \times \left(\frac{\alpha_0}{\alpha_{g_n}} - \alpha_0 \right)$$
$$f = f_{g_n}$$
$$\alpha = \alpha_{g_n}$$

Note: *Once again, keep in mind that these formulas are in their most general form. They can be significantly simplified when some sources of shape and opacity are not present or when shape and opacity need not be maintained separately. Furthermore, in each specific type of group (isolated or not, knockout or not), some terms of these formulas cancel or drop out. An efficient implementation should use the simplified derived formulas.*

7.4 Soft Masks

As stated in earlier sections, the shape and opacity values used in compositing an object can include components called the mask shape (f_m) and mask opacity (q_m), which originate from a source independent of the object itself. Such an independent source, called a *soft mask*, defines values that can vary across different points on the page. The word *soft* emphasizes that the mask value at a given point is not limited to just 0.0 or 1.0, but can take on intermediate fractional values as well. Such a mask is typically the only means of providing position-dependent opacity values, since elementary objects do not have intrinsic opacity of their own.

A mask used as a source of shape values is also called a *soft clip*, by analogy with the “hard” clipping path of the opaque imaging model (see Section 4.4.3, “Clipping Path Operators”). The soft clip is a generalization of the hard clip: a hard clip can be represented as a soft clip having shape values of 1.0 inside and 0.0 outside the clipping path. Everywhere inside a hard clipping path, the source object’s color replaces the backdrop; everywhere outside, the backdrop shows through unchanged. With a soft clip, by contrast, a gradual transition can be created between an object and its backdrop, as in a vignette.

A mask can be defined by creating a transparency group and painting objects into it, thereby defining color, shape, and opacity in the usual way. The resulting group can then be used to derive the mask in either of two ways, as described in the following sections.

7.4.1 Deriving a Soft Mask from Group Alpha

In the first method of defining a soft mask, the color, shape, and opacity of a transparency group G are first computed by the usual formula

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

where C_0 and α_0 represent an arbitrary backdrop whose value does not contribute to the eventual result. The C, f, and α results are the group's color, shape, and alpha, respectively, with the backdrop factored out.

The mask value at each point is then derived from the alpha of the group. Since the group's color is not used in this case, there is no need to compute it. The alpha value is passed through a separately specified transfer function, allowing the masking effect to be customized.

7.4.2 Deriving a Soft Mask from Group Luminosity

The second method of deriving a soft mask from a transparency group begins by compositing the group with a fully opaque backdrop of some selected color. The mask value at any given point is then defined to be the luminosity of the resulting color. This allows the mask to be derived from the shape and color of an arbitrary piece of artwork drawn with ordinary painting operators.

The color C used to create the mask from a group G is defined by

$$\begin{aligned} \langle C_g, f_g, \alpha_g \rangle &= \text{Composite}(\dot{C}_0, 1, G) \\ C &= (1 - \alpha_g) \times C_0 + \alpha_g \times C_g \end{aligned}$$

where C_0 is the selected backdrop color.

G can be any kind of group—isolated or not, knockout or not—producing various effects on the *C* result in each case. The color *C* is then converted to luminosity in one of the following ways, depending on the group's color space:

- For CIE-based spaces, convert to the CIE 1931 *XYZ* space and use the *Y* component as the luminosity. This produces a colorimetrically correct luminosity. In the case of a PDF **CalRGB** space, the formula is

  $$Y = Y_A \times A^{G_R} + Y_B \times B^{G_G} + Y_C \times C^{G_B}$$

  using components of the **Gamma** and **Matrix** entries of the color space dictionary (see Table 4.14 on page 218). An analogous computation applies to other CIE-based color spaces.

- For device color spaces, convert the color to **DeviceGray** by device-dependent means and use the resulting gray value as the luminosity, with no compensation for gamma or other color calibration. This method makes no pretense of colorimetric correctness; it merely provides a numerically simple means to produce continuous-tone mask values. Here are some recommended formulas for converting from **DeviceRGB** and **DeviceCMYK**, respectively:

  $$Y = 0.30 \times R + 0.59 \times G + 0.11 \times B$$

  $$\begin{aligned} Y = {} & 0.30 \times (1 - C) \times (1 - K) \\ & + 0.59 \times (1 - M) \times (1 - K) \\ & + 0.11 \times (1 - Y) \times (1 - K) \end{aligned}$$

Following this conversion, the result is passed through a separately specified transfer function, allowing the masking effect to be customized.

The backdrop color most likely to be useful is black, which causes any areas outside the group's shape to end up with zero luminosity values in the resulting mask. If the contents of the group are viewed as a positive mask, this produces the results that would be expected with respect to points outside the shape.

7.5 Specifying Transparency in PDF

The preceding sections have presented the transparent imaging model at an abstract level, with little mention of its representation in PDF. This section describes the facilities available for specifying transparency in PDF 1.4.

7.5.1 Specifying Source and Backdrop Colors

Single graphics objects, as defined in Section 4.1, "Graphics Objects," are treated as elementary objects for transparency compositing purposes (subject to special treatment for text objects, as described in Section 5.2.7, "Text Knockout"). That is, all of a given object is considered to be one element of a transparency stack; portions of an object are not composited with one another, even if they are described in a way that would seem to cause overlaps (such as a self-intersecting path, combined fill and stroke of a path, or a shading pattern containing an overlap or fold-over). An object's source color C_s, used in the color compositing formula, is specified in the same way as in the opaque imaging model: via the current color in the graphics state or the source samples in an image. The backdrop color C_b is the result of previous painting operations.

7.5.2 Specifying Blending Color Space and Blend Mode

The blending color space is an attribute of the transparency group within which an object is painted; its specification is described below in Section 7.5.5, "Transparency Group XObjects." The page as a whole is also treated as a group, the *page group* (see Section 7.3.6, "Page Group"), with a color space attribute of its own. If not otherwise specified, the page group's color space is inherited from the native color space of the output device.

The blend mode $B(C_b, C_s)$ is determined by the *current blend mode* parameter in the graphics state (see Section 4.3, "Graphics State"), set via the **BM** entry in a graphics state parameter dictionary (Section 4.3.4, "Graphics State Parameter Dictionaries"). Its value is either a name object, designating one of the standard blend modes listed in Tables 7.2 and 7.3 on pages 481 and 483, or an array of such names. In the latter case, the viewer application should use the first blend mode in the array that it recognizes (or **Normal** if it recognizes none of them). This allows new blend modes to be introduced in the future while providing reasonable fallback behavior by viewer applications that do not recognize them. (See implementation note 64 in Appendix H.)

Note: *The current blend mode always applies to process color components, but only sometimes to spot colorants; see "Blend Modes and Overprinting" on page 524 for details.*

7.5.3 Specifying Shape and Opacity

As discussed under "Source Shape and Opacity" on page 485, the shape *(f)* and opacity *(q)* values used in the compositing computation can come from a variety of sources:

- The intrinsic shape (f_j) and opacity (q_j) of the object being composited
- A separate shape (f_m) or opacity (q_m) mask independent of the object itself
- A scalar shape (f_k) or opacity (q_k) constant to be added at every point

The following sections describe how each of these shape and opacity sources are specified in PDF.

Object Shape and Opacity

The shape value f_j of an object painted with PDF painting operators is defined as follows:

- For objects defined by a path or a glyph and painted in a uniform color with a path-painting or text-showing operator (Sections 4.4.2, "Path-Painting Operators," and 5.3.2, "Text-Showing Operators"), the shape is always 1.0 inside and 0.0 outside the path.
- For images (Section 4.8, "Images"), the shape is nominally 1.0 inside the image rectangle and 0.0 outside it; this can be further modified by an explicit or color key mask ("Explicit Masking" on page 314 and "Color Key Masking" on page 315).
- For image masks ("Stencil Masking" on page 313), the shape is 1.0 for painted areas and 0.0 for masked areas.
- For objects painted with a tiling pattern (Section 4.6.2, "Tiling Patterns") or a shading pattern (Section 4.6.3, "Shading Patterns), the shape is further constrained by the objects that define the pattern (see Section 7.5.6, "Patterns and Transparency").
- For objects painted with the **sh** operator ("Shading Operator" on page 266), the shape is 1.0 inside and 0.0 outside the bounds of the shading's painting geometry, disregarding the **Background** entry in the shading dictionary (see "Shading Dictionaries" on page 266).

All elementary objects have an intrinsic opacity q_j of 1.0 everywhere. Any desired opacity less than 1.0 must be applied by means of an opacity mask or constant, as described in the following sections.

Mask Shape and Opacity

At most one mask input—called a *soft mask*, or *alpha mask*—can be provided to any PDF compositing operation. The mask can serve as a source of either shape (f_m) or opacity (q_m) values, depending on the setting of the *alpha source* parameter in the graphics state (see Section 4.3, "Graphics State"). This is a boolean flag, set with the **AIS** ("alpha is shape") entry in a graphics state parameter dictionary (Section 4.3.4, "Graphics State Parameter Dictionaries"): **true** if the soft mask contains shape values, **false** for opacity.

The soft mask can be specified in one of the following ways:

- The *current soft mask* parameter in the graphics state, set with the **SMask** entry in a graphics state parameter dictionary, contains a *soft-mask dictionary* (see "Soft-Mask Dictionaries" on page 510) defining the contents of the mask. The name **None** may be specified in place of a soft-mask dictionary, denoting the absence of a soft mask; in this case, the mask shape or opacity is implicitly 1.0 everywhere. (See implementation note 64 in Appendix H.)
- An image XObject can contain its own *soft-mask image* in the form of a subsidiary image XObject in the **SMask** entry of the image dictionary (see Section 4.8.4, "Image Dictionaries"). This mask, if present, overrides any explicit or color key mask specified by the image dictionary's **Mask** entry; either form of mask in the image dictionary overrides the current soft mask in the graphics state. (See implementation note 65 in Appendix H.)
- An image XObject that has a **JPXDecode** filter as its data source can specify an **SMaskInData** entry, indicating that the soft mask is embedded in the data stream (see Section 3.3.8, "JPXDecode Filter").

***Note:** The current soft mask in the graphics state is intended to be used to clip only a single object at a time (either an elementary object or a transparency group). If a soft mask is applied when painting two or more overlapping objects, the effect of the mask will multiply with itself in the area of overlap (except in a knockout group), producing a result shape or opacity that is probably not what is intended. To apply a soft mask to multiple objects, it is usually best to define the objects as a transparency* *group and apply the mask to the group as a whole. These considerations also apply to the current alpha constant (see the next section).*

Constant Shape and Opacity

The *current alpha constant* parameter in the graphics state (see Section 4.3, "Graphics State") specifies two scalar values—one for strokes and one for all other painting operations—to be used for the constant shape (f_k) or constant opacity (q_k) component in the color compositing formulas. This parameter can be thought of as analogous to the current color used when painting elementary objects. (Note, however, that the nonstroking alpha constant is also applied when painting a transparency group's results onto its backdrop; see also implementation note 64 in Appendix H.)

The stroking and nonstroking alpha constants are set, respectively, by the **CA** and **ca** entries in a graphics state parameter dictionary (see Section 4.3.4, "Graphics State Parameter Dictionaries"). As described above for the soft mask, the alpha source flag in the graphics state determines whether the alpha constants are interpreted as shape values (**true**) or opacity values (**false**).

Note: *The note at the end of "Mask Shape and Opacity," above, applies to the current alpha constant parameter as well as the current soft mask.*

7.5.4 Specifying Soft Masks

As noted under "Mask Shape and Opacity" on page 508, soft masks for use in compositing computations can be specified in one of the following ways:

- as a soft-mask dictionary in the current soft mask parameter of the graphics state; see "Soft-Mask Dictionaries," below, for more details.
- as a soft-mask image associated with a sampled image; see "Soft-Mask Images" on page 512 for more details.
- (in PDF 1.5) as a mask channel embedded in JPEG2000 encoded data; see Section 3.3.8, "JPXDecode Filter" and the **SMaskInData** entry of Table 4.36 for more details.

Soft-Mask Dictionaries

The most common way of defining a soft mask is with a *soft-mask dictionary* specified as the current soft mask in the graphics state (see Section 4.3, “Graphics State”). Table 7.10 shows the contents of this type of dictionary. (See implementation note 64 in Appendix H.)

The mask values are derived from those of a transparency group, using one of the two methods described in Sections 7.4.1, “Deriving a Soft Mask from Group Alpha,” and 7.4.2, “Deriving a Soft Mask from Group Luminosity.” The group is defined by a transparency group XObject (see Section 7.5.5, “Transparency Group XObjects”) designated by the **G** entry in the soft-mask dictionary. The **S** (subtype) entry specifies which of the two derivation methods to use:

- If the subtype is **Alpha**, the transparency group XObject **G** is evaluated to compute a group alpha only; the colors of the constituent objects are ignored and the color compositing computations are not performed. The transfer function **TR** is then applied to the computed group alpha to produce the mask values. Outside the bounding box of the transparency group, the mask value is the result of applying the transfer function to the input value 0.0.
- If the subtype is **Luminosity**, the transparency group XObject **G** is composited with a fully opaque backdrop whose color is everywhere defined by the soft-mask dictionary’s **BC** entry. The computed result color is then converted to a single-component luminosity value and the transfer function **TR** is applied to this luminosity to produce the mask values. Outside the transparency group’s bounding box, the mask value is derived by transforming the **BC** color to luminosity and applying the transfer function to the result.

TABLE 7.10 Entries in a soft-mask dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Mask** for a soft-mask dictionary.

KEY	TYPE	VALUE
S	name	*(Required)* A subtype specifying the method to be used in deriving the mask values from the transparency group specified by the **G** entry: **Alpha** Use the group's computed alpha, disregarding its color (see Section 7.4.1, "Deriving a Soft Mask from Group Alpha"). **Luminosity** Convert the group's computed color to a single-component luminosity value (see Section 7.4.2, "Deriving a Soft Mask from Group Luminosity").
G	stream	*(Required)* A transparency group XObject (see Section 7.5.5, "Transparency Group XObjects") to be used as the source of alpha or color values for deriving the mask. If the subtype **S** is **Luminosity**, the group attributes dictionary must contain a **CS** entry defining the color space in which the compositing computation is to be performed.
BC	array	*(Optional)* An array of component values specifying the color to be used as the backdrop against which to composite the transparency group XObject **G**. This entry is consulted only if the subtype **S** is **Luminosity**. The array consists of *n* numbers, where *n* is the number of components in the color space specified by the **CS** entry in the group attributes dictionary (see Section 7.5.5, "Transparency Group XObjects"). Default value: the color space's initial value, representing black.
TR	function or name	*(Optional)* A function object (see Section 3.9, "Functions") specifying the transfer function to be used in deriving the mask values. The function accepts one input, the computed group alpha or luminosity (depending on the value of the subtype **S**), and returns one output, the resulting mask value. Both the input and output must be in the range 0.0 to 1.0; if the computed output falls outside this range, it is forced to the nearest valid value. The name **Identity** may be specified in place of a function object to designate the identity function. Default value: **Identity**.

The mask's coordinate system is defined by concatenating the transformation matrix specified by the **Matrix** entry in the transparency group's form dictionary (see Section 4.9.1, "Form Dictionaries") with the current transformation matrix at the moment the soft mask is established in the graphics state with the **gs** operator.

Note: *In a transparency group XObject that defines a soft mask, spot color components are never available, even if they are available in the group or page on which the soft mask is used. If the group XObject's content stream specifies a* ***Separation*** *or*

***DeviceN** color space that uses spot color components, the alternate color space will be substituted (see "Separation Color Spaces" on page 234 and "DeviceN Color Spaces" on page 238).*

Soft-Mask Images

The second way to define a soft mask is by associating a *soft-mask image* with an image XObject. This is a subsidiary image XObject specified in the **SMask** entry of the parent XObject's image dictionary (see Section 4.8.4, "Image Dictionaries"; see also implementation note 65 in Appendix H). Entries in the subsidiary image dictionary for such a soft-mask image have the same format and meaning as in that of an ordinary image XObject (as described in Table 4.36 on page 303), subject to the restrictions listed in Table 7.11. This type of image dictionary can also optionally contain an additional entry, **Matte**, discussed below.

When an image is accompanied by a soft-mask image, it is sometimes advantageous for the image data to be *preblended* with some background color, called the *matte color*. Each image sample represents a weighted average of the original source color and the matte color, using the corresponding mask sample as the weighting factor. (This is a generalization of a technique commonly called "premultiplied alpha.")

If the image data is preblended, the matte color must be specified by a **Matte** entry in the soft-mask image dictionary (see Table 7.12). The preblending computation, performed independently for each component, is as follows:

$$c' = m + \alpha \times (c - m)$$

where

c' is the value to be provided in the image source data

c is the original image component value

m is the matte color component value

α is the corresponding mask sample

***Note:** This computation uses actual color component values, with the effects of the **Filter** and **Decode** transformations already performed. The computation is the same whether the color space is additive or subtractive.*

TABLE 7.11 Restrictions on the entries in a soft-mask image dictionary

KEY	RESTRICTION
Type	If present, must be **XObject**.
Subtype	Must be **Image**.
Width	If a **Matte** entry (see Table 7.12, below) is present, must be the same as the **Width** value of the parent image; otherwise independent of it. Both images are mapped to the unit square in user space (as are all images), whether or not the samples coincide individually.
Height	Same considerations as for **Width**.
ColorSpace	Required; must be **DeviceGray**.
BitsPerComponent	Required.
Intent	Ignored.
ImageMask	Must be **false** or absent.
Mask	Must be absent.
SMask	Must be absent.
Decode	Default value: [0 1].
Interpolate	Optional.
Alternates	Ignored.
Name	Ignored.
StructParent	Ignored.
ID	Ignored.
OPI	Ignored.

TABLE 7.12 Additional entry in a soft-mask image dictionary

KEY	TYPE	VALUE
Matte	array	*(Optional; PDF 1.4)* An array of component values specifying the matte color with which the image data in the parent image has been preblended. The array consists of *n* numbers, where *n* is the number of components in the color space specified by the **ColorSpace** entry in the parent image's image dictionary; the numbers must be valid color components in that color space. If this entry is absent, the image data is not preblended.

When preblended image data is used in transparency blending and compositing computations, the results are the same as if the original, unblended image data were used and no matte color were specified. In particular, the inputs to the blend function are the original color values. This may sometimes require the viewer application to invert the formula shown above in order to derive c from c'. If the resulting c value lies outside the range of color component values for the image color space, the results are unpredictable.

The preblending computation is done in the color space specified by the parent image's **ColorSpace** entry. This is independent of the group color space into which the image may be painted; if a color conversion is required, inversion of the preblending must precede the color conversion. If the image color space is an **Indexed** space (see "Indexed Color Spaces" on page 232), it is the color values in the color table (not the index values themselves) that are preblended.

7.5.5 Transparency Group XObjects

A transparency group is represented in PDF as a special type of group XObject (see Section 4.9.2, "Group XObjects") called a *transparency group XObject*. A group XObject is in turn a type of form XObject, distinguished by the presence of a **Group** entry in its form dictionary (see Section 4.9.1, "Form Dictionaries"). The value of this entry is a subsidiary *group attributes dictionary* defining the properties of the group. The format and meaning of the dictionary's contents are determined by its *group subtype*, specified by the dictionary's **S** entry; those for a transparency group (subtype **Transparency**) are shown in Table 7.13.

***Note:** A page object (see "Page Objects" on page 118) may also have a **Group** entry, whose value is a group attributes dictionary specifying the attributes of the page group (see Section 7.3.6, "Page Group"). Some of the dictionary entries are inter-*

preted slightly differently for a page group than for a transparency group XObject; see their descriptions in the table for details.

TABLE 7.13 Additional entries specific to a transparency group attributes dictionary

KEY	TYPE	VALUE
S	name	*(Required)* The *group subtype*, which identifies the type of group whose attributes this dictionary describes; must be **Transparency** for a transparency group.
CS	name or array	*(Sometimes required, as discussed below)* The group color space, which is used for the following purposes: • As the color space into which colors are converted when painted into the group • As the blending color space in which objects are composited within the group (see Section 7.2.3, "Blending Color Space") • As the color space of the group as a whole when it in turn is painted as an object onto its backdrop The group color space may be any device or CIE-based color space that treats its components as independent additive or subtractive values in the range 0.0 to 1.0, subject to the restrictions described in Section 7.2.3, "Blending Color Space." These restrictions exclude **Lab** and lightness-chromaticity **ICCBased** color spaces, as well as the special color spaces **Pattern**, **Indexed**, **Separation**, and **DeviceN**. Device color spaces are subject to remapping according to the **DefaultGray**, **DefaultRGB**, and **DefaultCMYK** entries in the **ColorSpace** subdictionary of the current resource dictionary (see "Default Color Spaces" on page 227). Ordinarily, the **CS** entry is allowed only for isolated transparency groups (those for which **I**, below, is **true**) and even then it is optional. However, this entry is required in the group attributes dictionary for any transparency group XObject that has no parent group or page from which to inherit—in particular, one that is the value of the **G** entry in a soft-mask dictionary of subtype **Luminosity** (see "Soft-Mask Dictionaries" on page 510).

KEY	TYPE	VALUE
		In addition, it is always permissible to specify **CS** in the group attributes dictionary associated with a page object, even if **I** is **false** or absent. In the normal case in which the page is imposed directly on the output medium, the page group is effectively isolated regardless of the **I** value, and the specified **CS** value is therefore honored. But if the page is in turn used as an element of some other page and if the group is non-isolated, **CS** is ignored and the color space is inherited from the actual backdrop with which the page is composited (see Section 7.3.6, "Page Group"). Default value: the color space of the parent group or page into which this transparency group is painted. (The parent's color space in turn can be either explicitly specified or inherited.) ***Note:*** *For a transparency group XObject used as an annotation appearance (see Section 8.4.4, "Appearance Streams"), the default color space is inherited from the page on which the annotation appears.*
I	boolean	*(Optional)* A flag specifying whether the transparency group is isolated (see Section 7.3.4, "Isolated Groups"). If this flag is **true**, objects within the group are composited against a fully transparent initial backdrop; if **false**, they are composited against the group's backdrop. Default value: **false**. In the group attributes dictionary for a page, the interpretation of this entry is slightly altered. In the normal case in which the page is imposed directly on the output medium, the page group is effectively isolated and the specified **I** value is ignored. But if the page is in turn used as an element of some other page, it is treated as if it were a transparency group XObject; the **I** value is interpreted in the normal way to determine whether the page group is isolated.
K	boolean	*(Optional)* A flag specifying whether the transparency group is a knockout group (see Section 7.3.5, "Knockout Groups"). If this flag is **false**, later objects within the group are composited with earlier ones with which they overlap; if **true**, they are composited with the group's initial backdrop and overwrite ("knock out") any earlier overlapping objects. Default value: **false**.

The transparency group XObject's content stream defines the graphics objects belonging to the group. Invoking the **Do** operator on the XObject executes its content stream and composites the resulting group color, shape, and opacity into the group's parent group or page as if they had come from an elementary graphics object. When applied to a transparency group XObject, **Do** performs the following actions in addition to the normal ones for a form XObject (as described in Section 4.9, "Form XObjects"):

- If the transparency group is non-isolated (the value of the **I** entry in its group attributes dictionary is **false**), its initial backdrop, within the bounding box specified by the XObject's **BBox** entry, is defined to be the accumulated color and alpha of the parent group or page—that is, the result of everything that has been painted in the parent up to that point. (However, if the parent is a knockout group, the initial backdrop is the same as that of the parent.) If the group is isolated (**I** is **true**), its initial backdrop is defined to be transparent.
- Before execution of the transparency group XObject's content stream, the current blend mode in the graphics state is initialized to **Normal**, the current stroking and nonstroking alpha constants to 1.0, and the current soft mask to **None**.

  Note: *The purpose of initializing these graphics state parameters at the beginning of execution is to ensure that they are not applied twice: once when member objects are painted into the group and again when the group itself is painted into the parent group or page.*
- Objects painted by operators in the transparency group XObject's content stream are composited into the group according to the rules described in Section 7.2.2, "Basic Compositing Formula." The knockout flag (**K**) in the group attributes dictionary and the transparency-related parameters of the graphics state contribute to this computation.
- If a group color space (**CS**) is specified in the group attributes dictionary, all painting operators convert source colors to that color space before compositing objects into the group, and the resulting color at each point is interpreted in that color space. If no group color space is specified, the prevailing color space is dynamically inherited from the parent group or page. (If not otherwise specified, the page group's color space is inherited from the native color space of the output device.)
- After execution of the transparency group XObject's content stream, the graphics state reverts to its former state before the invocation of the **Do** operator (as it does for any form XObject). The group's shape—the union of all objects painted into the group, clipped by the group XObject's bounding box—is then painted into the parent group or page, using the group's accumulated color and opacity at each point.

Note: *If the* **Do** *operator is invoked more than once for a given transparency group XObject, each invocation is treated as a separate transparency group. That is, the* *result is as if the group were independently composited with the backdrop on each invocation. Viewer applications that perform caching of rendered form XObjects must take this requirement into account.*

The actions described above occur only for a transparency group XObject—a form XObject having a **Group** entry designating a group attributes subdictionary whose group subtype (**S**) is **Transparency**. An ordinary form XObject—one having no **Group** entry—is not subject to any grouping behavior for transparency purposes. That is, the graphics objects it contains are composited individually, just as if they were painted directly into the parent group or page.

7.5.6 Patterns and Transparency

In the transparent imaging model, the graphics objects making up the pattern cell of a tiling pattern (see Section 4.6.2, "Tiling Patterns") can include transparent objects and transparency groups. Transparent compositing can occur both within the pattern cell and between it and the backdrop wherever the pattern is painted. Similarly, a shading pattern (Section 4.6.3, "Shading Patterns") composites with its backdrop as if the shading dictionary were applied with the **sh** operator.

In both cases, the pattern definition is treated as if it were implicitly enclosed in a non-isolated transparency group: a non-knockout group for tiling patterns, a knockout group for shading patterns. The definition does *not* inherit the current values of the graphics state parameters at the time it is evaluated; these take effect only when the resulting pattern is later used to paint an object. Instead, the graphics state parameters are initialized as follows:

- As always for transparency groups, those parameters related to transparency (blend mode, soft mask, and alpha constant) are initialized to their standard default values.
- All other parameters are initialized to their values at the beginning of the content stream (such as a page or a form XObject) in which the pattern is defined as a resource; this is simply the normal behavior for all patterns, in both the opaque and transparent imaging models.
- In the case of a shading pattern, the parameter values may be augmented by the contents of the **ExtGState** entry in the pattern dictionary (see Section 4.6.3, "Shading Patterns"). Only those parameters that affect the **sh** operator, such as the current transformation matrix and rendering intent, are used; parameters

that affect path-painting operators are not, since the execution of **sh** does not entail painting a path.

- If the shading dictionary has a **Background** entry, the pattern's implicit transparency group is filled with the specified background color before the **sh** operator is invoked.

When the pattern is later used to paint a graphics object, the color, shape, and opacity values resulting from the evaluation of the pattern definition are used as the object's source color (C_s), object shape (f_j), and object opacity (q_j) in the transparency compositing formulas. This painting operation is subject to the values of the graphics state parameters in effect at the time, just as in painting an object with a constant color.

Unlike the opaque imaging model, in which the pattern cell of a tiling pattern can be evaluated once and then replicated indefinitely to fill the painted area, the effect in the general transparent case is as if the pattern definition were reexecuted independently for each tile, taking into account the color of the backdrop at each point. However, in the common case in which the pattern consists entirely of objects painted with the **Normal** blend mode, this behavior can be optimized by treating the pattern cell as if it were an isolated group. Since in this case the results depend only on the color, shape, and opacity of the pattern cell itself and not on those of the backdrop, the pattern cell can be evaluated once and then replicated, just as in opaque painting.

***Note:** In a raster-based implementation of tiling, it is important that all tiles together be treated as a single transparency group. This avoids artifacts due to multiple marking of pixels along the boundaries between adjacent tiles.*

The foregoing discussion applies to both colored (**PaintType** 1) and uncolored (**PaintType** 2) tiling patterns. In the latter case, the restriction that an uncolored pattern's definition may not specify colors extends as well to any transparency group that the definition may include. There are no corresponding restrictions, however, on specifying transparency-related parameters in the graphics state.

7.6 Color Space and Rendering Issues

This section describes the interactions between transparency and other aspects of color specification and rendering in the Adobe imaging model.

7.6.1 Color Spaces for Transparency Groups

As discussed in Section 7.5.5, "Transparency Group XObjects," a transparency group can either have an explicitly declared color space of its own or inherit that of its parent group. In either case, the colors of source objects within the group are converted to the group's color space, if necessary, and all blending and compositing computations are done in that space (see Section 7.2.3, "Blending Color Space"). The resulting colors are then interpreted in that color space when the group is subsequently composited with its backdrop.

Under this arrangement, it is envisioned that all or most of a given piece of artwork will be created in a single color space—most likely, the working color space of the application generating it. The use of multiple color spaces typically will arise only when assembling independently produced artwork onto a page. After all the artwork has been placed on the page, the conversion from the group's color space to the page's device color space will be done as the last step, without any further transparency compositing. The transparent imaging model does not require that this convention be followed, however; the reason for adopting it is to avoid the loss of color information and the introduction of errors resulting from unnecessary color space conversions.

Only an isolated group may have an explicitly declared color space of its own; non-isolated groups must inherit their color space from the parent group (subject to special treatment for the page group, as described in Section 7.3.6, "Page Group"). This is because the use of an explicit color space in a non-isolated group would require converting colors from the backdrop's color space to that of the group in order to perform the compositing computations. Such conversion may not be possible (since some color conversions can be performed only in one direction), and even if possible, it would entail an excessive number of color conversions.

The choice of a group color space will have significant effects on the results that are produced. In particular:

- As noted in Section 7.2.3, "Blending Color Space," the results of compositing in a device color space will be device-dependent; in order for the compositing computations to work in a device-independent way, the group's color space must be CIE-based.
- A consequence of choosing a CIE-based group color space is that only CIE-based spaces can be used to specify the colors of objects within the group. This

is because conversion from device to CIE-based colors is not possible in general; the defined conversions work only in the opposite direction. See below for further discussion.

- The compositing computations and blend functions generally compute linear combinations of color component values, on the assumption that the component values themselves are linear. For this reason, it is usually best to choose a group color space that has a linear gamma function. If a nonlinear color space is chosen, the results will still be well-defined, but the appearance may not match the user's expectations. Note, in particular, that the CIE-based *sRGB* color space (see page 225) is nonlinear, and hence may be unsuitable for use as a group color space.

Note: *Implementations of the transparent imaging model are advised to use as much precision as possible in representing colors during compositing computations and in the accumulated group results. To minimize the accumulation of roundoff errors and avoid additional errors arising from the use of linear group color spaces, more precision is needed for intermediate results than is typically used to represent either the original source data or the final rasterized results.*

If a group's color space—whether specified explicitly or inherited from the parent group—is CIE-based, any use of device color spaces for painting objects is subject to special treatment. Device colors cannot be painted directly into such a group, since there is no generally defined method for converting them to the CIE-based color space. This problem arises in the following cases:

- **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces, unless remapped to default CIE-based color spaces (see "Default Color Spaces" on page 227)
- Operators (such as **rg**) that specify a device color space implicitly, unless that space is remapped
- Special color spaces whose base or underlying space is a device color space, unless that space is remapped

It is recommended that the default color space remapping mechanism always be employed when defining a transparency group whose color space is CIE-based. If a device color is specified and is not remapped, it will be converted to the CIE-based color space in an implementation-dependent fashion, producing unpredictable results.

Note:** The foregoing restrictions do not apply if the group's color space is implicitly converted to* **DeviceCMYK, as discussed in "Implicit Conversion of CIE-Based Color Spaces" on page 228.*

7.6.2 Spot Colors and Transparency

The foregoing discussion of color spaces has been concerned with *process colors*—those produced by combinations of an output device's process colorants. Process colors may be specified directly in the device's native color space (such as **DeviceCMYK**), or they may be produced by conversion from some other color space, such as a CIE-based (**CalRGB** or **ICCBased**) space. Whatever means is used to specify them, process colors are subject to conversion to and from the group's color space.

A *spot color* is an additional color component, independent of those used to produce process colors. It may represent either an additional separation to be produced or an additional colorant to be applied to the composite page (see "Separation Color Spaces" on page 234 and "DeviceN Color Spaces" on page 238). The color component value, or *tint*, for a spot color specifies the concentration of the corresponding spot colorant. Tints are conventionally represented as subtractive, rather than additive, values.

Spot colors are inherently device-dependent and are not always available. In the opaque imaging model, each use of a spot color component in a **Separation** or **DeviceN** color space is accompanied by an *alternate color space* and a *tint transformation function* for mapping tint values into that space. This enables the color to be approximated with process colorants when the corresponding spot colorant is not available on the device.

Spot colors can be accommodated straightforwardly in the transparent imaging model (except for issues relating to overprinting, discussed in Section 7.6.3, "Overprinting and Transparency"). When an object is painted transparently with a spot color component that is available in the output device, that color is composited with the corresponding spot color component of the backdrop, independently of the compositing that is performed for process colors. A spot color retains its own identity; it is not subject to conversion to or from the color space of the enclosing transparency group or page. If the object is an element of a transparency group, one of two things can happen:

- The group maintains a separate color value for each spot color component, independently of the group's color space. In effect, the spot color passes directly through the group hierarchy to the device, with no color conversions performed; however, it is still subject to blending and compositing with other objects that use the same spot color.
- The spot color is converted to its alternate color space. The resulting color is then subject to the usual compositing rules for process colors. In particular, spot colors are never available in a transparency group XObject that is used to define a soft mask; the alternate color space will always be substituted in that case.

Only a single shape value and opacity value are maintained at each point in the computed group results; they apply to both process and spot color components. In effect, every object is considered to paint every existing color component, both process and spot. Where no value has been explicitly specified for a given component in a given object, an additive value of 1.0 (or a subtractive tint value of 0.0) is assumed. For instance, when painting an object with a color specified in a **DeviceCMYK** or **ICCBased** color space, the process color components are painted as specified and the spot color components are painted with an additive value of 1.0. Likewise, when painting an object with a color specified in a **Separation** color space, the named spot color is painted as specified and all other components (both process colors and other spot colors) are painted with an additive value of 1.0. The consequences of this are discussed in Section 7.6.3, "Overprinting and Transparency."

The opaque imaging model also allows process color components to be addressed individually, as if they were spot colors. For instance, it is possible to specify a **Separation** color space named **Cyan**, which paints just the cyan component on a *CMYK* output device. However, this capability is very difficult to extend to transparency groups. In general, the color components in a group are not the process colorants themselves, but are converted to process colorants only after the completion of all color compositing computations for the group (and perhaps some of its parent groups as well). For instance, if the group's color space is **ICCBased**, the group has no **Cyan** component to be painted. Consequently, treating a process color component as if it were a spot color is permitted only within a group that inherits the native color space of the output device (or is implicitly converted to **DeviceCMYK**, as discussed in "Implicit Conversion of CIE-Based Color Spaces" on page 228). Attempting to do so in a group that specifies its own color space will result in conversion of the requested spot color to its alternate color space.

7.6.3 Overprinting and Transparency

In the opaque imaging model, overprinting is controlled by two parameters of the graphics state: the *overprint parameter* and the *overprint mode* (see Section 4.5.6, "Overprint Control"). Painting an object causes some specific set of device colorants to be marked, as determined by the current color space and current color in the graphics state. The remaining colorants are either erased or left unchanged, depending on whether the overprint parameter is **false** or **true**. When the current color space is **DeviceCMYK**, the overprint mode parameter additionally enables this selective marking of colorants to be applied to individual color components according to whether the component value is zero or nonzero.

Because this model of overprinting deals directly with the painting of device colorants, independently of the color space in which source colors have been specified, it is highly device-dependent and primarily addresses production needs rather than design intent. Overprinting is usually reserved for opaque colorants or for very dark colors, such as black. It is also invoked during late-stage production operations such as trapping (see Section 10.10.5, "Trapping Support"), when the actual set of device colorants has already been determined.

Consequently, it is best to think of transparency as taking place in appearance space, but overprinting of device colorants in device space. This means that colorant overprint decisions should be made at output time, based on the actual resultant colorants of any transparency compositing operation. On the other hand, effects similar to overprinting can be achieved in a device-independent manner by taking advantage of blend modes, as described in the next section.

Blend Modes and Overprinting

As stated in Section 7.6.2, "Spot Colors and Transparency," each graphics object painted affects all existing color components: all of the process colorants in the transparency group's color space as well as any available spot colorants. For color components whose value has not been specified, a source color value of 1.0 is assumed; when objects are fully opaque and the **Normal** blend mode is used, this has the effect of erasing those components. This treatment is consistent with the behavior of the opaque imaging model with the overprint parameter set to **false**.

The transparent imaging model defines some blend modes, such as **Darken**, that can be used to achieve effects similar to overprinting. The blend function for **Darken** is

$$B(c_b, c_s) = \min(c_b, c_s)$$

In this blend mode, the result of compositing will always be the same as the backdrop color when the source color is 1.0, as it is for all unspecified color components. When the backdrop is fully opaque, this leaves the result color unchanged from that of the backdrop. This is consistent with the behavior of the opaque imaging model with the overprint parameter set to **true**.

If the object or backdrop is not fully opaque, the actions described above are altered accordingly. That is, the erasing effect is reduced, and overprinting an object with a color value of 1.0 may affect the result color. While these results may or may not be useful, they lie outside the realm of the overprinting and erasing behavior defined in the opaque imaging model.

When process colors are overprinted or erased (because a spot color is being painted), the blending computations described above are done independently for each component in the group's color space. If that space is different from the native color space of the output device, its components are not the device's actual process colorants; the blending computations affect the process colorants only after the group's results are converted to the device color space. Thus the effect is different from that of overprinting or erasing the device's process colorants directly. On the other hand, this is a fully general operation that works uniformly, regardless of the type of object or of the computations that produced the source color.

The discussion so far has focused on those color components whose values are *not* specified and that are to be either erased or left unchanged. However, the **Normal** or **Darken** blend modes used for these purposes may not be suitable for use on those components whose color values *are* specified. In particular, using the **Darken** blend mode for such components would preclude overprinting a dark color with a lighter one. Moreover, some other blend mode may be specifically desired for those components.

The PDF graphics state specifies only one current blend mode parameter, which always applies to process colorants and sometimes to spot colorants as well. Specifically, only separable, white-preserving blend modes can be used for spot colors. A blend mode is *white-preserving* if its blend function B has the property that $B(1.0, 1.0) = 1.0$. (Of the standard separable blend modes listed in Table 7.2 on page 481, all except **Difference** and **Exclusion** are white-preserving.) If the specified blend mode is not separable and white-preserving, it applies only to process color components; the **Normal** blend mode is substituted for spot colors. This ensures that when objects accumulate in an isolated transparency group, the accumulated values for unspecified components remain 1.0 so long as only white-preserving blend modes are used. The group's results can then be overprinted using **Darken** (or other useful modes) while avoiding unwanted interactions with components whose values were never specified within the group.

Compatibility with Opaque Overprinting

Because the use of blend modes to achieve effects similar to overprinting does not make direct use of the overprint control parameters in the graphics state, such methods are usable only by transparency-aware applications. For compatibility with the methods of overprint control used in the opaque imaging model, a special blend mode, CompatibleOverprint, is provided that consults the overprint-related graphics state parameters to compute its result. This mode applies only when painting elementary graphics objects (fills, strokes, text, images, and shadings). It is never invoked explicitly and is not identified by any PDF name object; rather, it is implicitly invoked whenever an elementary graphics object is painted while overprinting is enabled (that is, when the overprint parameter in the graphics state is **true**).

***Note:** Earlier designs of the transparent imaging model included an additional blend mode named **Compatible**, which explicitly invoked the CompatibleOverprint blend mode described here. Because CompatibleOverprint is now invoked implicitly whenever appropriate, it is never necessary to specify the **Compatible** blend mode for use in compositing. It is still recognized as a valid blend mode for the sake of compatibility, but is simply treated as equivalent to **Normal**.*

The value of the blend function $B(c_b, c_s)$ in the CompatibleOverprint mode is either c_b or c_s, depending on the setting of the overprint mode parameter, the current and group color spaces, and the source color value c_s:

- If the overprint mode is 1 (nonzero overprint mode) and the current color space and group color space are both **DeviceCMYK**, then only process color components with nonzero values replace the corresponding component values of the backdrop; all other component values leave the existing backdrop value unchanged. That is, the value of the blend function $B(c_b, c_s)$ is the source component c_s for any process (**DeviceCMYK**) color component whose (subtractive) color value is nonzero; otherwise it is the backdrop component c_b. For spot color components, the value is always c_b.

- In all other cases, the value of $B(c_b, c_s)$ is c_s for all color components specified in the current color space, otherwise c_b. For instance, if the current color space is **DeviceCMYK** or **CalRGB**, the value of the blend function is c_s for process color components and c_b for spot components. On the other hand, if the current color space is a **Separation** space representing a spot color component, the value is c_s for that spot component and c_b for all process components and all other spot components.

Note: *In the descriptions above, the term* current color space *refers to the color space used for a painting operation. This may be specified by the current color space parameter in the graphics state (see Section 4.5.1, "Color Values"), implicitly by color operators such as* ***rg*** *(Section 4.5.7, "Color Operators"), or by the* ***ColorSpace*** *entry of an image XObject (Section 4.8.4, "Image Dictionaries"). In the case of an* ***Indexed*** *space, it refers to the base color space (see "Indexed Color Spaces" on page 232); likewise for* ***Separation*** *and* ***DeviceN*** *spaces that revert to their alternate color space, as described under "Separation Color Spaces" on page 234 and "DeviceN Color Spaces" on page 238.*

If the current blend mode when CompatibleOverprint is invoked is any mode other than **Normal**, the object being painted is implicitly treated as if it were defined in a non-isolated, non-knockout transparency group and painted using the CompatibleOverprint blend mode; the group's results are then painted using the current blend mode in the graphics state.

Note: *It is not necessary to create such an implicit transparency group if the current blend mode is* ***Normal****; simply substituting the CompatibleOverprint blend mode while painting the object produces equivalent results. There are some additional cases in which the implicit transparency group can be optimized out.*

Plate 20 shows the effects of all four possible combinations of blending and overprinting, using the **Screen** blend mode in the **DeviceCMYK** color space. The label "overprint enabled" means that the overprint parameter in the graphics state is **true** and the overprint mode is 1. In the upper half of the figure, a light green oval is painted opaquely (opacity = 1.0) over a backdrop shading from pure yellow to pure magenta. In the lower half, the same object is painted with transparency (opacity = 0.5).

Special Path-Painting Considerations

The overprinting considerations discussed above also affect those path-painting operations that combine filling and stroking a path in a single operation. These include the **B**, **B***, **b**, and **b*** operators (see Section 4.4.2, "Path-Painting Operators") and the painting of glyphs with text rendering mode 2 or 6 (Section 5.2.5, "Text Rendering Mode"). For transparency compositing purposes, the combined fill and stroke are treated as a single graphics object, as if they were enclosed in a transparency group. This implicit group is established and used as follows:

- If overprinting is enabled (the overprint parameter in the graphics state is **true**) and the current stroking and nonstroking alpha constants are equal, a non-isolated, non-knockout transparency group is established. Within the group, the fill and stroke are performed with an alpha value of 1.0 but with the CompatibleOverprint blend mode. The group results are then composited with the backdrop using the originally specified alpha and blend mode.
- In all other cases, a non-isolated knockout group is established. Within the group, the fill and stroke are performed with their respective prevailing alpha constants and the prevailing blend mode. The group results are then composited with the backdrop using an alpha value of 1.0 and the **Normal** blend mode.

Note that in the case of showing text with the combined filling and stroking text rendering modes, this behavior is independent of the text knockout parameter in the graphics state (see Section 5.2.7, "Text Knockout").

The purpose of these rules is to avoid having a non-opaque stroke composite with the result of the fill in the region of overlap, which would produce a "double border" effect that is usually undesirable. The special case that applies when the overprint parameter is **true** is for backward compatibility with the overprinting behavior of the opaque imaging model. If a desired effect cannot be achieved with a combined filling and stroking operator or text rendering mode, it can be achieved by specifying the fill and stroke with separate path objects and an explicit transparency group.

Note: *Overprinting of the stroke over the fill does not work in the second case described above (although either the fill or the stroke can still overprint the backdrop). Furthermore, if the overprint graphics state parameter is* **true**, *the results are discontinuous at the transition between equal and unequal values of the stroking and nonstroking alpha constants. For this reason, it is best not to use overprinting for* *combined filling and stroking operations if the stroking and nonstroking alpha constants are being varied independently.*

Summary of Overprinting Behavior

Tables 7.14 and 7.15 summarize the overprinting and erasing behavior in the opaque and transparent imaging models, respectively. Table 7.14 shows the overprinting rules used in the opaque model, as described in Section 4.5.6, "Overprint Control"; Table 7.15 shows the equivalent rules as implemented by the CompatibleOverprint blend mode in the transparent model. The names **OP** and **OPM** in the tables refer to the overprint and overprint mode parameters of the graphics state.

TABLE 7.14 Overprinting behavior in the opaque imaging model

SOURCE COLOR SPACE	AFFECTED COLOR COMPONENT	EFFECT ON COLOR COMPONENT		
		OP FALSE	OP TRUE, OPM 0	OP TRUE, OPM 1
DeviceCMYK, specified directly, not in a sampled image	*C*, *M*, *Y*, or *K*	Paint source	Paint source	Paint source if $\neq 0.0$ Do not paint if $= 0.0$
	Process colorant other than *CMYK*	Paint source	Paint source	Paint source
	Spot colorant	Paint 0.0	Do not paint	Do not paint
Any process color space (including other cases of **DeviceCMYK**)	Process colorant	Paint source	Paint source	Paint source
	Spot colorant	Paint 0.0	Do not paint	Do not paint
Separation or **DeviceN**	Process colorant	Paint 0.0	Do not paint	Do not paint
	Spot colorant named in source space	Paint source	Paint source	Paint source
	Spot colorant not named in source space	Paint 0.0	Do not paint	Do not paint

TABLE 7.15 Overprinting behavior in the transparent imaging model

SOURCE COLOR SPACE	AFFECTED COLOR COMPONENT OF GROUP COLOR SPACE	VALUE OF BLEND FUNCTION $B(c_b, c_s)$ EXPRESSED AS TINT		
		OP FALSE	OP TRUE, OPM 0	OP TRUE, OPM 1
DeviceCMYK, specified directly, not in a sampled image	*C*, *M*, *Y*, or *K*	c_s	c_s	c_s if $c_s \neq 0.0$ c_b if $c_s = 0.0$
	Process color component other than *CMYK*	c_s	c_s	c_s
	Spot colorant	c_s (= 0.0)	c_b	c_b
Any process color space (including other cases of **DeviceCMYK**)	Process color component	c_s	c_s	c_s
	Spot colorant	c_s (= 0.0)	c_b	c_b
Separation or **DeviceN**	Process color component	c_s (= 0.0)	c_b	c_b
	Spot colorant named in source space	c_s	c_s	c_s
	Spot colorant not named in source space	c_s (= 0.0)	c_b	c_b
A group (not an elementary object)	All color components	c_s	c_s	c_s

Color component values are represented in these tables as subtractive tint values, because overprinting is typically applied to subtractive colorants such as inks, rather than to additive ones such as phosphors on a display screen. The CompatibleOverprint blend mode is therefore described as if it took subtractive arguments and returned subtractive results. In reality, however, Compatible-Overprint (like all blend modes) treats color components as additive values; subtractive components must be complemented before and after application of the blend function.

Note an important difference between the two tables. In Table 7.14, the process color components being discussed are the actual device colorants—the color components of the output device's native color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**). In Table 7.15, the process color components are those of the group's color space, which is not necessarily the same as that of the output device (and can even be something like **CalRGB** or **ICCBased**). For this reason, the process color components of the group color space cannot be treated as if they were spot colors in a **Separation** or **DeviceN** color space (see Section 7.6.2, "Spot Colors and Transparency"). This difference between opaque and transparent overprinting and erasing rules arises only within a transparency group (including the page group, if its color space is different from the native color space of the output device). There is no difference in the treatment of spot color components.

Table 7.15 has one additional row at the bottom. It applies when painting an object that is itself a transparency group rather than an elementary object (fill, stroke, text, image, or shading). As stated in Section 7.6.2, "Spot Colors and Transparency," a group is considered to paint all color components, both process and spot. Color components that were not explicitly painted by any object in the group have an additive color value of 1.0 (subtractive tint 0.0). Since no information is retained about which components were actually painted within the group, compatible overprinting is not possible in this case; the CompatibleOverprint blend mode reverts to **Normal**, with no consideration of the overprint and overprint mode parameters. (Note that a transparency-aware application can choose a more suitable blend mode, such as **Darken**, if it desires to produce an effect similar to overprinting.)

7.6.4 Rendering Parameters and Transparency

The opaque imaging model has several graphics state parameters dealing with the rendering of color: the current halftone (see Section 6.4.4, "Halftone Dictionaries"), transfer functions (Section 6.3, "Transfer Functions"), rendering intent ("Rendering Intents" on page 230), and black-generation and undercolor-removal functions (Section 6.2.3, "Conversion from DeviceRGB to DeviceCMYK"). All of these rendering parameters can be specified on a per-object basis; they control how a particular object will be rendered. When all objects are opaque, it is easy to define what this means. But when they are transparent, more than one object can contribute to the color at a given point; it is unclear which rendering parameters to apply in an area where transparent objects overlap. At the same time, the transparent imaging model should be consistent with the opaque model when only opaque objects are painted.

Furthermore, some of the rendering parameters—the halftone and transfer functions, in particular—can be applied only when the final color at a given point is known. In the presence of transparency, these parameters must be treated somewhat differently from those (rendering intent, black generation, and undercolor removal) that apply whenever colors must be converted from one color space to another. When objects are transparent, the rendering of an object does not occur when the object is specified, but at some later time; hence for rendering parameters in the former category, the implementation must keep track of the rendering parameters at each point from the time they are specified until the time the rendering actually occurs. This means that these rendering parameters must be associated with regions of the page rather than with individual objects.

Halftone and Transfer Function

The halftone and transfer function to be used at any given point on the page are those in effect at the time of painting the last (topmost) elementary graphics object enclosing that point, but only if the object is fully opaque. (Only elementary objects are relevant; the rendering parameters associated with a group object are ignored.) The *topmost object* at any point is defined to be the topmost elementary object in the entire page stack that has a nonzero object shape value (f_j) at that point (that is, for which the point is inside the object). An object is considered to be *fully opaque* if all of the following conditions hold at the time the object is painted:

- The current alpha constant in the graphics state (stroking or nonstroking, depending on the painting operation) is 1.0.
- The current blend mode in the graphics state is **Normal** (or **Compatible**, which is treated as equivalent to **Normal**).
- The current soft mask in the graphics state is **None**. If the object is an image XObject, there is no **SMask** entry in its image dictionary.
- The foregoing three conditions were also true at the time the **Do** operator was invoked for the group containing the object, as well as for any direct ancestor groups.
- If the current color is a tiling pattern, all objects in the definition of its pattern cell also satisfy the foregoing conditions.

Taken together, these conditions ensure that only the object itself contributes to the color at the given point, completely obscuring the backdrop. For portions of the page whose topmost object is not fully opaque or that are never painted at all, the default halftone and transfer function for the page are used.

Note:** If a graphics object is painted with overprinting enabled—that is, if the applicable (stroking or nonstroking) overprint parameter in the graphics state is* **true—the halftone and transfer function to use at a given point must be determined independently for each color component. Overprinting implicitly invokes the CompatibleOverprint blend mode (see "Compatibility with Opaque Overprinting" on page 526). An object is considered opaque for a given component only if CompatibleOverprint yields the source color (not the backdrop color) for that component.*

Rendering Intent and Color Conversions

The rendering intent, black-generation, and undercolor-removal parameters need to be handled somewhat differently. The rendering intent influences the conversion from a CIE-based color space to a target color space, taking into account the target space's color gamut (the range of colors it can reproduce). Whereas in the opaque imaging model the target space is always the native color space of the output device, in the transparent model it may instead be the group color space of a transparency group into which an object is being painted.

The rendering intent is needed at the moment such a conversion must be performed—that is, when painting an elementary or group object specified in a CIE-based color space into a parent group having a different color space. This differs from the current halftone and transfer function, whose values are used only when all color compositing has been completed and rasterization is being performed.

In all cases, the rendering intent to use for converting an object's color (whether that of an elementary object or of a transparency group) is determined by the rendering intent parameter associated with the object. In particular:

- When painting an elementary object with a CIE-based color into a transparency group having a different color space, the rendering intent used is the current rendering intent in effect in the graphics state at the time of the painting operation.
- When painting a transparency group whose color space is CIE-based into a parent group having a different color space, the rendering intent used is the current rendering intent in effect at the time the **Do** operator is applied to the group.

- When the color space of the page group is CIE-based, the rendering intent used to convert colors to the native color space of the output device is the default rendering intent for the page.

***Note:** Since there may be one or more nested transparency groups having different CIE-based color spaces, the color of an elementary source object may be converted to the device color space in multiple stages, controlled by the rendering intent in effect at each stage. The proper choice of rendering intent at each stage depends on the relative gamuts of the source and target color spaces. It is specified explicitly by the document producer, not prescribed by the PDF specification, since no single policy for managing rendering intents is appropriate for all situations.*

A similar approach works for the black-generation and undercolor-removal functions, which are applied only during conversion from **DeviceRGB** to **DeviceCMYK** color spaces:

- When painting an elementary object with a **DeviceRGB** color directly into a transparency group whose color space is **DeviceCMYK**, the functions used are the current black-generation and undercolor-removal functions in effect in the graphics state at the time of the painting operation.
- When painting a transparency group whose color space is **DeviceRGB** into a parent group whose color space is **DeviceCMYK**, the functions used are the ones in effect at the time the **Do** operator is applied to the group.
- When the color space of the page group is **DeviceRGB** and the native color space of the output device is **DeviceCMYK**, the functions used to convert colors to the device's color space are the default functions for the page.

7.6.5 PostScript Compatibility

Because the PostScript language does not support the transparent imaging model, PDF 1.4 viewer applications must have some means for converting the appearance of a document that uses transparency into a purely opaque description for printing on PostScript output devices. Similar techniques can also be used to convert such documents into a form that can be correctly viewed by PDF 1.3 and earlier viewers.

Converting the contents of a page from transparent to opaque form entails some combination of shape decomposition and prerendering to "flatten" the stack of transparent objects on the page, perform all the needed transparency computations, and describe the final appearance using opaque objects only. Whether the page contains transparent content needing to be flattened can be determined by straightforward analysis of the page's resources; it is not necessary to analyze the content stream itself. The conversion to opaque form is irreversible, since all information about how the transparency effects were produced is lost.

In order to perform the transparency computations properly, the viewer application needs to know the native color space of the output device. This is no problem when the viewer controls the output device directly. However, when generating PostScript output, the viewer has no way of knowing the native color space of the PostScript output device. An incorrect assumption will ruin the calibration of any CIE-based colors appearing on the page. This problem can be addressed in either of two ways:

- If the entire page consists of CIE-based colors, flatten the colors to a single CIE-based color space rather than to a device color space. The preferred color space for this purpose can easily be determined if the page has a group attributes dictionary (**Group** entry in the page object) specifying a CIE-based color space (see Section 7.5.5, "Transparency Group XObjects").
- Otherwise, flatten the colors to some assumed device color space with predetermined calibration. In the generated PostScript output, paint the flattened colors in a CIE-based color space having that calibration.

Because the choice between using spot colorants and converting them to an alternate color space affects the flattened results of process colors, a decision must also be made during PostScript conversion about the set of available spot colorants to assume. (This differs from strictly opaque painting, where the decision can be deferred until the generated PostScript code is executed.)

CHAPTER 8

Interactive Features

THIS CHAPTER DESCRIBES those features of PDF that allow a user to interact with a document on the screen, using the mouse and keyboard (with the exception of multimedia features, which are described in Chapter 9, "Multimedia Features"). These include:

- *Preference settings* to control the way the document is presented on the screen (Section 8.1, "Viewer Preferences")
- *Navigation* facilities for moving through the document in a variety of ways (Sections 8.2, "Document-Level Navigation," and 8.3, "Page-Level Navigation")
- *Annotations* for adding text notes, sounds, movies, and other ancillary information to the document (Section 8.4, "Annotations")
- *Actions* that can be triggered by specified events (Section 8.5, "Actions")
- *Interactive forms* for gathering information from the user (Section 8.6, "Interactive Forms")
- *Digital signatures* that authenticate the identity of a user and the validity of the document's contents (Section 8.7, "Digital Signatures")

8.1 Viewer Preferences

The **ViewerPreferences** entry in a document's catalog (see Section 3.6.1, "Document Catalog") designates a *viewer preferences dictionary (PDF 1.2)* controlling the way the document is to be presented on the screen or in print. If no such dictionary is specified, viewer applications should behave in accordance with their own current user preference settings. Table 8.1 shows the contents of the viewer preferences dictionary. (See implementation note 66 in Appendix H.)

TABLE 8.1 Entries in a viewer preferences dictionary

KEY	TYPE	VALUE
HideToolbar	boolean	*(Optional)* A flag specifying whether to hide the viewer application's tool bars when the document is active. Default value: **false**.
HideMenubar	boolean	*(Optional)* A flag specifying whether to hide the viewer application's menu bar when the document is active. Default value: **false**.
HideWindowUI	boolean	*(Optional)* A flag specifying whether to hide user interface elements in the document's window (such as scroll bars and navigation controls), leaving only the document's contents displayed. Default value: **false**.
FitWindow	boolean	*(Optional)* A flag specifying whether to resize the document's window to fit the size of the first displayed page. Default value: **false**.
CenterWindow	boolean	*(Optional)* A flag specifying whether to position the document's window in the center of the screen. Default value: **false**.
DisplayDocTitle	boolean	*(Optional; PDF 1.4)* A flag specifying whether the window's title bar should display the document title taken from the **Title** entry of the document information dictionary (see Section 10.2.1, "Document Information Dictionary"). If **false**, the title bar should instead display the name of the PDF file containing the document. Default value: **false**.
NonFullScreenPageMode	name	*(Optional)* The document's *page mode*, specifying how to display the document on exiting full-screen mode: UseNone Neither document outline nor thumbnail images visible UseOutlines Document outline visible UseThumbs Thumbnail images visible UseOC Optional content group panel visible This entry is meaningful only if the value of the **PageMode** entry in the catalog dictionary (see Section 3.6.1, "Document Catalog") is FullScreen; it is ignored otherwise. Default value: UseNone.
Direction	name	*(Optional; PDF 1.3)* The predominant reading order for text: L2R Left to right R2L Right to left (including vertical writing systems such as Chinese, Japanese, and Korean) This entry has no direct effect on the document's contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed *n*-up. Default value: L2R.

KEY	TYPE	VALUE
ViewArea	name	*(Optional; PDF 1.4)* The name of the page boundary representing the area of a page to be displayed when viewing the document on the screen. The value is the key designating the relevant page boundary in the page object (see "Page Objects" on page 118 and Section 10.10.1, "Page Boundaries"). If the specified page boundary is not defined in the page object, its default value will be used, as specified in Table 3.27 on page 118. Default value: **CropBox**. ***Note:*** *This entry is intended primarily for use by prepress applications that interpret or manipulate the page boundaries as described in Section 10.10.1, "Page Boundaries." Most PDF consumer applications will disregard it.*
ViewClip	name	*(Optional; PDF 1.4)* The name of the page boundary to which the contents of a page are to be clipped when viewing the document on the screen. The value is the key designating the relevant page boundary in the page object (see "Page Objects" on page 118 and Section 10.10.1, "Page Boundaries"). If the specified page boundary is not defined in the page object, its default value will be used, as specified in Table 3.27 on page 118. Default value: **CropBox**. ***Note:*** *This entry is intended primarily for use by prepress applications that interpret or manipulate the page boundaries as described in Section 10.10.1, "Page Boundaries." Most PDF consumer applications will disregard it.*
PrintArea	name	*(Optional; PDF 1.4)* The name of the page boundary representing the area of a page to be rendered when printing the document. The value is the key designating the relevant page boundary in the page object (see "Page Objects" on page 118 and Section 10.10.1, "Page Boundaries"). If the specified page boundary is not defined in the page object, its default value will be used, as specified in Table 3.27 on page 118. Default value: **CropBox**. ***Note:*** *This entry is intended primarily for use by prepress applications that interpret or manipulate the page boundaries as described in Section 10.10.1, "Page Boundaries." Most PDF consumer applications will disregard it.*

KEY	TYPE	VALUE
PrintClip	name	*(Optional; PDF 1.4)* The name of the page boundary to which the contents of a page are to be clipped when printing the document. The value is the key designating the relevant page boundary in the page object (see "Page Objects" on page 118 and Section 10.10.1, "Page Boundaries"). If the specified page boundary is not defined in the page object, its default value will be used, as specified in Table 3.27 on page 118. Default value: **CropBox**. ***Note:*** *This entry is intended primarily for use by prepress applications that interpret or manipulate the page boundaries as described in Section 10.10.1, "Page Boundaries." Most PDF consumer applications will disregard it.*

8.2 Document-Level Navigation

The features described in this section allow a PDF viewer application to present the user with an interactive, global overview of a document in either of two forms:

- As a hierarchical *outline* showing the document's internal structure
- As a collection of *thumbnail images* representing the pages of the document in miniature form

Each item in the outline or each thumbnail image can then be associated with a corresponding *destination* in the document, allowing the user to jump directly to that destination by clicking with the mouse.

8.2.1 Destinations

A *destination* defines a particular view of a document, consisting of the following:

- The page of the document to be displayed
- The location of the document window on that page
- The magnification (zoom) factor to use when displaying the page

Destinations may be associated with outline items (see Section 8.2.2, "Document Outline"), annotations ("Link Annotations" on page 576), or actions ("Go-To Actions" on page 598 and "Remote Go-To Actions" on page 599). In each case, the destination specifies the view of the document to be presented when the outline item or annotation is opened or the action is performed. In addition, the optional **OpenAction** entry in a document's catalog (Section 3.6.1, "Document Catalog") may specify a destination to be displayed when the document is opened. A destination may be specified either explicitly, by an array of parameters defining its properties, or indirectly by name.

Explicit Destinations

Table 8.2 shows the allowed syntactic forms for specifying a destination explicitly in a PDF file. In each case, *page* is an indirect reference to a page object. All coordinate values (*left*, *right*, *top*, and *bottom*) are expressed in the default user space coordinate system. The page's *bounding box* is the smallest rectangle enclosing all of its contents. (If any side of the bounding box lies outside the page's crop box, the corresponding side of the crop box is used instead; see Section 10.10.1, "Page Boundaries," for further discussion of the crop box.)

TABLE 8.2 Destination syntax

SYNTAX	MEANING
[*page* /XYZ *left top zoom*]	Display the page designated by *page*, with the coordinates (*left*, *top*) positioned at the top-left corner of the window and the contents of the page magnified by the factor *zoom*. A null value for any of the parameters *left*, *top*, or *zoom* specifies that the current value of that parameter is to be retained unchanged. A *zoom* value of 0 has the same meaning as a null value.
[*page* /Fit]	Display the page designated by *page*, with its contents magnified just enough to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the page within the window in the other dimension.
[*page* /FitH *top*]	Display the page designated by *page*, with the vertical coordinate *top* positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.
[*page* /FitV *left*]	Display the page designated by *page*, with the horizontal coordinate *left* positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.

SYNTAX	MEANING
[*page* /FitR *left bottom right top*]	Display the page designated by *page*, with its contents magnified just enough to fit the rectangle specified by the coordinates *left*, *bottom*, *right*, and *top* entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window in the other dimension.
[*page* /FitB]	*(PDF 1.1)* Display the page designated by *page*, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the bounding box within the window in the other dimension.
[*page* /FitBH *top*]	*(PDF 1.1)* Display the page designated by *page*, with the vertical coordinate *top* positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window.
[*page* /FitBV *left*]	*(PDF 1.1)* Display the page designated by *page*, with the horizontal coordinate *left* positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window.

***Note:** No page object can be specified for a destination associated with a remote go-to action (see "Remote Go-To Actions" on page 599), because the destination page is in a different PDF document. In this case, the* page *parameter specifies a page number within the remote document instead of a page object in the current document.*

Named Destinations

Instead of being defined directly, using the explicit syntax shown in Table 8.2, a destination may be referred to indirectly, using a name object *(PDF 1.1)* or a string *(PDF 1.2)*. This capability is especially useful when the destination is located in another PDF document. For example, a link to the beginning of Chapter 6 in another document might refer to the destination by a name such as Chap6.begin instead of giving an explicit page number in the other document. This would allow the location of the chapter opening to change within the other document without invalidating the link. If an annotation or outline item that refers to a named destination has an associated action, such as a remote go-to action (see "Remote Go-To Actions" on page 599) or a thread action ("Thread

Actions" on page 601), the destination is in the file specified by the action's **F** entry, if any; if there is no **F** entry, the destination is in the current file.

In PDF 1.1, the correspondence between name objects and destinations is defined by the **Dests** entry in the document catalog (see Section 3.6.1, "Document Catalog"). The value of this entry is a dictionary in which each key is a destination name and the corresponding value is either an array defining the destination, using the syntax shown in Table 8.2, or a dictionary with a **D** entry whose value is such an array. The latter form allows additional attributes to be associated with the destination, as well as enabling a go-to action (see "Go-To Actions" on page 598) to be used as the target of a named destination.

In PDF 1.2, the correspondence between strings and destinations is defined by the **Dests** entry in the document's name dictionary (see Section 3.6.3, "Name Dictionary"). The value of this entry is a name tree (Section 3.8.5, "Name Trees") mapping name strings to destinations. (The keys in the name tree may be treated as text strings for display purposes.) The destination value associated with a key in the name tree may be either an array or a dictionary, as described in the preceding paragraph.

***Note:** The use of strings as destination names is a PDF 1.2 feature. If compatibility with earlier versions of PDF is required, only name objects may be used to refer to named destinations. Where such compatibility is not a consideration, however, applications that generate large numbers of named destinations should use the string form of representation instead, since there are essentially no implementation limits.*

8.2.2 Document Outline

A PDF document may optionally display a *document outline* on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of *outline items* (sometimes called *bookmarks*), which serve as a "visual table of contents" to display the document's structure to the user. The user can interactively open and close individual items by clicking them with the mouse. When an item is open, its immediate children in the hierarchy become visible on the screen; each child may in turn be open or closed, selectively revealing or hiding further parts of the hierarchy. When an item is closed, all of its descendants in the hierarchy are hidden. Clicking the text of any visible item with the mouse *activates* the item, causing the viewer application to jump to a destination or trigger an action associated with the item.

The root of a document’s outline hierarchy is an *outline dictionary* specified by the **Outlines** entry in the document catalog (see Section 3.6.1, “Document Catalog”). Table 8.3 shows the contents of this dictionary. Each individual outline item within the hierarchy is defined by an *outline item dictionary* (Table 8.4). The items at each level of the hierarchy form a linked list, chained together through their **Prev** and **Next** entries and accessed through the **First** and **Last** entries in the parent item (or in the outline dictionary in the case of top-level items). When displayed on the screen, the items at a given level appear in the order in which they occur in the linked list. (See also implementation note 67 in Appendix H.)

TABLE 8.3 Entries in the outline dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Outlines** for an outline dictionary.
First	dictionary	*(Required if there are any open or closed outline entries; must be an indirect reference)* An outline item dictionary representing the first top-level item in the outline.
Last	dictionary	*(Required if there are any open or closed outline entries; must be an indirect reference)* An outline item dictionary representing the last top-level item in the outline.
Count	integer	*(Required if the document has any open outline entries)* The total number of open items at all levels of the outline. This entry should be omitted if there are no open outline items.

TABLE 8.4 Entries in an outline item dictionary

KEY	TYPE	VALUE
Title	text string	*(Required)* The text to be displayed on the screen for this item.
Parent	dictionary	*(Required; must be an indirect reference)* The parent of this item in the outline hierarchy. The parent of a top-level item is the outline dictionary itself.
Prev	dictionary	*(Required for all but the first item at each level; must be an indirect reference)* The previous item at this outline level.
Next	dictionary	*(Required for all but the last item at each level; must be an indirect reference)* The next item at this outline level.

KEY	TYPE	VALUE
First	dictionary	*(Required if the item has any descendants; must be an indirect reference)* The first of this item's immediate children in the outline hierarchy.
Last	dictionary	*(Required if the item has any descendants; must be an indirect reference)* The last of this item's immediate children in the outline hierarchy.
Count	integer	*(Required if the item has any descendants)* If the item is open, the total number of its open descendants at all lower levels of the outline hierarchy. If the item is closed, a negative integer whose absolute value specifies how many descendants would appear if the item were reopened.
Dest	name, string, or array	*(Optional; not permitted if an* ***A*** *entry is present)* The destination to be displayed when this item is activated (see Section 8.2.1, "Destinations"; see also implementation note 68 in Appendix H).
A	dictionary	*(Optional; PDF 1.1; not permitted if a* ***Dest*** *entry is present)* The action to be performed when this item is activated (see Section 8.5, "Actions").
SE	dictionary	*(Optional; PDF 1.3; must be an indirect reference)* The structure element to which the item refers (see Section 10.6.1, "Structure Hierarchy"). ***Note:*** *The ability to associate an outline item with a structure element (such as the beginning of a chapter) is a PDF 1.3 feature. For backward compatibility with earlier PDF versions, such an item should also specify a destination (****Dest****) corresponding to an area of a page where the contents of the designated structure element are displayed.*
C	array	*(Optional; PDF 1.4)* An array of three numbers in the range 0.0 to 1.0, representing the components in the **DeviceRGB** color space of the color to be used for the outline entry's text. Default value: [0.0 0.0 0.0].
F	integer	*(Optional; PDF 1.4)* A set of flags specifying style characteristics for displaying the outline item's text (see Table 8.5). Default value: 0.

The value of the outline item dictionary's **F** entry *(PDF 1.4)* is an unsigned 32-bit integer containing flags specifying style characteristics for displaying the item. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table 8.5 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

TABLE 8.5 Outline item flags

BIT POSITION	NAME	MEANING
1	Italic	If set, display the item in italic.
2	Bold	If set, display the item in bold.

Example 8.1 shows a typical outline dictionary and outline item dictionary. See Appendix G for an example of a complete outline hierarchy.

Example 8.1

```
21 0 obj
    << /Count 6
       /First 22 0 R
       /Last 29 0 R
    >>
endobj

22 0 obj
    << /Title (Chapter 1)
       /Parent 21 0 R
       /Next 26 0 R
       /First 23 0 R
       /Last 25 0 R
       /Count 3
       /Dest [3 0 R /XYZ 0 792 0]
    >>
endobj
```

8.2.3 Thumbnail Images

A PDF document may define *thumbnail images* representing the contents of its pages in miniature form. A viewer application can then display these images on the screen, allowing the user to navigate to a page by clicking its thumbnail image with the mouse.

Note: *Thumbnail images are not required, and may be included for some pages and not for others.*

The thumbnail image for a page is an image XObject specified by the **Thumb** entry in the page object (see “Page Objects” on page 118). It has the usual structure for an image dictionary (Section 4.8.4, "Image Dictionaries"), but only the **Width**, **Height**, **ColorSpace**, **BitsPerComponent**, and **Decode** entries are significant; all of the other entries listed in Table 4.36 on page 303 are ignored if present. (If a **Subtype** entry is specified, its value must be **Image**.) The image's color space must be either **DeviceGray** or **DeviceRGB**, or an **Indexed** space based on one of these. Example 8.2 shows a typical thumbnail image definition.

Example 8.2

```
12 0 obj
    << /Width 76
       /Height 99
       /ColorSpace /DeviceRGB
       /BitsPerComponent 8
       /Length 13 0 R
       /Filter [/ASCII85Decode /DCTDecode]
    >>
stream
s4IA>!"M;*Ddm8XA,lT0!!3,S!/(=R!<E3%!<N<(!WrK*!WrN,
…Omitted data…
endstream
endobj

13 0 obj                                        % Length of stream
    …
endobj
```

8.3 Page-Level Navigation

This section describes PDF facilities that allow the user to navigate from page to page within a document. These include:

- *Page labels* for numbering or otherwise identifying individual pages
- *Article threads*, which chain together items of content within the document that are logically connected but not physically sequential
- *Presentations* that display the document in the form of a "slide show," advancing from one page to the next either automatically or under user control

For another important form of page-level navigation, see "Link Annotations" on page 576.

8.3.1 Page Labels

Each page in a PDF document is identified by an integer *page index* that expresses the page's relative position within the document. In addition, a document may optionally define *page labels (PDF 1.3)* to identify each page visually on the screen or in print. Page labels and page indices need not coincide: the indices are fixed, running consecutively through the document starting from 0 for the first page, but the labels can be specified in any way that is appropriate for the particular document. For example, if the document begins with 12 pages of front matter numbered in roman numerals and the remainder of the document is numbered in arabic, then the first page would have a page index of 0 and a page label of i, the twelfth page would have index 11 and label xii, and the thirteenth page would have index 12 and label 1.

For purposes of page labeling, a document can be divided into *labeling ranges*, each of which is a series of consecutive pages using the same numbering system. Pages within a range are numbered sequentially in ascending order. A page's label consists of a numeric portion based on its position within its labeling range, optionally preceded by a *label prefix* denoting the range itself. For example, the pages in an appendix might be labeled with decimal numeric portions prefixed with the string A–; the resulting page labels would be A–1, A–2, and so on.

A document's labeling ranges are defined by the **PageLabels** entry in the document catalog (see Section 3.6.1, "Document Catalog"). The value of this entry is a number tree (Section 3.8.6, "Number Trees"), each of whose keys is the page index of the first page in a labeling range; the corresponding value is a *page label dictionary* defining the labeling characteristics for the pages in that range. The tree must include a value for page index 0. Table 8.6 shows the contents of a page label dictionary. (See implementation note 69 in Appendix H.)

Example 8.3 shows a document with pages labeled

i, ii, iii, iv, 1, 2, 3, A–8, A–9, …

Example 8.3

```
1 0 obj
   << /Type /Catalog
      /PageLabels << /Nums [ 0 << /S /r >>       % A number tree containing
                             4 << /S /D >>       %    three page label dictionaries
                             7 << /S /D
                                   /P (A–)
                                   /St 8
                                >>
                           ]
                  >>
      …
   >>
endobj
```

TABLE 8.6 Entries in a page label dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **PageLabel** for a page label dictionary.
S	name	*(Optional)* The numbering style to be used for the numeric portion of each page label: D Decimal arabic numerals R Uppercase roman numerals r Lowercase roman numerals A Uppercase letters (A to Z for the first 26 pages, AA to ZZ for the next 26, and so on) a Lowercase letters (a to z for the first 26 pages, aa to zz for the next 26, and so on) There is no default numbering style; if no **S** entry is present, page labels will consist solely of a label prefix with no numeric portion. For example, if the **P** entry (below) specifies the label prefix Contents, each page will simply be labeled Contents with no page number. (If the **P** entry is also missing or empty, the page label will be an empty string.)
P	text string	*(Optional)* The label prefix for page labels in this range.
St	integer	*(Optional)* The value of the numeric portion for the first page label in the range. Subsequent pages will be numbered sequentially from this value, which must be greater than or equal to 1. Default value: 1.

8.3.2 Articles

Some types of document may contain sequences of content items that are logically connected but not physically sequential. For example, a news story may begin on the first page of a newsletter and run over onto one or more nonconsecutive interior pages. To represent such sequences of physically discontiguous but logically related items, a PDF document may define one or more *articles (PDF 1.1)*. The sequential flow of an article is defined by an *article thread*; the individual content items that make up the article are called *beads* on the thread. PDF viewer applications can provide navigation facilities to allow the user to follow a thread from one bead to the next.

The optional **Threads** entry in the document catalog (see Section 3.6.1, "Document Catalog") holds an array of *thread dictionaries* (Table 8.7) defining the document's articles. Each individual bead within a thread is represented by a *bead dictionary* (Table 8.8). The thread dictionary's **F** entry points to the first bead in the thread; the beads are then chained together sequentially in a doubly linked list through their **N** (next) and **V** (previous) entries. In addition, for each page on which article beads appear, the page object (see "Page Objects" on page 118) should contain a **B** entry whose value is an array of indirect references to the beads on the page, in drawing order.

TABLE 8.7 Entries in a thread dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Thread** for a thread dictionary.
F	dictionary	*(Required; must be an indirect reference)* The first bead in the thread.
I	dictionary	*(Optional)* A thread information dictionary containing information about the thread, such as its title, author, and creation date. The contents of this dictionary are similar to those of the document information dictionary (see Section 10.2.1, "Document Information Dictionary").

TABLE 8.8 Entries in a bead dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Bead** for a bead dictionary.
T	dictionary	*(Required for the first bead of a thread; optional for all others; must be an indirect reference)* The thread to which this bead belongs. ***Note:*** *In PDF 1.1, this entry is permitted only for the first bead of a thread. In PDF 1.2 and higher, it is permitted for any bead but required only for the first.*
N	dictionary	*(Required; must be an indirect reference)* The next bead in the thread. In the last bead, this entry points to the first.
V	dictionary	*(Required; must be an indirect reference)* The previous bead in the thread. In the first bead, this entry points to the last.
P	dictionary	*(Required; must be an indirect reference)* The page object representing the page on which this bead appears.
R	rectangle	*(Required)* A rectangle specifying the location of this bead on the page.

Example 8.4 shows a thread with three beads.

Example 8.4

```
22 0 obj
   << /F 23 0 R
       /I << /Title (Man Bites Dog) >>
   >>
endobj

23 0 obj
   << /T 22 0 R
       /N 24 0 R
       /V 25 0 R
       /P 8 0 R
       /R [158 247 318 905]
   >>
endobj
```

```
24  0  obj
    <<  /T  22 0 R
        /N  25 0 R
        /V  23 0 R
        /P  8 0 R
        /R  [322 246 486 904]
    >>
endobj

25  0  obj
    <<  /T  22 0 R
        /N  23 0 R
        /V  24 0 R
        /P  10 0 R
        /R  [157 254 319 903]
    >>
endobj
```

8.3.3 Presentations

Some PDF viewer applications may allow a document to be displayed in the form of a *presentation* or "slide show," advancing from one page to the next either automatically or under user control. In addition, PDF 1.5 introduces the ability to advance between different states of the same page (see "Sub-page Navigation" on page 555).

Note: *PDF 1.4 introduces a different mechanism, known as* alternate presentations, *for slide show displays, described in Section 9.4, "Alternate Presentations."*

A page object (see "Page Objects" on page 118) may contain two optional entries, **Dur** and **Trans** *(PDF 1.1)*, to specify how to display that page in presentation mode. The **Trans** entry contains a *transition dictionary* describing the style and duration of the visual transition to use when moving from another page to the given page during a presentation. Table 8.9 shows the contents of the transition dictionary. (Some of the entries shown are needed only for certain transition styles, as indicated in the table.)

The **Dur** entry in the page object specifies the page's *display duration* (also called its *advance timing*): the maximum length of time, in seconds, that the page will be displayed before the presentation automatically advances to the next page. (The user can advance the page manually before the specified time has expired.) If no **Dur** entry is specified in the page object, the page does not advance automatically.

TABLE 8.9 Entries in a transition dictionary

KEY	TYPE	VALUE	
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Trans** for a transition dictionary.	
S	name	*(Optional)* The *transition style* to use when moving to this page from another during a presentation. Default value: R.	
		Split	Two lines sweep across the screen, revealing the new page. The lines may be either horizontal or vertical and may move inward from the edges of the page or outward from the center, as specified by the **Dm** and **M** entries, respectively.
		Blinds	Multiple lines, evenly spaced across the screen, synchronously sweep in the same direction to reveal the new page. The lines may be either horizontal or vertical, as specified by the **Dm** entry. Horizontal lines move downward, vertical lines to the right.
		Box	A rectangular box sweeps inward from the edges of the page or outward from the center, as specified by the **M** entry, revealing the new page.
		Wipe	A single line sweeps across the screen from one edge to the other in the direction specified by the **Di** entry, revealing the new page.
		Dissolve	The old page "dissolves" gradually to reveal the new one.
		Glitter	Similar to Dissolve, except that the effect sweeps across the page in a wide band moving from one side of the screen to the other in the direction specified by the **Di** entry.
		R	The new page simply replaces the old one with no special transition effect; the **D** entry is ignored.
		Fly	*(PDF 1.5)* Changes are "flown" out or in (as specified by **M**), in the direction specified by **Di**, to or from a location that is offscreen except when **Di** is None.
		Push	*(PDF 1.5)* The old page slides off the screen while the new page slides in, "pushing" the old page out in the direction specified by **Di**.
		Cover	*(PDF 1.5)* The new page slides on to the screen in the direction specified by **Di**, "covering" the old page.
		Uncover	*(PDF 1.5)* The old page slides off the screen in the direction specified by **Di**, "uncovering" the new page in the direction specified by **Di**.
		Fade	*(PDF 1.5)* The new page gradually becomes visible through the old one.

KEY	TYPE	VALUE
D	number	*(Optional)* The duration of the transition effect, in seconds. Default value: 1.
Dm	name	*(Optional; Split and Blinds transition styles only)* The dimension in which the specified transition effect occurs: H Horizontal V Vertical Default value: H.
M	name	*(Optional; Split, Box and Fly transition styles only)* The direction of motion for the specified transition effect: I Inward from the edges of the page O Outward from the center of the page Default value: I.
Di	number or name	*(Optional; Wipe, Glitter, Fly, Cover, Uncover and Push transition styles only)* The direction in which the specified transition effect moves, expressed in degrees counterclockwise starting from a left-to-right direction. (Note that this differs from the page object's **Rotate** entry, which is measured clockwise from the top.) The following numeric values are valid: 0 Left to right 90 Bottom to top (Wipe only) 180 Right to left (Wipe only) 270 Top to bottom 315 Top-left to bottom-right (Glitter only) The only valid name value is None, which is relevant only for the Fly transition when the value of SS is not 1.0. Default value: 0.
SS	number	(*Optional; PDF 1.5; Fly transition style only*) The starting or ending scale at which the changes are drawn. If **M** specifies an inward transition, the scale of the changes drawn progresses from **SS** to 1.0 over the course of the transition; if **M** specifies an outward transition, the scale of the changes drawn progresses from 1.0 to **SS** over the course of the transition Default: 1.0.
B	boolean	(*Optional; PDF 1.5; Fly transition style only*) If **true**, the area to be flown in is rectangular and opaque. Default: **false**.

Figure 8.1 illustrates the relationship between transition duration (**D** in the transition dictionary) and display duration (**Dur** in the page object). Note that the tran-

sition duration specified for a page (page 2 in the figure) governs the transition *to* that page from another page; the transition *from* the page is governed by the next page's transition duration.

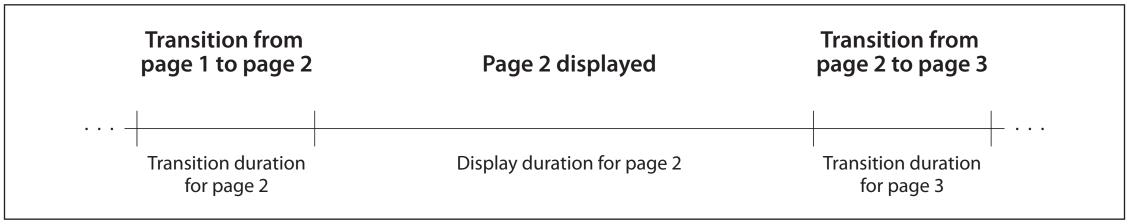

Transition from page 1 to page 2
Page 2 displayed
Transition from page 2 to page 3
. . .
. . .
Transition duration for page 2
Display duration for page 2
Transition duration for page 3

FIGURE 8.1 *Presentation timing*

Example 8.5 shows the presentation parameters for a page to be displayed for 5 seconds. Before the page is displayed, there is a 3.5-second transition in which two vertical lines sweep outward from the center to the edges of the page.

Example 8.5

```
10 0 obj
   << /Type /Page
      /Parent 4 0 R
      /Contents 16 0 R
      /Dur 5
      /Trans << /Type /Trans
                /D 3.5
                /S /Split
                /Dm /V
                /M /O
             >>
   >>
endobj
```

Sub-page Navigation

Sub-page navigation (PDF 1.5) allows navigating not only between pages but also between different states of the same page. For example, a single page in a PDF presentation could have a series of bullet points that could be individually turned on and off. In such an example, the bullets would be represented by optional content (see Section 4.10, "Optional Content"), and each state of the page would be represented as a *navigation node*.

Note: *Viewer applications should save the state of optional content groups when a user enters presentation mode and restore it when presentation mode ends. This ensures, for example, that transient changes to bullets do not affect the printing of the document.*

A navigation node dictionary (see Table 8.10) specifies actions to execute when the user makes a navigation request, for example by pressing an arrow key. The navigation nodes on a page form a doubly linked list by means of their **Next** and **Prev** entries. The primary node on a page is determined by the optional **PresSteps** entry in a page dictionary (see Table 3.27).

Note: *It is recommended that a viewer application respect navigation nodes only when in presentation mode (see Section 8.3.3, "Presentations").*

TABLE 8.10 Entries in a navigation node dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; must be **NavNode** for a navigation node dictionary.
NA	dictionary	*(Optional)* The sequence of actions to execute when a user navigates forward.
PA	dictionary	*(Optional)* The sequence of actions to execute when a user navigates backward.
Next	dictionary	*(Optional)* The next navigation node, if any.
Prev	dictionary	*(Optional)* The previous navigation node, if any.
Dur	number	*(Optional)* The maximum number of seconds before the viewer application should automatically advance forward to the next navigation node. If this entry is not specified, no automatic advance should occur.

A viewer application should support the notion of a *current* navigation node. When a user navigates to a page, if the page dictionary has a **PresSteps** entry, the node specified by that entry becomes the current node. (Otherwise, there is no current node.) If there is a request to navigate forward (such as an arrow key press), and there is a current navigation node, the following occurs:

1. The sequence of actions specified by **NA** (if present) is executed.

   Note: *If* ***NA*** *specifies an action that navigates to another page, the actions described below for navigating to another page take place, and* ***Next*** *should not be present.*

2. The node specified by **Next** (if present) becomes the new current navigation node.

Similarly, if there is a request to navigate backward, and there is a current navigation node, the following occurs:

1. The sequence of actions specified by **PA** (if present) is executed.

   Note: *If* ***PA*** *specifies an action that navigates to another page, the actions described below for navigating to another page take place, and* ***Prev*** *should not be present.*

2. The node specified by **Prev** (if present) becomes the new current navigation node.

When navigating between nodes, it is possible to specify transition effects. These are similar to the page transitions specified in the previous section. However, they use a different mechanism; see "Transition Actions" on page 611.

Note: *"Forward" and "backward" are determined by user actions, such as pressing right or left arrow keys, not by the actual page that is the destination of an action.*

If there is a request to navigate to another page (regardless of whether there is a current node), and that page's dictionary contains a **PresSteps** entry, the following occurs:

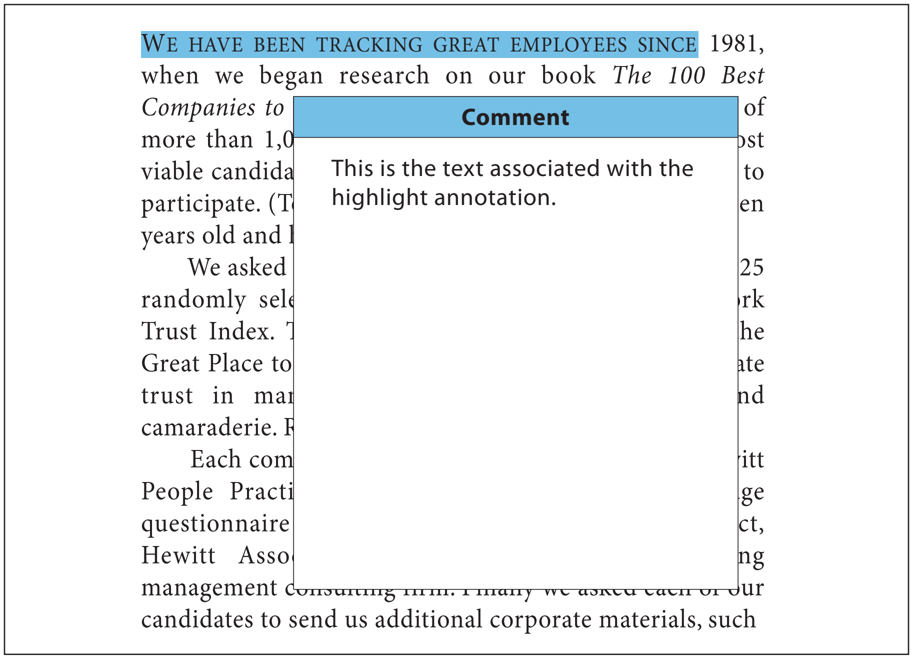
1. The navigation node represented by **PresSteps** becomes the current node.
2. If the navigation request was forward, or if the navigation request was for random access (such as by clicking on a link), the actions specified by **NA** are executed and the node specified by **Next** becomes the new current node, as described above.

If the navigation request was backward, the actions specified by **PA** are executed and the node specified by **Prev** becomes the new current node, as described above.

3. The viewer application makes the new page the current page and displays it. This includes any page transitions specified by the **Trans** entry of the page dictionary.

8.4 Annotations

An *annotation* associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a means of interacting with the user via the mouse and keyboard. PDF includes a wide variety of standard annotation types, described in detail in Section 8.4.5, "Annotation Types."


WE HAVE BEEN TRACKING GREAT EMPLOYEES SINCE 1981,
when we began research on our book *The 100 Best*
Companies to of
more than 1,0 ost
viable candida to
participate. (T en
years old and l
We asked 25
randomly sel rk
Trust Index. T he
Great Place to ate
trust in man nd
camaraderie. F
Each com itt
People Practi ge
questionnaire ct,
Hewitt Asso ng
management ur
candidates to send us additional corporate materials, such

Comment

This is the text associated with the highlight annotation.

FIGURE 8.2 *Open annotation*

Many of the standard annotation types may be displayed in either the *open* or the *closed* state. When closed, they appear on the page in some distinctive form depending on the specific annotation type, such as an icon, a box, or a rubber stamp. When the user *activates* the annotation by clicking it with the mouse, it exhibits its associated object, such as by opening a pop-up window displaying a text note (Figure 8.2) or by playing a sound or a movie.

Viewer applications may permit the user to navigate through the annotations on a page by using the keyboard (in particular, the tab key); see implementation note 70 in Appendix H. Starting with PDF 1.5, PDF producers may make the navigation order explicit with the optional **Tabs** entry in a page object (see Table 3.27). The possible values for this entry are:

- R (row order): annotations are visited in rows running horizontally across the page; the direction within a row is determined by the **Direction** entry in the **ViewerPreferences** dictionary (see Section 8.1, "Viewer Preferences"). The first annotation visited is the first annotation in the topmost row; when the end of a row is encountered, the first annotation in the next row is visited.
- C (column order): annotations are visited in columns running vertically up and down the page; columns are ordered by the **Direction** entry in the **ViewerPreferences** dictionary (see Section 8.1, "Viewer Preferences"). The first annotation visited is the one at the top of the first column; when the end of a column is encountered, the first annotation in the next column is visited.
- S (structure order): annotations are visited in the order they appear in the structure tree (see Section 10.6, "Logical Structure"). The order for annotations that are not included in the structure tree is application-dependent.

Note: *The descriptions above assume the page is being viewed in the orientation specified by the* ***Rotate*** *entry.*

The behavior of each annotation type is implemented by a software module called an *annotation handler.* Handlers for the standard annotation types are built directly into the PDF viewer application; handlers for additional types can be supplied as plug-in extensions.

8.4.1 Annotation Dictionaries

The optional **Annots** entry in a page object (see "Page Objects" on page 118) holds an array of *annotation dictionaries*, each representing an annotation associated with the given page. Table 8.11 shows the required and optional entries that are common to all annotation dictionaries. The dictionary may contain additional entries specific to a particular annotation type; see the descriptions of individual annotation types in Section 8.4.5, "Annotation Types," for details.

***Note:** A given annotation dictionary may be referenced from the* **Annots** *array of only one page. Attempting to share an annotation dictionary among multiple pages will produce unpredictable behavior. This requirement applies only to the annotation dictionary itself, not to subsidiary objects, which can be shared among multiple annotations without causing any difficulty.*

TABLE 8.11 Entries common to all annotation dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Annot** for an annotation dictionary.
Subtype	name	*(Required)* The type of annotation that this dictionary describes; see Table 8.16 on page 570 for specific values.
Contents	text string	*(Required or optional, depending on the annotation type)* Text to be displayed for the annotation or, if this type of annotation does not display text, an alternate description of the annotation's contents in human-readable form. In either case, this text is useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.2, "Alternate Descriptions"). See Section 8.4.5, "Annotation Types" for more details on the meaning of this entry for each annotation type.
P	dictionary	*(Optional; PDF 1.3; not used in FDF files)* An indirect reference to the page object with which this annotation is associated. ***Note:** This entry is required for screen annotations associated with rendition actions (PDF 1.5; see "Screen Annotations" on page 588 and "Rendition Actions" on page 609).*
Rect	rectangle	*(Required)* The *annotation rectangle*, defining the location of the annotation on the page in default user space units.
NM	text	*(Optional; PDF 1.4)* The *annotation name*, a text string uniquely identifying it among all the annotations on its page.
M	date or string	*(Optional; PDF 1.1)* The date and time when the annotation was most recently modified. The preferred format is a date string as described in Section 3.8.3, "Dates," but viewer applications should be prepared to accept and display a string in any format. (See implementation note 71 in Appendix H.)

KEY	TYPE	VALUE
F	integer	*(Optional; PDF 1.1)* A set of flags specifying various characteristics of the annotation (see Section 8.4.2, "Annotation Flags"). Default value: 0.
BS	dictionary	*(Optional; PDF 1.2)* A border style dictionary specifying the characteristics of the annotation's border (see Section 8.4.3, "Border Styles"; see also implementation note 72 in Appendix H). ***Note:** This entry also specifies the width and dash pattern for the lines drawn by line, square, circle, and ink annotations. See the note under **Border** (below) for additional information.*
Border	array	*(Optional)* An array specifying the characteristics of the annotation's border. The border is specified as a "rounded rectangle." In PDF 1.0, the array consists of three numbers defining the horizontal corner radius, vertical corner radius, and border width, all in default user space units. If the corner radii are 0, the border has square (not rounded) corners; if the border width is 0, no border is drawn. (See implementation note 74 in Appendix H.) In PDF 1.1, the array may have a fourth element, an optional *dash array* defining a pattern of dashes and gaps to be used in drawing the border. The dash array is specified in the same format as in the line dash pattern parameter of the graphics state (see "Line Dash Pattern" on page 187). For example, a **Border** value of `[0 0 1 [3 2]]` specifies a border 1 unit wide, with square corners, drawn with 3-unit dashes alternating with 2-unit gaps. Note that no dash phase is specified; the phase is assumed to be 0. (See implementation note 75 in Appendix H.) ***Note:** In PDF 1.2 or later, annotations may ignore this entry and use the **BS** entry (see above) to specify their border styles instead. In PDF 1.2 and 1.3, only widget annotations do so; in PDF 1.4, all of the standard annotation types except **Link** (see Table 8.16 on page 570) use **BS** rather than **Border** if both are present. For backward compatibility, however, **Border** is still supported for all annotation types.* Default value: `[0 0 1]`.
AP	dictionary	*(Optional; PDF 1.2)* An *appearance dictionary* specifying how the annotation is presented visually on the page (see Section 8.4.4, "Appearance Streams"; see also implementation notes 72 and 73 in Appendix H). Individual annotation handlers may ignore this entry and provide their own appearances.

KEY	TYPE	VALUE
AS	name	*(Required if the appearance dictionary* ***AP*** *contains one or more subdictionaries; PDF 1.2)* The annotation's *appearance state*, which selects the applicable appearance stream from an appearance subdictionary (see Section 8.4.4, "Appearance Streams"; see also implementation note 72 in Appendix H).
C	array	*(Optional; PDF 1.1)* An array of three numbers in the range 0.0 to 1.0, representing the components of a color in the **DeviceRGB** color space. This color will be used for the following purposes: • The background of the annotation's icon when closed • The title bar of the annotation's pop-up window • The border of a link annotation
A	dictionary	*(Optional; PDF 1.1)* An action to be performed when the annotation is activated (see Section 8.5, "Actions"). ***Note:*** *This entry is not permitted in link annotations if a* ***Dest*** *entry is present (see "Link Annotations" on page 576). Also note that the* ***A*** *entry in movie annotations has a different meaning (see "Movie Annotations" on page 587).*
AA	dictionary	*(Optional; PDF 1.2)* An additional-actions dictionary defining the annotation's behavior in response to various trigger events (see Section 8.5.2, "Trigger Events"). At the time of publication, this entry is used only by widget annotations.
StructParent	integer	*(Required if the annotation is a structural content item; PDF 1.3)* The integer key of the annotation's entry in the structural parent tree (see "Finding Structure Elements from Content Items" on page 739).
OC	dictionary	*(Optional; PDF 1.5)* An optional content group or optional content membership dictionary (see Section 4.10, "Optional Content"), specifying the optional content properties for the annotation. Before the annotation is drawn, its visibility is determined based on this entry as well as the annotation flags specified in the **F** entry (see Section 8.4.2, "Annotation Flags", below). If it is determined to be invisible, the annotation is skipped, as if it were not in the document at all.

8.4.2 Annotation Flags

The value of the annotation dictionary's **F** entry is an unsigned 32-bit integer containing flags specifying various characteristics of the annotation. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table

8.12 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

TABLE 8.12 Annotation flags

BIT POSITION	NAME	MEANING
1	Invisible	If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available. If clear, display such an unknown annotation using an appearance stream specified by its appearance dictionary, if any (see Section 8.4.4, "Appearance Streams").
2	Hidden	*(PDF 1.2)* If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available. In cases where screen space is limited, the ability to hide and show annotations selectively can be used in combination with appearance streams (see Section 8.4.4, "Appearance Streams") to display auxiliary pop-up information similar in function to online help systems. (See implementation note 76 in Appendix H.)
3	Print	*(PDF 1.2)* If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page. (See implementation note 76 in Appendix H.)
4	NoZoom	*(PDF 1.3)* If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification. See below for further discussion.
5	NoRotate	*(PDF 1.3)* If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation. See below for further discussion.
6	NoView	*(PDF 1.3)* If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the Print flag), but should be considered hidden for purposes of on-screen display and user interaction.

BIT POSITION	NAME	MEANING
7	ReadOnly	*(PDF 1.3)* If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the NoView and Print flags), but should not respond to mouse clicks or change its appearance in response to mouse motions. ***Note:*** *This flag is ignored for widget annotations; its function is subsumed by the ReadOnly flag of the associated form field (see Table 8.61 on page 616).*
8	Locked	*(PDF 1.4)* If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this does not restrict changes to the annotation's contents, such as a form field value. (See in Appendix H.)
9	ToggleNoView	*(PDF 1.5)* If set, invert the interpretation of the NoView flag for certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it; see implementation note 78 in Appendix H.

If the NoZoom flag is set, the annotation always maintains the same fixed size on the screen and is unaffected by the magnification level at which the page itself is displayed. Similarly, if the NoRotate flag is set, the annotation retains its original orientation on the screen when the page is rotated (by changing the **Rotate** entry in the page object; see "Page Objects" on page 118).

In either case, the annotation's position is determined by the coordinates of the upper-left corner of its annotation rectangle, as defined by the **Rect** entry in the annotation dictionary and interpreted in the default user space of the page. When the default user space is scaled or rotated, the positions of the other three corners of the annotation rectangle will be different in the altered user space than they were in the original user space. The viewer application performs this alteration automatically; however, it does not actually change the annotation's **Rect** entry, which continues to describe the annotation's relationship with the unscaled, unrotated user space.

For example, Figure 8.3 shows how an annotation whose NoRotate flag is set remains upright when the page it is on is rotated 90 degrees clockwise. The upper-left corner of the annotation remains at the same point in default user space; the annotation pivots around that point.

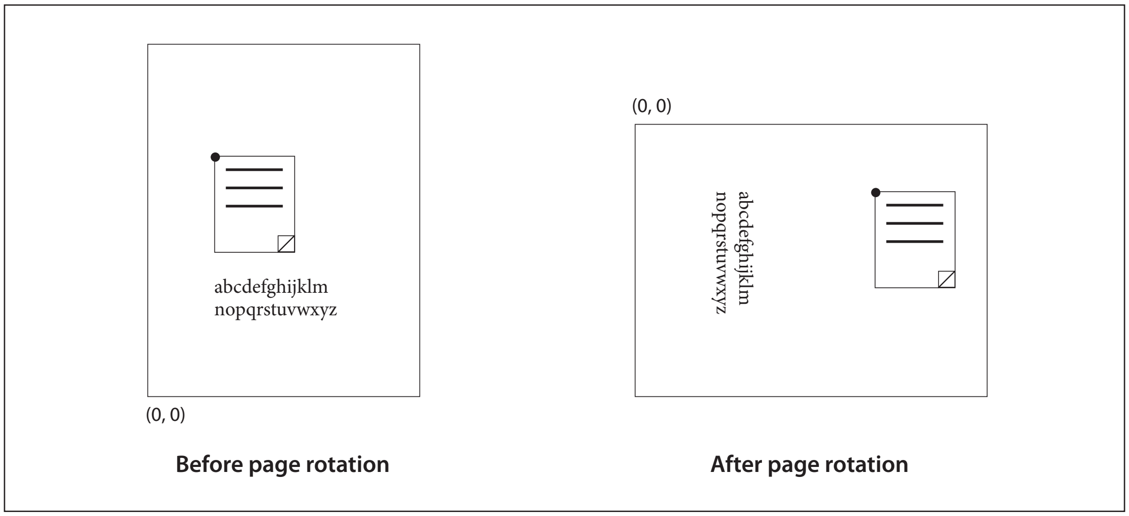


FIGURE 8.3 *Coordinate adjustment with the NoRotate flag*

8.4.3 Border Styles

An annotation may optionally be surrounded by a border when displayed or printed. If present, the border is drawn completely inside the annotation rectangle. In PDF 1.1, the characteristics of the border are specified by the **Border** entry in the annotation dictionary (see Table 8.11 on page 560). Beginning with PDF 1.2, some types of annotation may instead specify their border characteristics in a *border style dictionary* designated by the annotation's **BS** entry. Such dictionaries are also used to specify the width and dash pattern for the lines drawn by line, square, circle, and ink annotations. Table 8.13 summarizes the contents of the border style dictionary. If neither the **Border** nor the **BS** entry is present, the border is drawn as a solid line with a width of 1 point.

TABLE 8.13 Entries in a border style dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Border** for a border style dictionary.
W	number	*(Optional)* The border width in points. If this value is 0, no border is drawn. Default value: 1.

KEY	TYPE	VALUE
S	name	*(Optional)* The border style: S (Solid) A solid rectangle surrounding the annotation. D (Dashed) A dashed rectangle surrounding the annotation. The dash pattern is specified by the **D** entry (see below). B (Beveled) A simulated embossed rectangle that appears to be raised above the surface of the page. I (Inset) A simulated engraved rectangle that appears to be recessed below the surface of the page. U (Underline) A single line along the bottom of the annotation rectangle. Other border styles may be defined in the future. (See implementation note 79 in Appendix H.) Default value: S.
D	array	*(Optional)* A *dash array* defining a pattern of dashes and gaps to be used in drawing a dashed border (border style D above). The dash array is specified in the same format as in the line dash pattern parameter of the graphics state (see "Line Dash Pattern" on page 187). The dash phase is not specified and is assumed to be 0. For example, a **D** entry of [3 2] specifies a border drawn with 3-point dashes alternating with 2-point gaps. Default value: [3].

***Note:** In PDF 1.2 and 1.3, only widget annotations use border style dictionaries to specify their border styles; all of the other standard annotation types (see Table 8.16 on page 570) use the **Border** entry instead. In PDF 1.4, all except link annotations use border style dictionaries.*

Beginning with PDF 1.5, some annotations (square, circle and polygon) may have a **BE** entry, which is a *border effect dictionary* that specifies an effect to be applied to the border of the annotations. Its entries are listed in Table 8.14.

TABLE 8.14 Entries in a border effect dictionary

KEY	TYPE	VALUE
S	name	*(Optional)* A name representing the border effect to apply. Possible values are: S No effect: the border is as described by the annotation dictionary's **BS** entry. C The border should appear "cloudy". The width and dash array specified by **BS** are honored. Default value: S.
I	number	*(Optional; valid only if the value of **S** is C)* A number describing the intensity of the effect. Suggested values range from 0 to 2. Default value: 0.

8.4.4 Appearance Streams

Beginning with PDF 1.2, an annotation can specify one or more *appearance streams* as an alternative to the simple border and color characteristics available in earlier versions. This allows the annotation to be presented visually on the page in different ways to reflect its interactions with the user. Each appearance stream is a form XObject (see Section 4.9, "Form XObjects"): a self-contained content stream to be rendered inside the annotation rectangle.

The following method is used to map from the coordinate system of the appearance XObject (as defined by its **Matrix** entry; see Table 4.42) to the annotation's rectangle in default user space:

Algorithm 8.1

1. The appearance's bounding box (as specified by its **BBox** entry) is transformed, using **Matrix**, to produce a quadrilateral with arbitrary orientation. The *transformed appearance box* is defined as the smallest upright rectangle that encompasses this quadrilateral.
2. A matrix *A* is computed, which scales and translates the transformed appearance box to align with the edges of the annotation's rectangle (as specified by the **Rect** entry). *A* maps the lower-left corner (that is, the corner with the smallest x and y coordinates) and the upper-right corner (the corner with the greatest x and y coordinates) of the transformed appearance box to the corresponding corners of the annotation's rectangle.
3. **Matrix** is concatenated with *A* to form a matrix *AA* that maps from the appearance's coordinate system to the annotation's rectangle in default user space:

$AA = A \times \mathbf{Matrix}$

The annotation may be further scaled and rotated if either the NoZoom or NoRotate flag is set (see Section 8.4.2, "Annotation Flags"). Any transformation applied to the annotation as a whole is also applied to the appearance within it.

In PDF 1.4, an annotation appearance can include transparency. If the appearance's stream dictionary does not contain a **Group** entry, it is treated as a non-isolated, non-knockout transparency group; otherwise, the isolated and knockout values specified in the group dictionary (see Section 7.5.5, "Transparency Group XObjects") are used.

The transparency group is composited with a backdrop consisting of the page content along with any previously painted annotations, using a blend mode of **Normal**, an alpha constant of 1.0, and a soft mask of **None**. (See implementation note 80 in Appendix H.)

Note: *If a transparent annotation appearance is painted over an annotation that is drawn without using an appearance stream, the effect is implementation-dependent. This is because such annotations are sometimes drawn by means that do not conform to the Adobe imaging model. Also, the effect of highlighting a transparent annotation appearance is implementation-dependent.*

An annotation can define as many as three separate appearances:

- The *normal appearance* is used when the annotation is not interacting with the user. This is also the appearance that is used for printing the annotation.
- The *rollover appearance* is used when the user moves the cursor into the annotation's active area without pressing the mouse button.
- The *down appearance* is used when the mouse button is pressed or held down within the annotation's active area.

Note: *As used here, the term* mouse *denotes a generic pointing device that controls the location of a cursor on the screen and has at least one button that can be pressed, held down, and released. See Section 8.5.2, "Trigger Events," for further discussion.*

The normal, rollover, and down appearances are defined in an *appearance dictionary*, which in turn is the value of the **AP** entry in the annotation dictionary (see Table 8.11 on page 560). Table 8.15 shows the contents of the appearance dictionary.

TABLE 8.15 Entries in an appearance dictionary

KEY	TYPE	VALUE
N	stream or dictionary	*(Required)* The annotation's normal appearance.
R	stream or dictionary	*(Optional)* The annotation's rollover appearance. Default value: the value of the **N** entry.
D	stream or dictionary	*(Optional)* The annotation's down appearance. Default value: the value of the **N** entry.

Each entry in the appearance dictionary may contain either a single appearance stream or an *appearance subdictionary.* In the latter case, the subdictionary defines multiple appearance streams corresponding to different *appearance states* of the annotation.

For example, an annotation representing an interactive checkbox might have two appearance states named On and Off. Its appearance dictionary might be defined as follows:

```
/AP << /N << /On formXObject1
             /Off formXObject2
          >>
       /D << /On formXObject3
             /Off formXObject4
          >>
    >>
```

where *formXObject*$_1$ and *formXObject*$_2$ define the checkbox's normal appearance in its checked and unchecked states, while *formXObject*$_3$ and *formXObject*$_4$ provide visual feedback, such as emboldening its outline, when the user clicks it with the mouse. (No **R** entry is defined because no special appearance is needed when the user moves the cursor over the checkbox without pressing the mouse button.) The choice between the checked and unchecked appearance states is determined by the **AS** entry in the annotation dictionary (see Table 8.11 on page 560).

Note: *Some of the standard PDF annotation types, such as movie annotations—as well as all custom annotation types defined by third parties—are implemented through plug-in extensions. If the plug-in for a particular annotation type is not available, PDF viewer applications should display the annotation with its normal (**N**) appearance. Viewer applications should also attempt to provide reasonable be-*

havior (such as displaying nothing at all) if an annotation's ***AS*** *entry designates an appearance state for which no appearance is defined in the appearance dictionary.*

For convenience in managing appearance streams that are used repeatedly, the **AP** entry in a PDF document's name dictionary (see Section 3.6.3, "Name Dictionary") can contain a name tree mapping name strings to appearance streams. The name strings have no standard meanings; no PDF objects refer to appearance streams by name.

8.4.5 Annotation Types

PDF supports the standard annotation types listed in Table 8.16. The following sections describe each of these types in detail. Plug-in extensions may add new annotation types, and further standard types may be added in the future. (See implementation note 81 in Appendix H.)

The values in the first column of Table 8.16 represent the value of the annotation dictionary's **Subtype** entry. The third column indicates whether the annotation is a *markup annotation*, as described in "Markup Annotations," below. The section also provides more information about the value of the **Contents** entry for different annotation types.

TABLE 8.16 Annotation types

ANNOTATION TYPE	DESCRIPTION	MARKUP?	DISCUSSED IN SECTION
Text	Text annotation	Yes	"Text Annotations" on page 574
Link	Link annotation	No	"Link Annotations" on page 576
FreeText	*(PDF 1.3)* Free text annotation	Yes	"Free Text Annotations" on page 577
Line	*(PDF 1.3)* Line annotation	Yes	"Line Annotations" on page 578
Square	*(PDF 1.3)* Square annotation	Yes	"Square and Circle Annotations" on page 579
Circle	*(PDF 1.3)* Circle annotation	Yes	"Square and Circle Annotations" on page 579
Polygon	*(PDF 1.5)* Polygon annotation	Yes	"Polygon and Polyline Annotations" on page 581
Polyline	*(PDF 1.5)* Polyline annotation	Yes	"Polygon and Polyline Annotations" on page 581

ANNOTATION TYPE	DESCRIPTION	MARKUP?	DISCUSSED IN SECTION
Highlight	*(PDF 1.3)* Highlight annotation	Yes	"Text Markup Annotations" on page 582
Underline	*(PDF 1.3)* Underline annotation	Yes	"Text Markup Annotations" on page 582
Squiggly	*(PDF 1.4)* Squiggly-underline annotation	Yes	"Text Markup Annotations" on page 582
StrikeOut	*(PDF 1.3)* Strikeout annotation	Yes	"Text Markup Annotations" on page 582
Stamp	*(PDF 1.3)* Rubber stamp annotation	Yes	"Rubber Stamp Annotations" on page 584
Caret	*(PDF 1.5)* Caret annotation	Yes	"Caret Annotations" on page 583
Ink	*(PDF 1.3)* Ink annotation	Yes	"Ink Annotations" on page 584
Popup	*(PDF 1.3)* Pop-up annotation	No	"Pop-up Annotations" on page 585
FileAttachment	*(PDF 1.3)* File attachment annotation	Yes	"File Attachment Annotations" on page 586
Sound	*(PDF 1.2)* Sound annotation	Yes	"Sound Annotations" on page 586
Movie	*(PDF 1.2)* Movie annotation	No	"Movie Annotations" on page 587
Widget	*(PDF 1.2)* Widget annotation	No	"Widget Annotations" on page 588
Screen	*(PDF 1.5)* Screen annotation	No	"Screen Annotations" on page 588
PrinterMark	*(PDF 1.4)* Printer's mark annotation	No	"Printer's Mark Annotations" on page 591
TrapNet	*(PDF 1.3)* Trap network annotation	No	"Trap Network Annotations" on page 591

Markup Annotations

As mentioned in Section 8.4.1, "Annotation Dictionaries", the meaning of an annotation's **Contents** entry varies by annotation type. Typically, it is the text to be displayed for the annotation or, if the annotation does not display text, an alternate description of the annotation's contents in human-readable form. In either case, the **Contents** entry is useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.2, "Alternate Descriptions").

Many annotation types are defined as *markup annotations* because they are used primarily to mark up PDF documents (see Table 8.16). These annotations have text that appears as part of the annotation, and may be displayed in other ways by a viewer application, such as in a Comments pane.

Markup annotations can be divided into the following groups:

- Free text annotations display text directly on the page. The annotation's **Contents** entry specifies the displayed text.
- Most other markup annotations have an associated pop-up window that may contain text. The annotation's **Contents** entry specifies the text to be displayed when the pop-up window is opened. These include text, line, square, circle, polygon, polyline, highlight, underline, squiggly-underline, strikeout, rubber stamp, caret, ink, and file attachment annotations.
- Sound annotations do not have a pop-up window, but may also have associated text, specified by the **Contents** entry.

Note: *When separating text into paragraphs, a carriage return should be used (and not, for example, a line feed character).*

Note: *A subset of markup annotations are called* text markup annotations *(see "Text Markup Annotations" on page 582*).

The remaining annotation types are not considered markup annotations:

- The pop-up annotation type typically does not appear by itself, but is associated with a markup annotation that uses it to display text.

  Note: *The* **Contents** *entry for a pop-up annotation is relevant only if it has no parent; then it represents the text of the annotation.*

- For all other annotation types (**Link**, **Movie**, **Widget**, **PrinterMark**, and **TrapNet**), the **Contents** entry provides an alternate representation of the annotation's contents in human-readable form, useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.2, "Alternate Descriptions").

Table 8.17 lists entries that apply to all markup annotations.

TABLE 8.17 Additional entries specific to markup annotations

KEY	TYPE	VALUE
T	text string	*(Optional; PDF 1.1)* The text label to be displayed in the title bar of the annotation's pop-up window when open and active. By convention, this entry identifies the user who added the annotation.
Popup	dictionary	*(Optional; PDF 1.3)* An indirect reference to a pop-up annotation for entering or editing the text associated with this annotation.
CA	number	*(Optional; PDF 1.4)* The constant opacity value to be used in painting the annotation (see Sections 7.1, "Overview of Transparency," and 7.2.6, "Shape and Opacity Computations"). This value applies to all visible elements of the annotation in its closed state (including its background and border), but not to the pop-up window that appears when the annotation is opened. The specified value is not used if the annotation has an appearance stream (see Section 8.4.4, "Appearance Streams"); in that case, the appearance stream itself must specify any desired transparency. (However, if the viewer regenerates the annotation's appearance stream, it may incorporate the **CA** value into the stream's content.) The implicit blend mode (see Section 7.2.4, "Blend Mode") is **Normal**. Default value: 1.0. ***Note:*** *If no explicit appearance stream is defined for the annotation, it will be painted by implementation-dependent means that do not necessarily conform to the Adobe imaging model; in this case, the effect of this entry is implementation-dependent as well.*
RC	text string or text stream	*(Optional; PDF 1.5)* A rich text string (see "Rich Text Strings" on page 620) to be displayed in the pop-up window when the annotation is opened.
CreationDate	date	*(Optional; PDF 1.5)* The date and time (Section 3.8.3, "Dates") when the annotation was created.
Subj	text string	*(Optional; PDF 1.5)* Text representing a short description of the subject being addressed by the annotation.

Annotation States

Beginning with PDF 1.5, annotations may have author-specific *state* associated with them. The state is not specified in the annotation itself, but in a separate text annotation that refers to the original annotation by means of its **IRT** ("in reply to") entry (see Table 8.19). States are grouped into a number of *state models*, as shown in Table 8.18.

TABLE 8.18 Annotation states

STATE MODEL	STATE	DESCRIPTION
Marked	Marked	The annotation has been "marked" by the user.
	Unmarked	The annotation has not been "marked" by the user (the default).
Review	Accepted	The user agrees with the change.
	Rejected	The user disagrees with the change.
	Cancelled	The change has been cancelled.
	Completed	The change has been completed.
	None	The user has indicated nothing about the change (the default).

Annotations can be thought of as initially being in the default state for each state model. State changes made by a user are indicated in a text annotation with the following entries:

- The **T** entry (see Table 8.17) specifies the user.
- The **IRT** entry (see Table 8.19)refers to the original annotation.
- **State** and **StateModel** (see Table 8.19) update the state of the original annotation, for the specified user.

Additional state changes are made by adding text annotations in reply to the previous reply for a given user.

Text Annotations

A *text annotation* represents a "sticky note" attached to a point in the PDF document. When closed, the annotation appears as an icon; when open, it displays a pop-up window containing the text of the note, in a font and size chosen by the viewer application. Text annotations do not scale and rotate with the page; they behave as if the NoZoom and NoRotate annotation flags (see Table 8.12 on page 563) were always set. Table 8.19 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.19 Additional entries specific to a text annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Text** for a text annotation.
Open	boolean	*(Optional)* A flag specifying whether the annotation should initially be displayed open. Default value: **false** (closed).
Name	name	*(Optional)* The name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for at least the following standard names: Comment Key Note Help NewParagraph Paragraph Insert Additional names may be supported as well. Default value: Note. ***Note:** The annotation dictionary's **AP** entry, if present, takes precedence over the **Name** entry; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*
IRT	dictionary	*(Required if **State** is present, otherwise optional; PDF 1.5)* A reference to the annotation which this annotation is "in reply to". Both annotations must be on the same page of the document. Annotations containing this entry should not be displayed individually, but rather along with the annotations to which they are in reply, in the form of threaded comments. (Pre-PDF 1.5 viewers will treat them as ordinary annotations.) If this entry is present in an FDF file (see Section 8.6.6, "Forms Data Format"), its type is not a dictionary but a text string containing the contents of the **NM** entry of the annotation being replied to, to allow for a situation where the annotation being replied to is not in the same FDF file.
State	text string	*(Optional; PDF 1.5)* The state to which the original annotation should be set; see "Annotation States," above. Default: "Unmarked" if **StateModel** is "Marked"; "None" if **StateModel** is "Review".
StateModel	text string	*(Required if **State** is present, otherwise optional; PDF 1.5)* The state model corresponding to **State**; see "Annotation States," above.

Example 8.6 shows the definition of a text annotation.

Example 8.6

```
22 0 obj
   << /Type /Annot
      /Subtype /Text
      /Rect [266 116 430 204]
      /Contents (The quick brown fox ate the lazy mouse.)
   >>
endobj
```

Link Annotations

A *link annotation* represents either a hypertext link to a destination elsewhere in the document (see Section 8.2.1, "Destinations") or an action to be performed (Section 8.5, "Actions"). Table 8.20 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.20 Additional entries specific to a link annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Link** for a link annotation.
Dest	array, name, or string	*(Optional; not permitted if an* ***A*** *entry is present)* A destination to be displayed when the annotation is activated (see Section 8.2.1, "Destinations"; see also implementation note 82 in Appendix H).
H	name	*(Optional; PDF 1.2)* The annotation's *highlighting mode*, the visual effect to be used when the mouse button is pressed or held down inside its active area: N (None) No highlighting. I (Invert) Invert the contents of the annotation rectangle. O (Outline) Invert the annotation's border. P (Push) Display the annotation as if it were being "pushed" below the surface of the page; see implementation note 83 in Appendix H. Default value: I. ***Note:*** *In PDF 1.1, highlighting is always done by inverting colors inside the annotation rectangle.*

KEY	TYPE	VALUE
PA	dictionary	*(Optional; PDF 1.3)* A URI action (see "URI Actions" on page 602) formerly associated with this annotation. When Web Capture (Section 10.9, "Web Capture") changes an annotation from a URI to a go-to action ("Go-To Actions" on page 598), it uses this entry to save the data from the original URI action so that it can be changed back in case the target page for the go-to action is subsequently deleted.

Example 8.7 shows a link annotation that jumps to a destination elsewhere in the document.

Example 8.7

```
93 0 obj
   << /Type /Annot
      /Subtype /Link
      /Rect [71 717 190 734]
      /Border [16 16 1]
      /Dest [3 0 R /FitR –4 399 199 533]
   >>
endobj
```

Free Text Annotations

A *free text annotation (PDF 1.3)* displays text directly on the page. Unlike an ordinary text annotation (see "Text Annotations" on page 574), a free text annotation has no open or closed state; instead of being displayed in a pop-up window, the text is always visible. Table 8.21 shows the annotation dictionary entries specific to this type of annotation. "Variable Text" on page 617 describes the process of using these entries to generate the appearance of the text in these annotations.

TABLE 8.21 Additional entries specific to a free text annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **FreeText** for a free text annotation.
DA	string	*(Required)* The default appearance string to be used in formatting the text (see "Variable Text" on page 617). ***Note:*** *The annotation dictionary's **AP** entry, if present, takes precedence over the **DA** entry; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

KEY	TYPE	VALUE
Q	integer	*(Optional; PDF 1.4)* A code specifying the form of *quadding* (justification) to be used in displaying the annotation's text: 0 Left-justified 1 Centered 2 Right-justified Default value: 0 (left-justified).
RC	text string or text stream	*(Optional; PDF 1.5)* A rich text string (see "Rich Text Strings" on page 620) to be used to generate the appearance of the annotation.
DS	text string	*(Optional; PDF 1.5)* A default style string, as described in "Rich Text Strings" on page 620.

Line Annotations

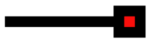
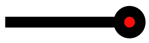
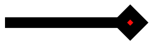
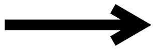
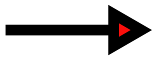
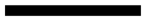
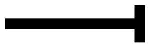
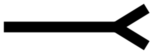
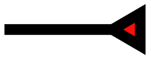
A *line annotation (PDF 1.3)* displays a single straight line on the page. When opened, it displays a pop-up window containing the text of the associated note. Table 8.22 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.22 Additional entries specific to a line annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Line** for a line annotation.
L	array	*(Required)* An array of four numbers, [x_1 y_1 x_2 y_2], specifying the starting and ending coordinates of the line in default user space.
BS	dictionary	*(Optional)* A border style dictionary (see Table 8.13 on page 565) specifying the width and dash pattern to be used in drawing the line. ***Note:*** *The annotation dictionary's* ***AP*** *entry, if present, takes precedence over the* ***L*** *and* ***BS*** *entries; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*
LE	array	*(Optional; PDF 1.4)* An array of two names specifying the line ending styles to be used in drawing the line. The first and second elements of the array specify the line ending styles for the endpoints defined, respectively, by the first and second pairs of coordinates, (x_1, y_1) and (x_2, y_2), in the **L** array. Table 8.23 shows the possible values. Default value: [/None /None].

KEY	TYPE	VALUE
IC	array	*(Optional; PDF 1.4)* An array of three numbers in the range 0.0 to 1.0 specifying the components, in the **DeviceRGB** color space, of the *interior color* with which to fill the annotation's line endings (see Table 8.23). If this entry is absent, the interiors of the line endings are left transparent.

TABLE 8.23 Line ending styles

NAME	APPEARANCE	DESCRIPTION
Square		A square filled with the annotation's interior color, if any
Circle		A circle filled with the annotation's interior color, if any
Diamond		A diamond shape filled with the annotation's interior color, if any
OpenArrow		Two short lines meeting in an acute angle, forming an open arrowhead
ClosedArrow		Two short lines meeting in an acute angle as in the OpenArrow style (see above), connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color, if any
None		No line ending
Butt		*(PDF 1.5)* A short line at the endpoint perpendicular to the line itself.
ROpenArrow		*(PDF 1.5)* Two short lines in the reverse direction from OpenArrow.
RClosedArrow		*(PDF 1.5)* A triangular closed arrowhead in the reverse direction from ClosedArrow.

Square and Circle Annotations

Square and *circle annotations (PDF 1.3)* display, respectively, a rectangle or an ellipse on the page. When opened, they display a pop-up window containing the text of the associated note. The rectangle or ellipse is inscribed within the annotation rectangle defined by the annotation dictionary's **Rect** entry (see Table 8.11 on page 560); Figure 8.4 shows an example, using a border width of 18 points. Despite the names *square* and *circle*, the width and height of the annotation rectangle need not be equal. Table 8.24 shows the annotation dictionary entries specific to these types of annotation.

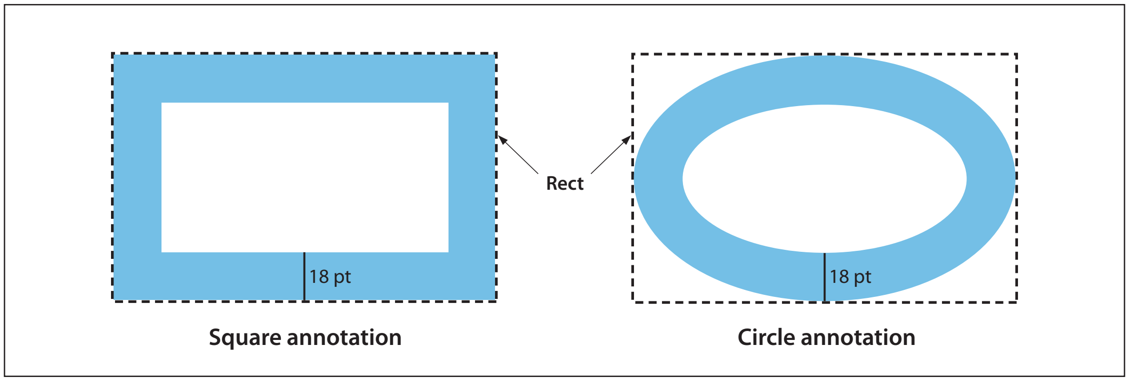


FIGURE 8.4 *Square and circle annotations*

TABLE 8.24 Additional entries specific to a square or circle annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Square** or **Circle** for a square or circle annotation, respectively.
BS	dictionary	*(Optional)* A border style dictionary (see Table 8.13 on page 565) specifying the line width and dash pattern to be used in drawing the rectangle or ellipse. ***Note:*** *The annotation dictionary's **AP** entry, if present, takes precedence over the **Rect** and **BS** entries; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*
IC	array	*(Optional; PDF 1.4)* An array of three numbers in the range 0.0 to 1.0 specifying the components, in the **DeviceRGB** color space, of the *interior color* with which to fill the annotation's rectangle or ellipse (see Table 8.23). If this entry is absent, the interior of the annotation is left transparent.
BE	dictionary	*(Optional; PDF 1.5)* A *border effect dictionary* describing an effect applied to the border described by the **BS** entry (see Table 8.14).

KEY	TYPE	VALUE
RD	rectangle	*(Optional; PDF 1.5)* A set of 4 numbers describing the numerical differences between two rectangles: the **Rect** entry of the annotation and the actual boundaries of the underlying square or circle. Such a difference can occur in situations where a border effect (described by **BE**) causes the size of the **Rect** to increase beyond that of the square or circle. The 4 numbers correspond to the differences in default user space between the left, top, right and bottom coordinates of **Rect** and those of the square or circle, respectively. Each value must be greater than or equal to 0. The sum of the top and bottom differences must be less than the height of **Rect**, and the sum of the left and right differences must be less than the width of **Rect**.

Polygon and Polyline Annotations

Polygon annotations (PDF 1.5) display closed polygons on the page. Such polygons may have any number of vertices, connected by straight lines. *Polyline annotations (PDF 1.5)* are similar to polygons, except that the first and last vertex are not implicitly connected.

TABLE 8.25 Additional entries specific to a polygon or polyline annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Polygon** or **Polyline** for a polygon or polyline annotation, respectively.
Vertices	array	*(Required)* An array of numbers representing the alternating horizontal and vertical coordinates, respectively, of each vertex, in default user space.
LE	array	*(Optional; meaningful only for **Polyline** annotations)* An array of two names specifying the line ending styles. The first and second elements of the array specify the line ending styles for the endpoints defined, respectively, by the first and last pairs of coordinates in the **Vertices** array. Table 8.23 shows the possible values. Default value: `[/None /None]`.
BS	dictionary	*(Optional)* A border style dictionary (see Table 8.13 on page 565) specifying the width and dash pattern to be used in drawing the line. ***Note:** The annotation dictionary's **AP** entry, if present, takes precedence over the **Vertices** and **BS** entries; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

KEY	TYPE	VALUE
IC	array	*(Optional; PDF 1.4)* An array of three numbers in the range 0.0 to 1.0 specifying the components, in the **DeviceRGB** color space, of the *interior color* with which to fill the annotation's line endings (see Table 8.23). If this entry is absent, the interiors of the line endings are left transparent.
BE	dictionary	*(Optional; meaningful only for **Polygon** annotations)* A *border effect dictionary* describing an effect applied to the border described by the **BS** entry (see Table 8.14).

Text Markup Annotations

Text markup annotations appear as highlights, underlines, strikeouts *(all PDF 1.3)*, or jagged ("squiggly") underlines *(PDF 1.4)* in the text of a document. When opened, they display a pop-up window containing the text of the associated note. Table 8.26 shows the annotation dictionary entries specific to these types of annotation.

TABLE 8.26 Additional entries specific to text markup annotations

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Highlight**, **Underline**, **Squiggly**, or **StrikeOut** for a highlight, underline, squiggly-underline, or strikeout annotation, respectively.
QuadPoints	array	*(Required)* An array of $8 \times n$ numbers specifying the coordinates of n quadrilaterals in default user space. Each quadrilateral encompasses a word or group of contiguous words in the text underlying the annotation. The coordinates for each quadrilateral are given in the order $x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ x_4\ y_4$ specifying the quadrilateral's four vertices in counterclockwise order (see Figure 8.5). The text is oriented with respect to the edge connecting points (x_1, y_1) and (x_2, y_2). (See implementation note 84 in Appendix H.) ***Note:** The annotation dictionary's **AP** entry, if present, takes precedence over the **QuadPoints** entry; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

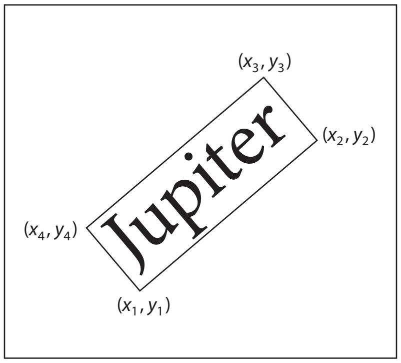


FIGURE 8.5 ***QuadPoints*** *specification*

Caret Annotations

A caret annotation *(PDF 1.5)* is a visual symbol that indicates the presence of text edits. Table 8.27 lists the entries specific to caret annotations.

TABLE 8.27 Additional entries specific to a caret annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Caret** for a caret annotation.
RD	rectangle	*(Optional; PDF 1.5)* A set of 4 numbers describing the numerical differences between two rectangles: the **Rect** entry of the annotation and the actual boundaries of the underlying caret. Such a difference can occur, for example, when a paragraph symbol specified by **Sy** is displayed along with the caret. The 4 numbers correspond to the differences in default user space between the left, top, right and bottom coordinates of **Rect** and those of the caret, respectively. Each value must be greater than or equal to 0. The sum of the top and bottom differences must be less than the height of **Rect**, and the sum of the left and right differences must be less than the width of **Rect**.
Sy	name	*(Optional)* A name specifying a symbol to be associated with the caret: P A new paragraph symbol (“¶”) should be associated with the caret. None No symbol should be associated with the caret. Default value: None.

Rubber Stamp Annotations

A *rubber stamp annotation (PDF 1.3)* displays text or graphics intended to look as if they were stamped on the page with a rubber stamp. When opened, it displays a pop-up window containing the text of the associated note. Table 8.28 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.28 Additional entries specific to a rubber stamp annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Stamp** for a rubber stamp annotation.
Name	name	*(Optional)* The name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for at least the following standard names: Approved, AsIs, Confidential, Departmental, Draft, Experimental, Expired, Final, ForComment, ForPublicRelease, NotApproved, NotForPublicRelease, Sold, TopSecret Additional names may be supported as well. Default value: Draft. ***Note:** The annotation dictionary's **AP** entry, if present, takes precedence over the **Name** entry; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

Ink Annotations

An *ink annotation (PDF 1.3)* represents a freehand "scribble" composed of one or more disjoint paths. When opened, it displays a pop-up window containing the text of the associated note. Table 8.29 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.29 Additional entries specific to an ink annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Ink** for an ink annotation.

KEY	TYPE	VALUE
InkList	array	*(Required)* An array of *n* arrays, each representing a stroked path. Each array is a series of alternating horizontal and vertical coordinates in default user space, specifying points along the path. When drawn, the points are connected by straight lines or curves in an implementation-dependent way. (See implementation note 85 in Appendix H.)
BS	dictionary	*(Optional)* A border style dictionary (see Table 8.13 on page 565) specifying the line width and dash pattern to be used in drawing the paths. ***Note:*** *The annotation dictionary's* ***AP*** *entry, if present, takes precedence over the* ***InkList*** *and* ***BS*** *entries; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

Pop-up Annotations

A *pop-up annotation (PDF 1.3)* displays text in a pop-up window for entry and editing. It typically does not appear alone, but is associated with a markup annotation, its *parent annotation*, and is used for editing the parent's text. It has no appearance stream or associated actions of its own, and is identified by the **Popup** entry in the parent's annotation dictionary (see Table 8.17 on page 573). Table 8.30 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.30 Additional entries specific to a pop-up annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Popup** for a pop-up annotation.
Parent	dictionary	*(Optional; must be an indirect reference)* The parent annotation with which this pop-up annotation is associated. ***Note:*** *If this entry is present, the parent annotation's* ***Contents***, ***M***, ***C***, *and* ***T*** *entries (see Table 8.11 on page 560) override those of the pop-up annotation itself.*
Open	boolean	*(Optional)* A flag specifying whether the pop-up annotation should initially be displayed open. Default value: **false** (closed).

File Attachment Annotations

A *file attachment annotation (PDF 1.3)* contains a reference to a file, which typically will be embedded in the PDF file (see Section 3.10.3, “Embedded File Streams”). For example, a table of data might use a file attachment annotation to link to a spreadsheet file based on that data; activating the annotation will extract the embedded file and give the user an opportunity to view it or store it in the file system. Table 8.31 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.31 Additional entries specific to a file attachment annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **FileAttachment** for a file attachment annotation.
FS	file specification	*(Required)* The file associated with this annotation.
Name	name	*(Optional)* The name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for at least the following standard names: Graph, PushPin, Paperclip, Tag. Additional names may be supported as well. Default value: PushPin. ***Note:** The annotation dictionary’s **AP** entry, if present, takes precedence over the **Name** entry; see Table 8.11 on page 560 and Section 8.4.4, “Appearance Streams.”*

Sound Annotations

A *sound annotation (PDF 1.2)* is analogous to a text annotation, except that instead of a text note, it contains sound recorded from the computer’s microphone or imported from a file. When the annotation is activated, the sound is played. The annotation behaves like a text annotation in most ways, with a different icon (by default, a speaker) to indicate that it represents a sound. Table 8.32 shows the annotation dictionary entries specific to this type of annotation; sounds themselves are discussed in Section 9.2, “Sounds.”

TABLE 8.32 Additional entries specific to a sound annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Sound** for a sound annotation.
Sound	stream	*(Required)* A sound object defining the sound to be played when the annotation is activated (see Section 9.2, "Sounds").
Name	name	*(Optional)* The name of an icon to be used in displaying the annotation. Viewer applications should provide predefined icon appearances for at least the standard names Speaker and Mic; additional names may be supported as well. Default value: Speaker. ***Note:*** *The annotation dictionary's* ***AP*** *entry, if present, takes precedence over the* ***Name*** *entry; see Table 8.11 on page 560 and Section 8.4.4, "Appearance Streams."*

Movie Annotations

A *movie annotation (PDF 1.2)* contains animated graphics and sound to be presented on the computer screen and through the speakers. When the annotation is activated, the movie is played. Table 8.33 shows the annotation dictionary entries specific to this type of annotation; movies themselves are discussed in Section 9.3, "Movies."

TABLE 8.33 Additional entries specific to a movie annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Movie** for a movie annotation.
Movie	dictionary	*(Required)* A movie dictionary describing the movie's static characteristics (see Section 9.3, "Movies").
A	boolean or dictionary	*(Optional)* A flag or dictionary specifying whether and how to play the movie when the annotation is activated. If this value is a dictionary, it is a movie activation dictionary (see Section 9.3, "Movies") specifying how to play the movie; if it is the boolean value **true**, the movie should be played using default activation parameters; if it is **false**, the movie should not be played at all. Default value: **true**.

Screen Annotations

A *screen annotation (PDF 1.5)* specifies a region of a page upon which media clips may be played. It also serves as an object from which actions can be triggered. "Rendition Actions" on page 609 discusses the relationship between screen annotations and rendition actions. Table 8.34 shows the annotation dictionary entries specific to this type of annotation.

TABLE 8.34 Additional entries specific to a screen annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Screen** for a screen annotation.
MK	dictionary	*(Optional)* An appearance characteristics dictionary (see Table 8.36). The **I** entry of this dictionary provides the icon used in generating the appearance referred to by the screen annotation's **AP** entry.

In addition to the above entries, screen annotations use the common entries in the annotation dictionary (see Table 8.11) in the following ways:

- The **P** entry is required for a screen annotation referenced by a rendition action; it must reference a valid page object, and the annotation must be present in the page's **Annots** array for the action to be valid.
- The **AP** entry refers to an appearance dictionary (see Table 8.15) whose normal appearance provides the visual appearance for a screen annotation that is used for printing and default display when a media clip is not being played. If **AP** is not present, the screen annotation has no default visual appearance and is not printed.
- The **AA** entry refers to an additional-actions dictionary (see Table 8.38) that contains four entries introduced in PDF 1.5 to support multimedia.

Widget Annotations

Interactive forms (see Section 8.6, "Interactive Forms") use *widget annotations (PDF 1.2)* to represent the appearance of fields and to manage user interactions. As a convenience, when a field has only a single associated widget annotation, the contents of the field dictionary (Section 8.6.2, "Field Dictionaries") and the annotation dictionary may be merged into a single dictionary containing entries that pertain to both a field and an annotation. (This presents no ambiguity, since the contents of the two kinds of dictionary do not conflict.) Table 8.35 shows the annotation dictionary entries specific to this type of annotation; interactive forms and fields are discussed at length in Section 8.6.

TABLE 8.35 Additional entries specific to a widget annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **Widget** for a widget annotation.
H	name	*(Optional)* The annotation's *highlighting mode*, the visual effect to be used when the mouse button is pressed or held down inside its active area: N (None) No highlighting. I (Invert) Invert the contents of the annotation rectangle. O (Outline) Invert the annotation's border. P (Push) Display the annotation's down appearance, if any (see Section 8.4.4, "Appearance Streams"). If no down appearance is defined, offset the contents of the annotation rectangle to appear as if it were being "pushed" below the surface of the page. T (Toggle) Same as P (which is preferred). A highlighting mode other than P overrides any down appearance defined for the annotation. Default value: I.
MK	dictionary	*(Optional)* An appearance characteristics dictionary (see Table 8.36) to be used in constructing a dynamic appearance stream specifying the annotation's visual presentation on the page. *The name **MK** for this entry is of historical significance only and has no direct meaning.*

The **MK** entry can be used to provide an *appearance characteristics dictionary* containing additional information for constructing the annotation's appearance stream. Table 8.36 shows the contents of this dictionary.

TABLE 8.36 Entries in an appearance characteristics dictionary

KEY	TYPE	VALUE
R	integer	*(Optional)* The number of degrees by which the widget annotation is rotated counterclockwise relative to the page. The value must be a multiple of 90. Default value: 0.
BC	array	*(Optional)* An array of numbers in the range 0.0 to 1.0 specifying the color of the widget annotation's border. The number of array elements determines the color space in which the color is defined: 0 No color; transparent 1 **DeviceGray** 3 **DeviceRGB** 4 **DeviceCMYK**
BG	array	*(Optional)* An array of numbers in the range 0.0 to 1.0 specifying the color of the widget annotation's background. The number of array elements determines the color space, as described above for **BC**.
CA	text string	*(Optional; button fields only)* The widget annotation's *normal caption*, displayed when it is not interacting with the user. ***Note:*** *Unlike the remaining entries listed below, which apply only to widget annotations associated with pushbutton fields (see "Pushbuttons" on page 626), the* ***CA*** *entry can be used with any type of button field, including checkboxes ("Checkboxes" on page 626) and radio buttons ("Radio Buttons" on page 628).*
RC	text string	*(Optional; pushbutton fields only)* The widget annotation's *rollover caption*, displayed when the user rolls the cursor into its active area without pressing the mouse button.
AC	text string	*(Optional; pushbutton fields only)* The widget annotation's *alternate (down) caption*, displayed when the mouse button is pressed within its active area.
I	stream	*(Optional; pushbutton fields only; must be an indirect reference)* A form XObject defining the widget annotation's *normal icon*, displayed when it is not interacting with the user.
RI	stream	*(Optional; pushbutton fields only; must be an indirect reference)* A form XObject defining the widget annotation's *rollover icon*, displayed when the user rolls the cursor into its active area without pressing the mouse button.
IX	stream	*(Optional; pushbutton fields only; must be an indirect reference)* A form XObject defining the widget annotation's *alternate (down) icon*, displayed when the mouse button is pressed within its active area.

KEY	TYPE	VALUE
IF	dictionary	*(Optional; pushbutton fields only)* An icon fit dictionary (see Table 8.88 on page 656) specifying how to display the widget annotation's icon within its annotation rectangle. If present, the icon fit dictionary applies to all of the annotation's icons (normal, rollover, and alternate).
TP	integer	*(Optional; pushbutton fields only)* A code indicating where to position the text of the widget annotation's caption relative to its icon: 0 No icon; caption only 1 No caption; icon only 2 Caption below the icon 3 Caption above the icon 4 Caption to the right of the icon 5 Caption to the left of the icon 6 Caption overlaid directly on the icon Default value: 0.

Printer's Mark Annotations

A *printer's mark annotation (PDF 1.4)* represents a graphic symbol, such as a registration target, color bar, or cut mark, added to a page to assist production personnel in identifying components of a multiple-plate job and maintaining consistent output during production. See Section 10.10.2, "Printer's Marks," for further discussion.

Trap Network Annotations

A *trap network annotation (PDF 1.3)* defines the trapping characteristics for a page of a PDF document. (*Trapping* is the process of adding marks to a page along color boundaries to avoid unwanted visual artifacts resulting from misregistration of colorants when the page is printed.) A page may have at most one trap network annotation, whose **Subtype** entry has the value **TrapNet** and which is always the last element in the page object's **Annots** array (see "Page Objects" on page 118). See Section 10.10.5, "Trapping Support," for further discussion.

8.5 Actions

Instead of simply jumping to a destination in the document, an annotation or outline item can specify an *action (PDF 1.1)* for the viewer application to perform, such as launching an application, playing a sound, or changing an annotation's appearance state. The optional **A** entry in the annotation or outline item dictionary (see Tables 8.11 on page 560 and 8.4 on page 544) specifies an action to be performed when the annotation or outline item is activated; in PDF 1.2, a variety of other circumstances may trigger an action as well (see Section 8.5.2, "Trigger Events"). In addition, the optional **OpenAction** entry in a document's catalog (Section 3.6.1, "Document Catalog") may specify an action to be performed when the document is opened. PDF includes a wide variety of standard action types, described in detail in Section 8.5.3, "Action Types."

8.5.1 Action Dictionaries

An *action dictionary* defines the characteristics and behavior of an action. Table 8.37 shows the required and optional entries that are common to all action dictionaries. The dictionary may contain additional entries specific to a particular action type; see the descriptions of individual action types in Section 8.5.3, "Action Types," for details.

TABLE 8.37 Entries common to all action dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Action** for an action dictionary.
S	name	*(Required)* The type of action that this dictionary describes; see Table 8.42 on page 597 for specific values.
Next	dictionary or array	*(Optional; PDF 1.2)* The next action, or sequence of actions, to be performed after this one. The value is either a single action dictionary or an array of action dictionaries to be performed in order; see below for further discussion.

The action dictionary's **Next** entry *(PDF 1.2)* allows sequences of actions to be chained together. For example, the effect of clicking a link annotation with the mouse might be to play a sound, jump to a new page, and start up a movie. Note that the **Next** entry is not restricted to a single action, but may contain an array of actions, each of which in turn may have a **Next** entry of its own. The actions may thus form a tree instead of a simple linked list. Actions within each **Next** array are executed in order, each followed in turn by any actions specified in *its* **Next** entry, and so on recursively. Viewer applications should attempt to provide reasonable behavior in anomalous situations; for example, self-referential actions should not be executed more than once, and actions that close the document or otherwise render the next action impossible should terminate the execution sequence. Applications should also provide some mechanism for the user to interrupt and manually terminate a sequence of actions.

PDF 1.5 introduces transition actions, which allow the control of drawing during a sequence of actions; see "Transition Actions" on page 611.

Note: *No action should modify its own action dictionary or any other in the action tree in which it resides. The effect of such modification on subsequent execution of actions in the tree is undefined.*

8.5.2 Trigger Events

An annotation, page object, or (beginning with PDF 1.3) interactive form field may include an entry named **AA** that specifies an *additional-actions dictionary (PDF 1.2)*, extending the set of events that can trigger the execution of an action. In PDF 1.4, the document catalog dictionary (see Section 3.6.1, "Document Catalog") may also contain an **AA** entry for trigger events affecting the document as a whole. Tables 8.38 to 8.41 show the contents of this type of dictionary. (See implementation notes 86 and 87 in Appendix H.)

PDF 1.5 introduces four new trigger events to support multimedia presentations:

- The **PO** and **PC** entries have a similar function to the **O** and **C** entries in the page object's additional-actions dictionary (see Table 8.39). However, associating these triggers with annotations allows annotation objects to be self-contained and greatly simplifies authoring. For example, annotations containing such actions can be copied or moved between pages without requiring page open/close actions to be changed.
- The **PV** and **PI** entries allow a distinction between pages that are open and pages that are visible. At any one time, only a single page is considered "open" in the viewer application, while more than one page may be visible, depending on the page layout.

***Note:** For these new trigger events, the values of the flags specified by the annotation's **F** entry (see Section 8.4.2, "Annotation Flags") have no bearing on whether or not a given trigger event occurs.*

TABLE 8.38 Entries in an annotation's additional-actions dictionary

KEY	TYPE	VALUE
E	dictionary	*(Optional; PDF 1.2)* An action to be performed when the cursor enters the annotation's active area.
X	dictionary	*(Optional; PDF 1.2)* An action to be performed when the cursor exits the annotation's active area.
D	dictionary	*(Optional; PDF 1.2)* An action to be performed when the mouse button is pressed inside the annotation's active area. (The name **D** stands for "down.")
U	dictionary	*(Optional; PDF 1.2)* An action to be performed when the mouse button is released inside the annotation's active area. (The name **U** stands for "up.") ***Note:** For backward compatibility, the **A** entry in an annotation dictionary, if present, takes precedence over this entry (see Table 8.11 on page 560).*
Fo	dictionary	*(Optional; PDF 1.2; widget annotations only)* An action to be performed when the annotation receives the input focus.
Bl	dictionary	*(Optional; PDF 1.2; widget annotations only)* (Uppercase B, lowercase L) An action to be performed when the annotation loses the input focus. (The name **Bl** stands for "blurred.")
PO	dictionary	*(Optional; PDF 1.5)* An action to be performed when the page containing the annotation is opened (for example, when the user navigates to it from the next or previous page or via a link annotation or outline item). The action is executed after the **O** action in the page's additional-actions dictionary (see Table 8.39) and the **OpenAction** entry in the document catalog (see Table 3.25), if such actions are present.
PC	dictionary	*(Optional; PDF 1.5)* An action to be performed when the page containing the annotation is closed (for example, when the user navigates to the next or previous page, or follows a link annotation or outline item). The action is executed before the **C** action in the page's additional-actions dictionary (see Table 8.39), if present.
PV	dictionary	*(Optional; PDF 1.5)* An action to be performed when the page containing the annotation becomes visible in the viewer application's user interface.
PI	dictionary	*(Optional; PDF 1.5)* An action to be performed when the page containing the annotation is no longer visible in the viewer application's user interface.

TABLE 8.39 Entries in a page object's additional-actions dictionary

KEY	TYPE	VALUE
O	dictionary	*(Optional; PDF 1.2)* An action to be performed when the page is opened (for example, when the user navigates to it from the next or previous page or via a link annotation or outline item). This action is independent of any that may be defined by the **OpenAction** entry in the document catalog (see Section 3.6.1, "Document Catalog"), and is executed after such an action. (See implementation note 88 in Appendix H.)
C	dictionary	*(Optional; PDF 1.2)* An action to be performed when the page is closed (for example, when the user navigates to the next or previous page or follows a link annotation or an outline item). This action applies to the page being closed, and is executed before any other page is opened. (See implementation note 88 in Appendix H.)

TABLE 8.40 Entries in a form field's additional-actions dictionary

KEY	TYPE	VALUE
K	dictionary	*(Optional; PDF 1.3)* A JavaScript action to be performed when the user types a keystroke into a text field or combo box or modifies the selection in a scrollable list box. This allows the keystroke to be checked for validity and rejected or modified.
F	dictionary	*(Optional; PDF 1.3)* A JavaScript action to be performed before the field is formatted to display its current value. This allows the field's value to be modified before formatting.
V	dictionary	*(Optional; PDF 1.3)* A JavaScript action to be performed when the field's value is changed. This allows the new value to be checked for validity. (The name **V** stands for "validate.")
C	dictionary	*(Optional; PDF 1.3)* A JavaScript action to be performed in order to recalculate the value of this field when that of another field changes. (The name **C** stands for "calculate.") The order in which the document's fields are recalculated is defined by the **CO** entry in the interactive form dictionary (see Section 8.6.1, "Interactive Form Dictionary").

TABLE 8.41 Entries in the document catalog's additional-actions dictionary

KEY	TYPE	VALUE
DC	dictionary	*(Optional; PDF 1.4)* A JavaScript action to be performed before closing a document. (The name **DC** stands for "document close.")
WS	dictionary	*(Optional; PDF 1.4)* A JavaScript action to be performed before saving a document. (The name **WS** stands for "will save.")

KEY	TYPE	VALUE
DS	dictionary	*(Optional; PDF 1.4)* A JavaScript action to be performed after saving a document. (The name **DS** stands for "did save.")
WP	dictionary	*(Optional; PDF 1.4)* A JavaScript action to be performed before printing a document. (The name **WP** stands for "will print.")
DP	dictionary	*(Optional; PDF 1.4)* A JavaScript action to be performed after printing a document. (The name **DP** stands for "did print.")

For purposes of the trigger events **E** (enter), **X** (exit), **D** (down), and **U** (up), the term *mouse* denotes a generic pointing device with the following characteristics:

- A selection button that can be *pressed*, *held down*, and *released*. If there is more than one mouse button, this is typically the left button.
- A notion of *location*—that is, an indication of where on the screen the device is pointing. This is typically denoted by a screen cursor.
- A notion of *focus*—that is, which element in the document is currently interacting with the user. In many systems, this is denoted by a blinking caret, a focus rectangle, or a color change.

PDF viewer applications must ensure the presence of such a device in order for the corresponding actions to be executed correctly. Mouse-related trigger events are subject to the following constraints:

- An **E** (enter) event can occur only when the mouse button is up.
- An **X** (exit) event cannot occur without a preceding **E** event.
- A **U** (up) event cannot occur without a preceding **E** and **D** event.
- In the case of overlapping or nested annotations, entering a second annotation's active area causes an **X** event to occur for the first annotation.

***Note:** The field-related trigger events **K** (keystroke), **F** (format), **V** (validate), and **C** (calculate) are not defined for button fields (see "Button Fields" on page 625). Note that the effects of an action triggered by one of these events are limited only by the action itself and can occur outside the described scope of the event. For example, even though the **F** event is used to trigger actions that format field values prior to display, it is possible for an action triggered by this event to perform a calculation or make any other modification to the document.*

Note also that these field-related trigger events can occur either through user interaction or programmatically, such as in response to the **NeedAppearances** *entry in the interactive form dictionary (see Section 8.6.1, "Interactive Form Dictionary"), importation of FDF data (Section 8.6.6, "Forms Data Format"), or JavaScript actions ("JavaScript Actions" on page 645). For example, the user's modifying a field value can trigger a cascade of calculations and further formatting and validation for other fields in the document.*

8.5.3 Action Types

PDF supports the standard action types listed in Table 8.42. The following sections describe each of these types in detail. Plug-in extensions may add new action types.

TABLE 8.42 Action types

ACTION TYPE	DESCRIPTION	DISCUSSED IN SECTION
GoTo	Go to a destination in the current document.	"Go-To Actions" on page 598
GoToR	("Go-to remote") Go to a destination in another document.	"Remote Go-To Actions" on page 599
Launch	Launch an application, usually to open a file.	"Launch Actions" on page 600
Thread	Begin reading an article thread.	"Thread Actions" on page 601
URI	Resolve a uniform resource identifier.	"URI Actions" on page 602
Sound	*(PDF 1.2)* Play a sound.	"Sound Actions" on page 604
Movie	*(PDF 1.2)* Play a movie.	"Movie Actions" on page 605
Hide	*(PDF 1.2)* Set an annotation's Hidden flag.	"Hide Actions" on page 606
Named	*(PDF 1.2)* Execute an action predefined by the viewer application.	"Named Actions" on page 607
SubmitForm	*(PDF 1.2)* Send data to a uniform resource locator.	"Submit-Form Actions" on page 639
ResetForm	*(PDF 1.2)* Set fields to their default values.	"Reset-Form Actions" on page 643
ImportData	*(PDF 1.2)* Import field values from a file.	"Import-Data Actions" on page 644
JavaScript	*(PDF 1.3)* Execute a JavaScript script.	"JavaScript Actions" on page 645

ACTION TYPE	DESCRIPTION	DISCUSSED IN SECTION
SetOCGState	*(PDF 1.5)* Set the states of optional content groups.	"Set-OCG-State Actions" on page 608
Rendition	*(PDF 1.5)* Controls the playing of multimedia content.	"Rendition Actions" on page 609
Trans	*(PDF 1.5)* Updates the display of a document using a transition dictionary.	"Transition Actions" on page 611

***Note:** Previous versions of the PDF specification described an action type known as the* set-state action*; this type of action is now considered obsolete and its use is no longer recommended. An additional action type, the* no-op action*, was defined in PDF 1.2 but never implemented; it is no longer defined and should be ignored.*

Go-To Actions

A *go-to action* changes the view to a specified destination (page, location, and magnification factor). Table 8.43 shows the action dictionary entries specific to this type of action.

TABLE 8.43 Additional entries specific to a go-to action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **GoTo** for a go-to action.
D	name, string, or array	*(Required)* The destination to jump to (see Section 8.2.1, "Destinations").

Specifying a go-to action in the **A** entry of a link annotation or outline item (see Tables 8.20 on page 576 and 8.4 on page 544) has the same effect as specifying the destination directly via the **Dest** entry. For example, the link annotation shown in Example 8.8, which uses a go-to action, has the same effect as the one in Example 8.7 on page 577, which specifies the destination directly. However, the go-to action is less compact and is not compatible with PDF 1.0, so using a direct destination is preferable.

Example 8.8

```
93 0 obj
   << /Type /Annot
       /Subtype /Link
       /Rect [71 717 190 734]
       /Border [16 16 1]
       /A << /Type /Action
              /S /GoTo
              /D [3 0 R /FitR –4 399 199 533]
           >>
   >>
endobj
```

Remote Go-To Actions

A *remote go-to action* is similar to an ordinary go-to action, but jumps to a destination in another PDF file instead of the current file. Table 8.44 shows the action dictionary entries specific to this type of action.

TABLE 8.44 Additional entries specific to a remote go-to action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **GoToR** for a remote go-to action.
F	file specification	*(Required)* The file in which the destination is located.
D	name, string, or array	*(Required)* The destination to jump to (see Section 8.2.1, "Destinations"). If the value is an array defining an explicit destination (as described under "Explicit Destinations" on page 541), its first element must be a page number within the remote document rather than an indirect reference to a page object in the current document. The first page is numbered 0.
NewWindow	boolean	*(Optional; PDF 1.2)* A flag specifying whether to open the destination document in a new window. If this flag is **false**, the destination document will replace the current document in the same window. If this entry is absent, the viewer application should behave in accordance with the current user preference.

Launch Actions

A *launch action* launches an application or opens or prints a document. Table 8.45 shows the action dictionary entries specific to this type of action.

The optional **Win**, **Mac**, and **Unix** entries allow the action dictionary to include platform-specific parameters for launching the designated application. If no such entry is present for the given platform, the **F** entry is used instead. Table 8.46 shows the platform-specific launch parameters for the Windows platform; those for the Mac OS and UNIX platforms are not yet defined at the time of publication.

TABLE 8.45 Additional entries specific to a launch action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Launch** for a launch action.
F	file specification	*(Required if none of the entries **Win**, **Mac**, or **Unix** is present)* The application to be launched or the document to be opened or printed. If this entry is absent and the viewer application does not understand any of the alternative entries, it should do nothing.
Win	dictionary	*(Optional)* A dictionary containing Windows-specific launch parameters (see Table 8.46; see also implementation note 89 in Appendix H).
Mac	(undefined)	*(Optional)* Mac OS–specific launch parameters; not yet defined.
Unix	(undefined)	*(Optional)* UNIX-specific launch parameters; not yet defined.
NewWindow	boolean	*(Optional; PDF 1.2)* A flag specifying whether to open the destination document in a new window. If this flag is **false**, the destination document will replace the current document in the same window. If this entry is absent, the viewer application should behave in accordance with the current user preference. This entry is ignored if the file designated by the **F** entry is not a PDF document.

TABLE 8.46 Entries in a Windows launch parameter dictionary

KEY	TYPE	VALUE
F	string	*(Required)* The file name of the application to be launched or the document to be opened or printed, in standard Windows pathname format. If the name string includes a backslash character (\\), the backslash must itself be preceded by a backslash. ***Note:*** *This value must be a simple string; it is not a file specification.*
D	string	*(Optional)* A string specifying the default directory in standard DOS syntax.
O	string	*(Optional)* A string specifying the operation to perform: open Open a document. print Print a document. If the **F** entry designates an application instead of a document, this entry is ignored and the application is launched. Default value: open.
P	string	*(Optional)* A parameter string to be passed to the application designated by the **F** entry. This entry should be omitted if **F** designates a document.

Thread Actions

A *thread action* jumps to a specified bead on an article thread (see Section 8.3.2, “Articles”), in either the current document or a different one. Table 8.47 shows the action dictionary entries specific to this type of action.

TABLE 8.47 Additional entries specific to a thread action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Thread** for a thread action.
F	file specification	*(Optional)* The file containing the desired thread. If this entry is absent, the thread is in the current file.

KEY	TYPE	VALUE
D	dictionary, integer, or text string	*(Required)* The desired destination thread, specified in one of the following forms: • An indirect reference to a thread dictionary (see Section 8.3.2, "Articles"). In this case, the thread must be in the current file. • The index of the thread within the **Threads** array of its document's catalog (see Section 3.6.1, "Document Catalog"). The first thread in the array has index 0. • The title of the thread, as specified in its thread information dictionary (see Table 8.7 on page 550). If two or more threads have the same title, the one appearing first in the document catalog's **Threads** array will be used.
B	dictionary or integer	*(Optional)* The desired bead in the destination thread, specified in one of the following forms: • An indirect reference to a bead dictionary (see Section 8.3.2, "Articles"). In this case, the thread must be in the current file. • The index of the bead within its thread. The first bead in a thread has index 0.

URI Actions

A *uniform resource identifier* (URI) is a string that identifies (*resolves* to) a resource on the Internet—typically a file that is the destination of a hypertext link, although it can also resolve to a query or other entity. (URIs are described in Internet RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*; see the Bibliography.)

A *URI action* causes a URI to be resolved. Table 8.48 shows the action dictionary entries specific to this type of action. (See implementation notes 90 and 91 in Appendix H.)

TABLE 8.48 Additional entries specific to a URI action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **URI** for a URI action.
URI	string	*(Required)* The uniform resource identifier to resolve, encoded in 7-bit ASCII.

KEY	TYPE	VALUE
IsMap	boolean	*(Optional)* A flag specifying whether to track the mouse position when the URI is resolved (see below). Default value: **false**. This entry applies only to actions triggered by the user's clicking an annotation; it is ignored for actions associated with outline items or with a document's **OpenAction** entry.

If the **IsMap** flag is **true** and the user has triggered the URI action by clicking an annotation with the mouse, the coordinates of the mouse position at the time the action is performed should be transformed from device space to user space and then offset relative to the upper-left corner of the annotation rectangle (that is, the value of the **Rect** entry in the annotation with which the URI action is associated). For example, if the mouse coordinates in user space are (x_m, y_m) and the annotation rectangle extends from (ll_x, ll_y) at the lower-left to (ur_x, ur_y) at the upper-right, the final coordinates (x_f, y_f) are as follows:

$$x_f = x_m - ll_x$$
$$y_f = ur_y - y_m$$

If the resulting coordinates (x_f, y_f) are fractional, they should be rounded to the nearest integer values. They are then appended to the URI to be resolved, separated by commas and preceded by a question mark. For example:

```
http://www.adobe.com/intro?100,200
```

To support URI actions, a PDF document's catalog (see Section 3.6.1, "Document Catalog") may include a **URI** entry whose value is a *URI dictionary.* At the time of publication, only one entry is defined for such a dictionary (see Table 8.49).

TABLE 8.49 Entry in a URI dictionary

KEY	TYPE	VALUE
Base	string	*(Optional)* The *base URI* to be used in resolving relative URI references. URI actions within the document may specify URIs in partial form, to be interpreted relative to this base address. If no base URI is specified, such partial URIs will be interpreted relative to the location of the document itself. The use of this entry is parallel to that of the body element <BASE>, as described in the *HTML 4.01 Specification* (see the Bibliography).

The **Base** entry allows the URI of the document itself to be recorded in situations in which the document may be accessed out of context. For example, if a document has been moved to a new location but contains relative links to other documents that have not, the **Base** entry could be used to refer such links to the true location of the other documents, rather than that of the moved document.

Sound Actions

A *sound action (PDF 1.2)* plays a sound through the computer's speakers. Table 8.50 shows the action dictionary entries specific to this type of action; sounds themselves are discussed in Section 9.2, "Sounds."

TABLE 8.50 Additional entries specific to a sound action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Sound** for a sound action.
Sound	stream	*(Required)* A sound object defining the sound to be played (see Section 9.2, "Sounds"; see also implementation note 92 in Appendix H).
Volume	number	*(Optional)* The volume at which to play the sound, in the range −1.0 to 1.0; see implementation note 94 in Appendix H. Default value: 1.0.
Synchronous	boolean	*(Optional)* A flag specifying whether to play the sound synchronously or asynchronously; see implementation note 94 in Appendix H. If this flag is **true**, the viewer application will retain control, allowing no further user interaction other than canceling the sound, until the sound has been completely played. Default value: **false**.
Repeat	boolean	*(Optional)* A flag specifying whether to repeat the sound indefinitely. If this entry is present, the **Synchronous** entry is ignored. Default value: **false**.
Mix	boolean	*(Optional)* A flag specifying whether to mix this sound with any other sound already playing; see implementation note 95 in Appendix H. If this flag is **false**, any previously playing sound will be stopped before starting this sound; this can be used to stop a repeating sound (see **Repeat**, above). Default value: **false**.

Movie Actions

A *movie action (PDF 1.2)* can be used to play a movie in a floating window or within the annotation rectangle of a movie annotation (see "Movie Annotations" on page 587 and Section 9.3, "Movies"). The movie annotation must be associated with the page that is the destination of the link annotation or outline item containing the movie action, or with the page object with which the action is associated. (See implementation note 96 in Appendix H.)

Note: *A movie action by itself does not guarantee that the page the movie is on will be displayed before attempting to play the movie; such page change actions must be done explicitly.*

The contents of a movie action dictionary are identical to those of a movie activation dictionary (see Table 9.31 on page 706), with the additional entries shown in Table 8.51. The contents of the activation dictionary associated with the movie annotation provide the default values; any information specified in the movie action dictionary overrides these values.

TABLE 8.51 Additional entries specific to a movie action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Movie** for a movie action.
Annotation	dictionary	*(Optional)* An indirect reference to a movie annotation identifying the movie to be played.
T	text string	*(Optional)* The title of a movie annotation identifying the movie to be played. ***Note:*** *The dictionary must include either an* ***Annotation*** *or a* ***T*** *entry, but not both.*

KEY	TYPE	VALUE
Operation	name	*(Optional)* The operation to be performed on the movie: Play — Start playing the movie, using the play mode specified by the dictionary's **Mode** entry (see Table 9.31 on page 706). If the movie is currently paused, it is repositioned to the beginning before playing (or to the starting point specified by the dictionary's **Start** entry, if present). Stop — Stop playing the movie. PausePause a playing movie. ResumeResume a paused movie. Default value: Play.

Hide Actions

A *hide action (PDF 1.2)* hides or shows one or more annotations on the screen by setting or clearing their Hidden flags (see Section 8.4.2, "Annotation Flags"). This type of action can be used in combination with appearance streams and trigger events (Sections 8.4.4, "Appearance Streams," and 8.5.2, "Trigger Events") to display pop-up help information on the screen. For example, the **E** (enter) and **X** (exit) trigger events in an annotation's additional-actions dictionary can be used to show and hide the annotation when the user rolls the cursor in and out of its active area on the page; this can be used to pop up a help label, or "tool tip," describing the effect of clicking the mouse at that location on the page. Table 8.52 shows the action dictionary entries specific to this type of action. (See implementation notes 97 and 98 in Appendix H.)

TABLE 8.52 Additional entries specific to a hide action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Hide** for a hide action.

KEY	TYPE	VALUE
T	dictionary, string, or array	*(Required)* The annotation or annotations to be hidden or shown, specified in any of the following forms: • An indirect reference to an annotation dictionary • A string giving the fully qualified field name of an interactive form field whose associated widget annotation or annotations are to be affected (see "Field Names" on page 616) • An array of such dictionaries or strings
H	boolean	*(Optional)* A flag indicating whether to hide the annotation (**true**) or show it (**false**). Default value: **true**.

Named Actions

Table 8.53 lists several *named actions (PDF 1.2)* that PDF viewer applications are expected to support; further names may be added in the future. (See implementation notes 99 and 100 in Appendix H.)

TABLE 8.53 Named actions

NAME	ACTION
NextPage	Go to the next page of the document.
PrevPage	Go to the previous page of the document.
FirstPage	Go to the first page of the document.
LastPage	Go to the last page of the document.

Note: *Viewer applications may support additional, nonstandard named actions, but any document using them will not be portable. If the viewer encounters a named action that is inappropriate for a viewing platform, or if the viewer does not recognize the name, it should take no action.*

Table 8.54 shows the action dictionary entries specific to named actions.

TABLE 8.54 Additional entries specific to named actions

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Named** for a named action.
N	name	*(Required)* The name of the action to be performed (see Table 8.53).

Set-OCG-State Actions

A *set-OCG-state action (PDF 1.5)* sets the state of one or more optional content groups (see Section 4.10, "Optional Content"). Table 8.55 shows the action dictionary entries specific to this type of action.

TABLE 8.55 Additional entries specific to a set-OCG-state action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **SetOCGState** for a set-OCG-state action.
State	array	*(Required)* An array consisting of any number of sequences beginning with a name object (**ON**, **OFF**, or **Toggle**) followed by one or more optional content group dictionaries. The array elements are processed from left to right; each name is applied to the subsequent groups until the next name is encountered: • **ON** sets the state of subsequent groups to **ON** • **OFF** sets the state of subsequent groups to **OFF** • **Toggle** reverses the state of subsequent groups.
PreserveRB	boolean	*(Optional)* If **true**, indicates that radio-button state relationships between optional content groups (as specified by the **RBGroups** entry in the current configuration dictionary; see Table 4.48 on page 338) should be preserved when the states in the **State** array are applied. That is, if a group is set to **ON** (either by **ON** or **Toggle**) during processing of the **State** array, any other groups belong to the same radio-button group are turned **OFF**. If a group is set to **OFF**, there is no effect on other groups. If **PreserveRB** is **false**, radio-button state relationships, if any, are ignored. Default value: **true**.

When a set-OCG-state action is performed, the **State** array is processed from left to right. Each name is applied to subsequent groups in the array until the next name is encountered, as shown in the following example.

Example 8.9

```
<< /S /SetOCGState
   /State [/OFF 2 0 R 3 0 R /Toggle 16 0 R 19 0 R /ON 5 0 R]
>>
```

A group can appear more than once in the **State** array; its state is set each time it is encountered, based on the most recent name. For example, if the array contained [/OFF 1 0 R /Toggle 1 0 R], the group's state would be **ON** after the action was performed. There is no required order of **ON**, **OFF** and **Toggle** sequences; more than one sequence in the array may contain the same name.

Note: *While the specification allows a group to appear more than once in the* ***State*** *array, this is not intended to implement animation or any other sequential drawing operations. PDF processing applications are free to accumulate all state changes and apply only the net changes simultaneously to all affected groups before redrawing.*

Rendition Actions

A *rendition action (PDF 1.5)* controls the playing of multimedia content (see Section 9.1, "Multimedia"). It can be used in the following ways:

- To begin the playing of a rendition object (see Section 9.1.2, "Renditions"), associating it with a screen annotation (see "Screen Annotations" on page 588). The screen annotation specifies where the rendition is played unless otherwise specified.
- To stop, pause or resume a playing rendition.
- To trigger the execution of a JavaScript script that may perform custom operations.

Table 8.56 lists the entries in a rendition action dictionary.

TABLE 8.56 Additional entries specific to a rendition action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Rendition** for a rendition action.
R	dictionary	(*Required when **OP** is present with a value of 0 or 4; otherwise optional*) A rendition object (see Section 9.1.2, "Renditions").
AN	dictionary	(*Required if **OP** is present with a value of 0, 1, 2, 3 or 4; otherwise optional*) An indirect reference to a screen annotation (see "Screen Annotations" on page 588).
OP	integer	(*Required if **JS** is not present; otherwise optional*) The operation to perform when the action is triggered. Possible values are: 0 If no rendition is associated with the annotation specified by **AN**, play the rendition specified by **R**, associating it with the annotation. If a rendition is already associated with the annotation, it is stopped, and the new rendition is associated with the annotation. 1 Stop any rendition being played in association with the annotation specified by **AN**, and remove the association. If no rendition is being played, there is no effect. 2 Pause any rendition being played in association with the annotation specified by **AN**. If no rendition is being played, there is no effect. 3 Resume any rendition being played in association with the annotation specified by **AN**. If no rendition is being played or the rendition is not paused, there is no effect. 4 Play the rendition specified by **R**, associating it with the annotation specified by **AN**. If a rendition is already associated with the annotation, resume the rendition if it is paused, otherwise do nothing.
JS	string or stream	(*Required if **OP** is not present; otherwise optional*) A string or stream containing a JavaScript script to be executed when the action is triggered.

Either the **JS** entry or the **OP** entry must be present; if both are present, **OP** is considered a fallback to be executed if the viewer application is unable to execute JavaScripts. If **OP** has an unrecognized value and there is no **JS** entry, the action is invalid.

In some situations, a pause (**OP** value of 2) or resume (**OP** value of 3) operation may not make sense (for example, for a JPEG image) or the player may not support it. In such cases, the user should be notified of the failure to perform the operation.

Before a rendition action is executed, the viewer application must make sure that the **P** entry of the screen annotation dictionary references a valid page object and that the annotation is present in the page object's **Annots** array (see Table 3.27). A rendition may play in the rectangle occupied by a screen annotation, even if the annotation itself is not visible; for example, if its Hidden or NoView flags (see Table 8.12) are set. If a screen annotation is not visible because its location on the page is not being displayed by the viewer, the rendition will not be visible; however, it may become visible if the view changes, such as by scrolling.

Transition Actions

A *transition action (PDF 1.5)* can be used to control drawing during a sequence of actions. As discussed in Section 8.5.1, "Action Dictionaries," the **Next** entry in an action dictionary can specify a sequence of actions. Viewer applications should normally suspend drawing when such a sequence begins and resume drawing when it ends. If a transition action is present during a sequence, the viewer should render the state of the page viewing area as it exists after completion of the previous action, and display it using a transition specified in the action dictionary (see Table 8.57). Once this transition completes, drawing should be suspended again.

TABLE 8.57 Additional entries specific to a transition action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **Trans** for a transition action.
Trans	dictionary	*(Required)* The transition to use for the update of the display (see Table 8.9).

8.6 Interactive Forms

An *interactive form (PDF 1.2)*—sometimes referred to as an *AcroForm*—is a collection of *fields* for gathering information interactively from the user. A PDF document may contain any number of fields appearing on any combination of pages, all of which make up a single, global interactive form spanning the entire document. Arbitrary subsets of these fields can be imported or exported from the document; see Section 8.6.4, "Form Actions."

***Note:** Interactive forms should not be confused with form XObjects (see Section 4.9, "Form XObjects"). Despite the similarity of names, the two are different, unrelated types of object.*

Each field in a document's interactive form is defined by a *field dictionary* (see Section 8.6.2, "Field Dictionaries"). For purposes of definition and naming, the fields can be organized hierarchically and can inherit attributes from their ancestors in the field hierarchy. A field's children in the hierarchy may also include widget annotations (see "Widget Annotations" on page 588) that define its appearance on the page; such a field is called a *terminal field*.

As a convenience, when a field has only a single associated widget annotation, the contents of the field dictionary and the annotation dictionary (Section 8.4.1, "Annotation Dictionaries") may be merged into a single dictionary containing entries that pertain to both a field and an annotation. (This presents no ambiguity, since the contents of the two kinds of dictionary do not conflict.) If such an object defines an appearance stream, the appearance must be consistent with the object's current value as a field.

***Note:** Fields containing text whose contents are not known in advance may need to construct their appearance streams dynamically instead of defining them statically in an appearance dictionary; see "Variable Text" on page 617.*

8.6.1 Interactive Form Dictionary

The contents and properties of a document's interactive form are defined by an *interactive form dictionary* that is referenced from the **AcroForm** entry in the document catalog (see Section 3.6.1, "Document Catalog"). Table 8.58 shows the contents of this dictionary.

TABLE 8.58 Entries in the interactive form dictionary

KEY	TYPE	VALUE
Fields	array	*(Required)* An array of references to the document's *root fields* (those with no ancestors in the field hierarchy).
NeedAppearances	boolean	*(Optional)* A flag specifying whether to construct appearance streams and appearance dictionaries for all widget annotations in the document (see "Variable Text" on page 617). Default value: **false**.
SigFlags	integer	*(Optional; PDF 1.3)* A set of flags specifying various document-level characteristics related to signature fields (see Table 8.59, below, and "Signature Fields" on page 636). Default value: 0.
CO	array	*(Required if any fields in the document have additional-actions dictionaries containing a **C** entry; PDF 1.3)* An array of indirect references to field dictionaries with calculation actions, defining the *calculation order* in which their values will be recalculated when the value of any field changes (see Section 8.5.2, "Trigger Events").
DR	dictionary	*(Optional)* A resource dictionary (see Section 3.7.2, "Resource Dictionaries") containing default resources (such as fonts, patterns, or color spaces) to be used by form field appearance streams. At a minimum, this dictionary must contain a **Font** entry specifying the resource name and font dictionary of the default font for displaying text. (See implementation notes 101 and 102 in Appendix H.)
DA	string	*(Optional)* A document-wide default value for the **DA** attribute of variable text fields (see "Variable Text" on page 617).
Q	integer	*(Optional)* A document-wide default value for the **Q** attribute of variable text fields (see "Variable Text" on page 617).
XFA	stream	*(Optional; PDF 1.5)* A stream containing an *XFA resource* (see Section 8.6.7, "XFA Forms"). The format of an XFA resource is described by the *XML Data Package Specification* (see the Bibliography).

The value of the interactive form dictionary's **SigFlags** entry is an unsigned 32-bit integer containing flags specifying various document-level characteristics related to signature fields (see "Signature Fields" on page 636). Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table 8.59 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

TABLE 8.59 Signature flags

BIT POSITION	NAME	MEANING
1	SignaturesExist	If set, the document contains at least one signature field. This flag allows a viewer application to enable user interface items (such as menu items or pushbuttons) related to signature processing without having to scan the entire document for the presence of signature fields.
2	AppendOnly	If set, the document contains signatures that may be invalidated if the file is saved (written) in a way that alters its previous contents, such as with the "optimize" option. Merely updating the file by appending new information to the end of the previous version is safe (see Section G.6, "Updating Example"). Viewer applications can use this flag to present a user requesting an optimized save with an additional alert box warning that signatures will be invalidated and requiring explicit confirmation before continuing with the operation.

8.6.2 Field Dictionaries

Each field in a document's interactive form is defined by a *field dictionary*, which must be an indirect object. The field dictionaries may be organized hierarchically into one or more tree structures. Many field attributes are *inheritable*, meaning that if they are not explicitly specified for a given field, their values are taken from those of its parent in the field hierarchy. Such inheritable attributes are designated as such in the tables below; the designation *(Required; inheritable)* means that an attribute must be defined for every field, whether explicitly in its own field dictionary or by inheritance from an ancestor in the hierarchy. Table 8.60 shows those entries that are common to all field dictionaries, regardless of type; those that pertain only to a particular type of field are described in the relevant sections below.

TABLE 8.60 Entries common to all field dictionaries

KEY	TYPE	VALUE
FT	name	*(Required for terminal fields; inheritable)* The type of field that this dictionary describes: **Btn** Button (see "Button Fields" on page 625) **Tx** Text (see "Text Fields" on page 631) **Ch** Choice (see "Choice Fields" on page 633) **Sig** *(PDF 1.3)* Signature (see "Signature Fields" on page 636) ***Note:*** *This entry may be present in a nonterminal field (one whose descendants are themselves fields) in order to provide an inheritable* ***FT*** *value. However, a nonterminal field does not logically have a type of its own; it is merely a container for inheritable attributes that are intended for descendant terminal fields of any type.*
Parent	dictionary	*(Required if this field is the child of another in the field hierarchy; absent otherwise)* The field that is the immediate parent of this one (the field, if any, whose **Kids** array includes this field). A field can have at most one parent; that is, it can be included in the **Kids** array of at most one other field.
Kids	array	*(Sometimes required, as described below)* An array of indirect references to the immediate children of this field. In a non-terminal field, the **Kids** array is required to refer to field dictionaries that are immediate descendants of this field. In a terminal field, the **Kids** array ordinarily must refer to one or more separate widget annotations that are associated with this field. However, if there is only one associated widget annotation, and its contents have been merged into the field dictionary, **Kids** must be omitted.
T	text string	*(Optional)* The partial field name (see "Field Names," below; see also implementation notes 103 and 104 in Appendix H).
TU	text string	*(Optional; PDF 1.3)* An alternate field name, to be used in place of the actual field name wherever the field must be identified in the user interface (such as in error or status messages referring to the field). This text is also useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.2, "Alternate Descriptions").
TM	text string	*(Optional; PDF 1.3)* The *mapping name* to be used when exporting interactive form field data from the document.
Ff	integer	*(Optional; inheritable)* A set of flags specifying various characteristics of the field (see Table 8.61). Default value: 0.

KEY	TYPE	VALUE
V	(various)	*(Optional; inheritable)* The field's value, whose format varies depending on the field type; see the descriptions of individual field types for further information.
DV	(various)	*(Optional; inheritable)* The default value to which the field reverts when a reset-form action is executed (see "Reset-Form Actions" on page 643). The format of this value is the same as that of **V**.
AA	dictionary	*(Optional; PDF 1.2)* An additional-actions dictionary defining the field's behavior in response to various trigger events (see Section 8.5.2, "Trigger Events"). This entry has exactly the same meaning as the **AA** entry in an annotation dictionary (see Section 8.4.1, "Annotation Dictionaries").

The value of the field dictionary's **Ff** entry is an unsigned 32-bit integer containing flags specifying various characteristics of the field. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). The flags shown in Table 8.61 are common to all types of field; flags that apply only to specific field types are discussed in the sections describing those types. All undefined flag bits are reserved and must be set to 0.

TABLE 8.61 Field flags common to all field types

BIT POSITION	NAME	MEANING
1	ReadOnly	If set, the user may not change the value of the field. Any associated widget annotations will not interact with the user; that is, they will not respond to mouse clicks or change their appearance in response to mouse motions. This flag is useful for fields whose values are computed or imported from a database.
2	Required	If set, the field must have a value at the time it is exported by a submit-form action (see "Submit-Form Actions" on page 639).
3	NoExport	If set, the field must not be exported by a submit-form action (see "Submit-Form Actions" on page 639).

Field Names

The **T** entry in the field dictionary (see Table 8.60 on page 615) holds a text string defining the field's *partial field name*. The *fully qualified field name* is not explicitly defined, but is constructed from the partial field names of the field and all of its ancestors. For a field with no parent, the partial and fully qualified names are the same; for a field that is the child of another field, the fully qualified name is formed by appending the child field's partial name to the parent's fully qualified name, separated by a period (.):

parent's_full_name.child's_partial_name

For example, if a field with the partial field name PersonalData has a child whose partial name is Address, which in turn has a child with the partial name ZipCode, then the fully qualified name of this last field would be

PersonalData.Address.ZipCode

Thus all fields descended from a common ancestor will share the ancestor's fully qualified field name as a common prefix in their own fully qualified names.

It is possible for different field dictionaries to have the same fully qualified field name if they are descendants of a common ancestor with that name and have no partial field names (**T** entries) of their own. Such field dictionaries are different representations of the same underlying field; they should differ only in properties that specify their visual appearance. In particular, field dictionaries with the same fully qualified field name must have the same field type (**FT**), value (**V**), and default value (**DV**).

Variable Text

When the contents and properties of a field are known in advance, its visual appearance can be specified by an appearance stream defined in the PDF file itself (see Section 8.4.4, "Appearance Streams," and "Widget Annotations" on page 588). In some cases, however, the field may contain text whose value is not known until viewing time. Examples include text fields to be filled in with text typed by the user from the keyboard and scrollable list boxes whose contents are determined interactively at the time the document is displayed.

In such cases, the PDF document cannot provide a statically defined appearance stream for displaying the field; rather, the viewer application must construct an appearance stream dynamically at viewing time. The dictionary entries shown in Table 8.62 provide general information about the field's appearance that can be combined with the specific text it contains to construct an appearance stream.

TABLE 8.62 Additional entries common to all fields containing variable text

KEY	TYPE	VALUE
DA	string	*(Required; inheritable)* The *default appearance string*, containing a sequence of valid page-content graphics or text state operators defining such properties as the field's text size and color.
Q	integer	*(Optional; inheritable)* A code specifying the form of *quadding* (justification) to be used in displaying the text: 0 Left-justified 1 Centered 2 Right-justified Default value: 0 (left-justified).
DS	text string	*(Optional; PDF 1.5)* A default style string, as described in "Rich Text Strings" on page 620.
RV	text string or text stream	*(Optional; PDF 1.5)* A rich text string, as described in "Rich Text Strings" on page 620.

The new appearance stream becomes the normal appearance (**N**) in the appearance dictionary associated with the field's widget annotation (see Table 8.15 on page 569). (If the widget annotation has no appearance dictionary, the viewer application must create one and store it in the annotation dictionary's **AP** entry.)

In PDF 1.5, form fields that have the RichText flag set (see Table 8.68) specify formatting information as described in "Rich Text Strings" on page 620. For these fields, the conventions described below are not used, and the entire annotation appearance is regenerated each time the value is changed.

For non-rich text fields, the appearance stream—which, like all appearance streams, is a form XObject—has the contents of its form dictionary initialized as follows:

- The resource dictionary (**Resources**) is created using resources from the interactive form dictionary's **DR** entry (see Table 8.58); see also implementation note 105 in Appendix H.
- The lower-left corner of the bounding box (**BBox**) is set to coordinates (0, 0) in the form coordinate system. The box's top and right coordinates are taken from

the dimensions of the annotation rectangle (the **Rect** entry in the widget annotation dictionary).

- All other entries in the appearance stream's form dictionary are set to their default values (see Section 4.9, "Form XObjects").

The appearance stream includes the following section of marked content, which represents the portion of the stream that draws the text:

Example 8.10

```
/Tx  BMC                                                   % Begin marked content with tag Tx
     q                                                     % Save graphics state
          … Any required graphics state changes, such as clipping …
          BT                                               % Begin text object
               … Default appearance string (DA) …
               … Text-positioning and text-showing operators to show the variable text …
          ET                                               % End text object
     Q                                                     % Restore graphics state
EMC                                                        % End marked content
```

The **BMC** (begin marked content) and **EMC** (end marked content) operators are discussed in Section 10.5, "Marked Content"; **q** (save graphics state) and **Q** (restore graphics state) in Section 4.3.3, "Graphics State Operators"; and **BT** (begin text object) and **ET** (end text object) in Section 5.3, "Text Objects." See Example 8.14 on page 633 for an example.

The default appearance string (**DA**) contains any graphics state or text state operators needed to establish the graphics state parameters, such as text size and color, for displaying the field's variable text. Only operators that are allowed within text objects may occur in this string (see Figure 4.1 on page 167). At a minimum, the string must include a **Tf** (text font) operator along with its two operands, *font* and *size*. The specified *font* value must match a resource name in the **Font** entry of the default resource dictionary (referenced from the **DR** entry of the interactive form dictionary; see Table 8.58). A zero value for *size* means that the font is to be *auto-sized*: its size is computed as a function of the height of the annotation rectangle.

The default appearance string should contain at most one **Tm** (text matrix) operator. If this operator is present, the viewer application should replace the horizontal and vertical translation components with positioning values it determines to be appropriate, based on the field value, the quadding (**Q**) attribute, and any layout rules it employs. If the default appearance string contains no **Tm** operator, the viewer should insert one in the appearance stream, with appropriate horizontal and vertical translation components, after the default appearance string and before the text-positioning and text-showing operators for the variable text.

To update an existing appearance stream to reflect a new field value, the viewer application should first copy any needed resources from the document's **DR** dictionary (see Table 8.58) into the stream's **Resources** dictionary. (If the **DR** and **Resources** dictionaries contain resources with the same name, the one already in the **Resources** dictionary should be left intact, *not* replaced with the corresponding value from the **DR** dictionary.) The viewer application should then replace the existing contents of the appearance stream from

```
/Tx  BMC
```

to the matching

```
EMC
```

with the corresponding new contents as shown in Example 8.10. (If the existing appearance stream contains no marked content with tag Tx, the new contents should be appended to the end of the original stream.) Also see implementation note 106 in Appendix H.

Rich Text Strings

Beginning with PDF 1.5, the text contents of variable text form fields, as well as markup annotations, can include formatting (style) information. These *rich text strings* are fully-formed XML documents that conform to a subset of the XFA Text Specification, which is itself a subset of the XHTML 1.0 specification, augmented with a restricted set of CSS2 style attributes (see the Bibliography for references to all these standards). This section describes the basic elements of this specification.

Table 8.63 lists the XHTML elements that are supported in rich text strings. The <body> element is the root element; its required attributes are listed in Table 8.64. Other elements (<p> and <span>) contain enclosed text that may take style attributes, which are listed in Table 8.65. These style attributes are CSS inline style property declarations of the form *name*:*value*, with each declaration separated by a semi-colon, as illustrated in Example 8.11 on page 624.

TABLE 8.63 XHTML Elements used in rich text strings

ELEMENT	DESCRIPTION
<body>	The element at the root of the XML document. Table 8.64 lists the required attributes for this element.
<p>	Encloses text that is interpreted as a paragraph. It may take the style attributes listed in Table 8.65.
<i>	Encloses text that is displayed in an italic font.
<b>	Encloses text that is displayed in a bold font.
<span>	Groups text solely for the purpose of applying styles (using the attributes in Table 8.65).

TABLE 8.64 Attributes of the <body> element

ATTRIBUTE	DESCRIPTION
xmlns	The default namespaces for elements within the rich text string. Must be xmlns="http://www.w3.org/1999/xhtml" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0".
xfa:contentType	Must be "text/html".
xfa:APIVersion	A string that identifies the software used to generate the rich text string. It must be of the form software_name:software_version, where: • software_name identifies the software by name. It must not contain spaces. • software_version identifies the version of the software. It consists of a series of integers separated by decimal points; each integer is a version number, the leftmost value being a major version number, with values to the right increasingly minor. When comparing strings, the versions are compared in order. For example "5.2" is less than "5.13" because 2 is less than 13; the string is not treated as a decimal number. When comparing strings with different numbers of sections, the string with fewer sections is implicitly padded on the right with sections containing "0" to make the number of sections equivalent.
xfa:spec	The version of the XFA specification to which the rich text string complies. For PDF 1.5, versions 2.02 and lower are supported.

TABLE 8.65 CSS2 style attributes used in rich text strings

ATTRIBUTE	VALUE	DESCRIPTION
text-align	keyword	Horizontal alignment. Possible values: left, right, center.
vertical-align	decimal	An amount by which to adjust the baseline of the enclosed text. A positive value indicates a superscript; a negative value indicates a subscript. The value is of the form <*decimal number*>pt, optionally preceded by a sign, and followed by "pt". Examples: -3pt, 4pt.
font-size	decimal	The font size of the enclosed text. The value is of the form <*decimal number*>pt.
font-style	keyword	Specifies whether the enclosed text should be displayed using a normal or italic (oblique) font. Possible values: normal, italic.
font-weight	keyword	The weight of the font for the enclosed text. Possible values: normal, bold, 100, 200, 300, 400, 500, 600, 700, 800, 900. ***Note:*** *normal is equivalent to 400, and bold is equivalent to 700.*
font-family	list	A font name or list of font names to be used to display the enclosed text. (If a list is provided, the first one containing glyphs for the specified text is used.)
font	list	A shorthand CSS font property of the form font:<*font-style*> <*font-weight*> <*font-size*> <*font-family*>
color	RGB value	The color of the enclosed text. The value is an RGB value specified in the sRGB color space (<http://www.srgb.com>). It can be in one of two forms: • #*rrggbb*, with a 2-digit hexadecimal value for each component • rgb*(rrr,ggg,bbb)* with a decimal value for each component. ***Note:*** *Although the values specified by the color property are interpreted as sRGB values, they are transformed into values in a non-ICC based color space when used to generate the annotation's appearance.*
text-decoration	keyword	One of the following keywords: • underline: the enclosed text should be underlined. • line-through: the enclosed text should have a line drawn through it.
font-stretch	keyword	Specifies a normal, condensed or extended face from a font family. Supported values from narrowest to widest are ultra-condensed, extra-condensed, condensed, semi-condensed, normal, semi-expanded, expanded, extra-expanded and ultra-expanded.

Rich text strings are specified by the **RV** entry of variable text form field dictionaries (see Table 8.62) and the **RC** entry of markup annotation dictionaries (see Table 8.17). Rich text strings may be packaged as *text streams* (see Section 3.8.2, “Text Streams”). Form fields using rich text streams should also have the RichText flag set (see Table 8.68).

A *default style string* is specified by the **DS** entry for free text annotations (see Table 8.21) or variable text form fields (see Table 8.62). This string specifies the default values for style attributes, which are used for any style attributes that are not explicitly specified for the annotation or field. All attributes listed in Table 8.65 are legal in the default style string. This string, in addition to the **RV** or **RC** entry, is used to generate the appearance. The following entries are ignored by PDF 1.5-compliant viewers: the **Contents** entry for annotations, the **DA** entry for free text annotations, and the **V**, **DA** and **Q** entries for form fields.

***Note:** Markup annotations other than free text annotations (see “Markup Annotations” on page 571) do not use a default style string, because their appearances are implemented using platform controls requiring the viewer application to pick an appropriate system font for display.*

When a form field or annotation contains rich text strings, the *flat text* (character data) of the string should also be preserved (in the **V** entry for form fields and the **Contents** entry for annotations). This enables older viewer applications to read and edit the data (although with loss of formatting information). The **DA** entry should be written out as well, when the file is saved.

If a document containing rich text strings is edited in a viewer that does not support PDF 1.5, the rich text strings remain unchanged (because they are unknown to the viewer), even though the corresponding flat text may have changed. When a viewer that supports PDF 1.5 reads a rich text string from a document, it must check whether the corresponding flat text has changed, using the following procedure:

1. Create a new flat text string containing the character data from the rich text string. Character references (such as &#13;) should be converted to their character equivalents.

   ***Note:** No attempt should be made to preserve formatting specified with markup elements. For example, although the <p> element implies a new line, a carriage return should not be generated in the associated flat text.*

2. If either of the values uses UTF-16 encoding, promote the other value to UTF-16 if necessary.
3. Compare the resulting strings.

If the strings are unequal, it is assumed the field has been modified by an older viewer, and a new rich text string should be created from the flat text.

When a rich text string specifies font attributes, the viewer application should use font name selection as described in section 15.3 of the CSS2 specification (see the Bibliography). It is strongly recommended that precedence be given to the fonts in the default resources dictionary, as specified by the **DR** entry in Table 8.58; see Implementation note 107 in Appendix H.

The following example illustrates the entries in a widget annotation dictionary for rich text. The **DS** entry specifies the default font. The **RV** entry contains two paragraphs of rich text: the first paragraph specifies bold and italic text, using the default font; the second paragraph changes the font size.

Example 8.11

```
/DS (font: 18pt Arial)                % Default style string using an abbreviated font
                                      % descriptor to specify 18pt text using an Arial font

/RV (<?xml version="1.0"?><body xmlns="http://www.w3.org/1999/xtml"
        xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
        xfa:contentType="text/html" xfa:APIVersion="Acrobat:6.0.0" xfa:spec="2.0.2">
        <p style="text-align:left">
            <b>
                <i>
                    Here is some bold italic text
                </i>
            </b>
        </p>
        <p style= "font-size:16pt">
            This text uses default text state parameters but changes the font size to 16.
        </p>
    </body> )
```

8.6.3 Field Types

Interactive forms support the following field types:

- *Button fields* represent interactive controls on the screen that the user can manipulate with the mouse. They include *pushbuttons*, *checkboxes*, and *radio buttons*.
- *Text fields* are boxes or spaces in which the user can enter text from the keyboard.
- *Choice fields* contain several text items, at most one of which may be selected as the field value. They include scrollable *list boxes* and *combo boxes*.
- *Signature fields* represent electronic "signatures" for authenticating the identity of a user and the validity of the document's contents.

The following sections describe each of these field types in detail. Further types may be added in the future.

Button Fields

A *button field* (field type **Btn**) represents an interactive control on the screen that the user can manipulate with the mouse. It may be any of the following:

- A *pushbutton* is a purely interactive control that responds immediately to user input without retaining a permanent value.
- A *checkbox* toggles between two states, on and off.
- *Radio button fields* contain a set of related buttons that can each be on or off. Typically, at most one radio button in a set may be on at any given time; selecting any one of the buttons automatically deselects all the others. (There are exceptions to this, as noted in "Radio Buttons" on page 628.)

The various types of button field are distinguished by flags in the **Ff** entry, as shown in Table 8.66.

TABLE 8.66 Field flags specific to button fields

BIT POSITION	NAME	MEANING
15	NoToggleToOff	*(Radio buttons only)* If set, exactly one radio button must be selected at all times; clicking the currently selected button has no effect. If clear, clicking the selected button deselects it, leaving no button selected.
16	Radio	If set, the field is a set of radio buttons; if clear, the field is a checkbox. This flag is meaningful only if the Pushbutton flag is clear.
17	Pushbutton	If set, the field is a pushbutton that does not retain a permanent value.
26	RadiosInUnison	*(PDF 1.5)* If set, a group of radio buttons within a radio button field that use the same value for the on state will turn on and off in unison; that is if one is checked, they are all checked. If clear, the buttons are mutually exclusive (the same behavior as HTML radio buttons).

Pushbuttons

The simplest type of field is a *pushbutton field*, which has a field type of **Btn** and the Pushbutton flag (see Table 8.66) set. Because this type of button retains no permanent value, it does not use the **V** and **DV** entries in the field dictionary (see Table 8.60 on page 615).

Checkboxes

A *checkbox field* represents one or more checkboxes that toggle between two states, on and off, when manipulated by the user with the mouse or keyboard. Its field type is **Btn** and its Pushbutton and Radio flags (see Table 8.66) are both clear. Each state can have a separate appearance, defined by an appearance stream in the appearance dictionary of the field's widget annotation (see Section 8.4.4, "Appearance Streams"). The appearance for the off state is optional, but if present must be stored in the appearance dictionary under the name Off. The recommended name for the on state is Yes, but this is not required.

The **V** entry in the field dictionary (see Table 8.60 on page 615) holds a name object representing the checkbox's appearance state, which is used to select the appropriate appearance from the appearance dictionary.

Example 8.12 shows a typical checkbox definition.

Example 8.12

```
1 0 obj
    << /FT /Btn
        /T (Urgent)
        /V /Yes
        /AS /Yes
        /AP << /N << /Yes 2 0 R /Off 3 0 R>>
    >>
endobj

2 0 obj
    << /Resources 20 0 R
        /Length 104
    >>

stream
    q
        0 0 1 rg
        BT
            /ZaDb 12 Tf
            0 0 Td
            (8) Tj
        ET
    Q
endstream
endobj

3 0 obj
    << /Resources 20 0 R
        /Length 104
    >>

stream
    q
        0 0 1 rg
        BT
            /ZaDb 12 Tf
            0 0 Td
            (8) Tj
        ET
    Q
endstream
endobj
```

Beginning with PDF 1.4, the field dictionary for checkboxes and radio buttons contains an optional **Opt** entry (see Table 8.67), which holds an array of text strings representing the export value of each annotation in the field. It is used for the following purposes:

- To represent the export values of checkbox and radio button fields in non-Latin writing systems. Because name objects in the appearance dictionary are limited to **PDFDocEncoding**, they cannot represent non-Latin text.

- To allow radio buttons or checkboxes to be checked independently, even if they have the same export value.

  An example of this is a group of checkboxes that are duplicated on more than one page, and the desired behavior is that when a user checks a box, the corresponding boxes on each of the other pages is also checked. In this case, each of the corresponding checkboxes is a widget in the **Kids** array of a checkbox field.

  Note: *For radio buttons, the same behavior occurs only if the RadiosInUnison flag is set. If it is not set, at most one radio button in a field can be set at a time. See implementation note 108 in Appendix H.*

TABLE 8.67 Additional entry specific to checkbox and radio button fields

KEY	TYPE	VALUE
Opt	array of text strings	*(Optional; inheritable; PDF 1.4)* An array containing one entry for each widget annotation in the in the **Kids** array of the radio button or checkbox field. Each entry is a text string representing the on state of the corresponding widget annotation. When this entry is present, the names used to represent the on state in the **AP** dictionary of each annotation are computer-generated numbers equivalent to the numerical position (starting with **0**) of the annotation in the **Kids** array. This allows distinguishing between the annotations even if two or more of them have the same value in the **Opt** array. For example, two radio buttons may have the same on state, but if the RadiosInUnison flag is not set, only one of them at a time can be checked by the user.

Radio Buttons

A *radio button field* is a set of related buttons. Like checkboxes, individual radio buttons have two states, on and off. A single radio button may not be turned off directly, but only as a result of another button being turned on. Typically, a set of radio buttons (annotations that are children of a single radio button field) have at most one button in the on state at any given time; selecting any of the buttons automatically deselects all the others.

***Note:** An exception occurs when multiple radio buttons in a field have the same "on" state and the RadiosInUnison flag is set; in that case, turning on one of those buttons turns on all the others.*

The field type is **Btn**, the Pushbutton flag (see Table 8.66 on page 626) is clear, and the Radio flag is set. This type of button field has an additional flag, NoToggleToOff, which specifies, if set, that exactly one of the radio buttons must be selected at all times. In this case, clicking the currently selected button has no effect; if the NoToggleToOff flag is clear, clicking the selected button deselects it, leaving no button selected.

The **Kids** entry in the radio button field's field dictionary (see Table 8.60 on page 615) holds an array of widget annotations representing the individual buttons in the set. The parent field's **V** entry holds a name object corresponding to the appearance state of whichever child field is currently in the on state; the default value for this entry is Off. Example 8.13 shows the object definitions for a set of radio buttons.

Example 8.13

```
10 0 obj                                          % Radio button field
   << /FT /Btn
      /Ff …                                       % …Radio flag = 1, Pushbutton = 0…
      /T (Credit card)
      /V /MasterCard
      /Kids [ 11 0 R
              12 0 R
            ]
   >>
endobj

11 0 obj                                          % First radio button
   << /Parent 10 0 R
      /AS /MasterCard
      /AP << /N << /MasterCard 8 0 R
                   /Off 9 0 R
                >>
          >>
   >>
endobj
```

```
12 0 obj                                          % Second radio button
   << /Parent 10 0 R
      /AS /Off
      /AP << /N << /Visa 8 0 R
                   /Off 9 0 R
                >>
          >>
   >>
endobj

8 0 obj                                           % Appearance stream for "on" state
   << /Resources 20 0 R
      /Length 104
   >>
stream
   q
      0 0 1 rg
      BT
         /ZaDb 12 Tf
         0 0 Td
         (8) Tj
      ET
   Q
endstream
endobj

9 0 obj                                           % Appearance stream for "off" state
   << /Resources 20 0 R
      /Length 104
   >>
stream
   q
      0 0 1 rg
      BT
         /ZaDb 12 Tf
         0 0 Td
         (4) Tj
      ET
   Q
endstream
endobj
```

Like a checkbox field, a radio button field can use the optional **Opt** entry in the field dictionary *(PDF 1.4)* to define export values for its constituent radio buttons using Unicode encoding for non-Latin characters (see Table 8.67). **Opt** holds an array of text strings corresponding to the widget annotations representing the individual buttons in the field's **Kids** array.

Text Fields

A *text field* (field type **Tx**) is a box or space in which the user can enter text from the keyboard. The text may be restricted to a single line or may be allowed to span multiple lines, depending on the setting of the Multiline flag in the field dictionary's **Ff** entry. Table 8.68 shows the flags pertaining to this type of field.

TABLE 8.68 Field flags specific to text fields

BIT POSITION	NAME	MEANING
13	Multiline	If set, the field may contain multiple lines of text; if clear, the field's text is restricted to a single line.
14	Password	If set, the field is intended for entering a secure password that should not be echoed visibly to the screen. Characters typed from the keyboard should instead be echoed in some unreadable form, such as asterisks or bullet characters. To protect password confidentiality, viewer applications should never store the value of the text field in the PDF file if this flag is set.
21	FileSelect	*(PDF 1.4)* If set, the text entered in the field represents the pathname of a file whose contents are to be submitted as the value of the field.
23	DoNotSpellCheck	*(PDF 1.4)* If set, the text entered in the field will not be spell-checked.
24	DoNotScroll	*(PDF 1.4)* If set, the field will not scroll (horizontally for single-line fields, vertically for multiple-line fields) to accommodate more text than will fit within its annotation rectangle. Once the field is full, no further text will be accepted.
25	Comb	*(PDF 1.5)* Meaningful only if the **MaxLen** entry is present in the text field dictionary (see Table 8.69) and if the Multiline, Password, and FileSelect flags are clear. If set, the field is automatically divided up into as many equally spaced positions, or *combs*, as the value of **MaxLen**, and the text is laid out into those combs.

BIT POSITION	NAME	MEANING
28	RichText	*(PDF 1.5)* If set, the value of this field should be represented as a rich text string (see “Rich Text Strings” on page 620). If the field has a value, the **RV** entry of the field dictionary (see Table 8.62) specifies the rich text string.

The field’s text is held in a text string or, starting with PDF 1.5, a stream, in the **V** (value) entry of the field dictionary. The contents of this text string or stream are used to construct an appearance stream for displaying the field, as described under “Variable Text” on page 617; the text is presented in a single style (font, size, color, and so forth), as specified by the **DA** (default appearance) string.

If the FileSelect flag *(PDF 1.4)* is set, the field functions as a *file-select control*. In this case, the field’s text represents the pathname of a file whose contents are to be submitted as the field’s value:

- For fields submitted in HTML Form format, the submission uses the MIME content type multipart/form-data, as described in Internet RFC 2045, *Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies* (see the Bibliography).
- For Forms Data Format (FDF) submission, the value of the **V** entry in the FDF field dictionary (see “FDF Fields” on page 653) is a file specification (Section 3.10, “File Specifications”) identifying the selected file.
- XML format is not supported for file-select controls, so no value is submitted in this case.

Besides the usual entries common to all fields (see Table 8.60 on page 615) and to fields containing variable text (see Table 8.62), the field dictionary for a text field can contain the additional entry shown in Table 8.69.

TABLE 8.69 Additional entry specific to a text field

KEY	TYPE	VALUE
MaxLen	integer	*(Optional; inheritable)* The maximum length of the field’s text, in characters.

Example 8.14 shows the object definitions for a typical text field.

Example 8.14

```
6 0 obj
    << /FT /Tx
        /Ff …                                          % Set Multiline flag
        /T (Silly prose)
        /DA (0 0 1 rg /Ti 12 Tf)
        /V (The quick brown fox ate the lazy mouse)
        /AP << /N 5 0 R >>
    >>
endobj

5 0 obj
    << /Resources 21 0 R
        /Length 172
    >>
stream
    /Tx BMC
        q
            BT
                0 0 1 rg
                /Ti 12 Tf
                1 0 0 1 100 100 Tm
                0 0 Td
                (The quick brown fox ) Tj
                0 −13 Td
                (ate the lazy mouse.) Tj
            ET
        Q
    EMC
endstream
endobj
```

Choice Fields

A *choice field* (field type **Ch**) contains several text items, one or more of which may be selected as the field value. The items may be presented to the user in either of two forms:

- A scrollable *list box*
- A *combo box* consisting of a drop list optionally accompanied by an editable text box in which the user can type a value other than the predefined choices

TABLE 8.70 Field flags specific to choice fields

BIT POSITION	NAME	MEANING
18	Combo	If set, the field is a combo box; if clear, the field is a list box.
19	Edit	If set, the combo box includes an editable text box as well as a drop list; if clear, it includes only a drop list. This flag is meaningful only if the Combo flag is set.
20	Sort	If set, the field's option items should be sorted alphabetically. This flag is intended for use by form authoring tools, not by PDF viewer applications; viewers should simply display the options in the order in which they occur in the **Opt** array (see Table 8.71).
22	MultiSelect	*(PDF 1.4)* If set, more than one of the field's option items may be selected simultaneously; if clear, no more than one item at a time may be selected.
23	DoNotSpellCheck	*(PDF 1.4)* If set, the text entered in the field will not be spell-checked. This flag is meaningful only if the Combo and Edit flags are both set.
27	CommitOnSelChange	*(PDF 1.5)* If set, the new value is committed as soon as a selection is made with the pointing device. This allows applications to perform an action once a selection is made, without requiring the user to exit the field. If clear, the new value is not committed until the user exits the field.

The various types of choice field are distinguished by flags in the **Ff** entry, as shown in Table 8.70. Table 8.71 shows the field dictionary entries specific to choice fields.

TABLE 8.71 Additional entries specific to a choice field

KEY	TYPE	VALUE
Opt	array	*(Optional)* An array of options to be presented to the user. Each element of the array is either a text string representing one of the available options or an array consisting of two text strings: the option's export value and the text to be displayed as the name of the option (see implementation note 109 in Appendix H). If this entry is not present, no choices should be presented to the user.
TI	integer	*(Optional)* For scrollable list boxes, the *top index* (the index in the **Opt** array of the first option visible in the list). Default value: 0.

KEY	TYPE	VALUE
I	array	*(Sometimes required, otherwise optional; PDF 1.4)* For choice fields that allow multiple selection (MultiSelect flag set), an array of integers, sorted in ascending order, representing the zero-based indices in the **Opt** array of the currently selected option items. This entry is required when two or more elements in the **Opt** array have different names but the same export value, or when the value of the choice field is an array; in other cases, it is permitted but not required. If the items identified by this entry differ from those in the **V** entry of the field dictionary (see below), the **V** entry takes precedence.

The **Opt** array specifies the list of options in the choice field, each of which is represented by a text string to be displayed on the screen. Each element of the **Opt** array contains either this text string by itself or a two-element array, whose second element is the text string and whose first element is a text string representing the export value to be used when exporting interactive form field data from the document.

The field dictionary's **V** (value) entry (see Table 8.60 on page 615) identifies the item or items currently selected in the choice field. If the field does not allow multiple selection—that is, if the MultiSelect flag *(PDF 1.4)* is not set—or if multiple selection is supported but only one item is currently selected, **V** is a text string representing the name of the selected item, as given in the field dictionary's **Opt** array; if multiple items are selected, it is an array of such strings. (For items represented in the **Opt** array by a two-element array, the name string is the second of the two array elements.) The default value of **V** is **null**, indicating that no item is currently selected.

Example 8.15 shows a typical choice field definition.

Example 8.15

```
<< /FT /Ch
   /Ff …
   /T (Body Color)
   /V (Blue)
   /Opt [ (Red)
          (My favorite color)
          (Blue)
        ]
>>
```

Signature Fields

A *signature field (PDF 1.3)* is a form field that contains a digital signature (see Section 8.7, "Digital Signatures"). The field dictionary representing a signature field may contain the additional entries listed in Table 8.72, as well as the standard entries described in Table 8.60. The field type (**FT**) is **Sig**, and the field value (**V**) is a *signature dictionary* containing the signature and specifying various attributes of the signature field (see Table 8.93).

Filling in ("signing") the signature field entails updating at least the **V** entry, and usually also the **AP** entry of the associated widget annotation. Exporting a signature field typically exports the **T**, **V**, and **AP** entries.

Like any other field, a signature field may actually be described by a widget annotation dictionary containing entries pertaining to an annotation as well as a field (see "Widget Annotations" on page 588). The annotation rectangle (**Rect**) in such a dictionary gives the position of the field on its page. Signature fields that are not intended to be visible, should have an annotation rectangle that has zero height and width.

The appearance dictionary (**AP**) of a signature field's widget annotation defines the field's visual appearance on the page (see Section 8.4.4, "Appearance Streams"). Information about how Acrobat handles digital signature appearances can be found in the technical note *Digital Signature Appearances* (see the Bibliography).

TABLE 8.72 Additional entries specific to a signature field

KEY	TYPE	VALUE
Lock	dictionary	*(Optional; must be an indirect reference; PDF 1.5)* A *signature field lock dictionary* that specifies a set of form fields to be locked when this signature field is signed. Table 8.73 lists the entries in this dictionary.
SV	dictionary	*(Optional; must be an indirect reference; PDF 1.5)* A *seed value dictionary* (see Table 8.74) containing information that constrains the properties of a signature that is applied to this field.

The value of the **SV** entry in the field dictionary is a seed value dictionary whose entries (see Table 8.74) provide constraining information for the corresponding entries in the signature dictionary. Its **Ff** entry specifies whether the other entries are required to be honored or whether they are merely recommendations.

Note: *The seed value dictionary may include seed values for private entries belonging to multiple handlers; a given handler should use only those entries that are pertinent to itself and strip out the others.*

TABLE 8.73 Entries in a signature field lock dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SigFieldLock** for a signature field lock dictionary.
Action	name	*(Required)* A name which, in conjunction with **Fields**, indicates the set of fields that should be locked. Possible values are: All All fields in the document IncludeAll fields specified in **Fields** ExcludeAll fields except those specified in **Fields**
Fields	array	*(Required if the value of* ***Action*** *is Include or Exclude)* An array of strings containing field names.

TABLE 8.74 Entries in a signature field seed value dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SV** for a seed value dictionary.
Filter	name	*(Optional)* The signature handler to be used to sign the signature field.
SubFilter	array	*(Optional)* An array of names indicating acceptable encodings to use when signing. The first name in the array that matches an encoding supported by the signature handler will be the encoding actually used for signing.
V	integer	*(Optional)* The minimum required version number of the signature handler to be used to sign the signature field.
Cert	dictionary	*(Optional)* A *certificate seed value dictionary* (see Table 8.75) containing information about the certificate to be used when signing.

KEY	TYPE	VALUE
Reasons	array	*(Optional)* An array of strings that specifying possible reasons for signing a document.
Ff	integer	*(Optional)* A set of bit flags specifying the interpretation of specific entries in this dictionary. A value of 1 for the flag means that a signer is required to use only the specified values for the entry; a value of 0 means that other values are permissible. Bit positions are 1 (**Filter**); 2 (**SubFilter**); 3 (**V**); 4 (**Reasons**). Default value: 0.

TABLE 8.75 Entries in a certificate seed value dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SVCert** for a certificate seed value dictionary.
Subject	array	*(Optional)* An array of strings containing DER-encoded X.509v3 certificates that are acceptable for signing. X.509v3 certificates are described in RFC 3280, *Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile* (see the Bibliography).
Issuer	array	*(Optional)* An array of strings containing DER-encoded X.509v3 certificates of acceptable issuer for certificates that can be used for signing.
OID	array	*(Optional)* An array of strings that contain Object Identifiers (OIDs) of the certificate policies that must be present in the signing certificate. This field is only applicable if the value of **Issuer** is not empty. The certificate policies extension is described in RFC 3280 (see the Bibliography).
URL	string	*(Optional)* A URL that can be used to enroll for a new credential if a matching credential is not found.
Ff	integer	*(Optional)* A set of bit flags specifying the interpretation of specific entries in this dictionary. A value of 1 for the flag means that a signer is required to use only the specified values for the entry; a value of 0 means that other values are permissible. Bit positions are 1 (**Subject**); 2 (**Issuer**); 3 (**OID**). Default value: 0.

8.6.4 Form Actions

Interactive forms support four special types of action in addition to those described in Section 8.5.3, "Action Types":

- *Submit-form actions* transmit the names and values of selected interactive form fields to a specified uniform resource locator (URL), presumably the address of a World Wide Web server that will process them and send back a response.
- *Reset-form actions* reset selected interactive form fields to their default values.
- *Import-data actions* import Forms Data Format (FDF) data into the document's interactive form from a specified file.
- *JavaScript actions (PDF 1.3)* cause a script to be compiled and executed by the JavaScript interpreter.

Submit-Form Actions

A *submit-form action* transmits the names and values of selected interactive form fields to a specified uniform resource locator (URL), presumably the address of a World Wide Web server that will process them and send back a response. Table 8.76 shows the action dictionary entries specific to this type of action.

The value of the action dictionary's **Flags** entry is an unsigned 32-bit integer containing flags specifying various characteristics of the action. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table 8.77 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

TABLE 8.76 Additional entries specific to a submit-form action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **SubmitForm** for a submit-form action.
F	file specification	*(Required)* A URL file specification (see Section 3.10.4, "URL Specifications") giving the uniform resource locator (URL) of the script at the Web server that will process the submission.

KEY	TYPE	VALUE
Fields	array	*(Optional)* An array identifying which fields to include in the submission or which to exclude, depending on the setting of the Include/Exclude flag in the **Flags** entry (see Table 8.77). Each element of the array is either an indirect reference to a field dictionary or *(PDF 1.3)* a string representing the fully qualified name of a field. Elements of both kinds may be mixed in the same array. If this entry is omitted, the Include/Exclude flag is ignored; all fields in the document's interactive form are submitted except those whose NoExport flag (see Table 8.61 on page 616) is set. (Fields with no values may also be excluded, depending on the setting of the IncludeNoValueFields flag; see Table 8.77.) See the text following Table 8.77 for further discussion.
Flags	integer	*(Optional; inheritable)* A set of flags specifying various characteristics of the action (see Table 8.77). Default value: 0.

TABLE 8.77 Flags for submit-form actions

BIT POSITION	NAME	MEANING
1	Include/Exclude	If clear, the **Fields** array (see Table 8.76) specifies which fields to include in the submission. (All descendants of the specified fields in the field hierarchy are submitted as well.) If set, the **Fields** array tells which fields to exclude; all fields in the document's interactive form are submitted *except* those listed in the **Fields** array and those whose NoExport flag (see Table 8.61 on page 616) is set.
2	IncludeNoValueFields	If set, all fields designated by the **Fields** array and the Include/Exclude flag are submitted, regardless of whether they have a value (**V** entry in the field dictionary); for fields without a value, only the field name is transmitted. If clear, fields without a value are not submitted.
3	ExportFormat	Meaningful only if the SubmitPDF and XFDF flags are clear. If set, field names and values are submitted in HTML Form format. If clear, they are submitted in Forms Data Format (FDF); see Section 8.6.6, "Forms Data Format."
4	GetMethod	If set, field names and values are submitted using an HTTP GET request; if clear, they are submitted using a POST request. This flag is meaningful only when the ExportFormat flag is set; if ExportFormat is clear, this flag must also be clear.

BIT POSITION	NAME	MEANING
5	SubmitCoordinates	If set, the coordinates of the mouse click that caused the submit-form action are transmitted as part of the form data. The coordinate values are relative to the upper-left corner of the field's widget annotation rectangle. They are represented in the data in the format *name*.x=*xval*&*name*.y=*yval* where *name* is the field's mapping name (**TM** in the field dictionary) if present, otherwise the field name. If the value of the **TM** entry is a single space character, both the name and the dot following it are suppressed, resulting in the format x=*xval*&y=*yval* This flag is meaningful only when the ExportFormat flag is set; if ExportFormat is clear, this flag must also be clear.
6	XFDF	*(PDF 1.4)* Meaningful only if the SubmitPDF flags are clear. If set, field names and values are submitted in XFDF format.
7	IncludeAppendSaves	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the XFDF and ExportFormat flags are clear). If set, the submitted FDF file includes the contents of all incremental updates to the underlying PDF document, as contained in the **Differences** entry in the FDF dictionary (see Table 8.84 on page 650); if clear, the incremental updates are not included.
8	IncludeAnnotations	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the XFDF and ExportFormat flags are clear). If set, the submitted FDF file includes all annotations in the underlying PDF document; if clear, the annotations are not included.
9	SubmitPDF	*(PDF 1.4)* If set, the document is submitted in PDF format, using the MIME content type application/pdf (described in Internet RFC 2045, *Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies*; see the Bibliography). If this flag is set, all other flags are ignored except GetMethod.
10	CanonicalFormat	*(PDF 1.4)* If set, any submitted field values representing dates are converted to the standard format described in Section 3.8.3, "Dates." (The interpretation of a form field as a date is not specified explicitly in the field itself, but only in the JavaScript code that processes it.)

BIT POSITION	NAME	MEANING
11	ExclNonUserAnnots	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the XFDF and Export-Format flags are clear) and the IncludeAnnotations flag is set. If set, will include only those annotations whose **T** entry matches the name of the current user, as determined by the remote server to which the form is being submitted. (The **T** entry, which specifies the text label to be displayed in the title bar of the annotation's pop-up window, is assumed to represent the name of the user authoring the annotation.) This allows multiple users to collaborate in annotating a single remote PDF document without affecting one another's annotations.
12	ExclFKey	*(PDF 1.4)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the XFDF and Export-Format flags are clear). If set, the submitted FDF will exclude the **F** entry.
14	EmbedForm	*(PDF 1.5)* Meaningful only when the form is being submitted in Forms Data Format (that is, when both the XFDF and Export-Format flags are clear). If set, the **F** entry of the submitted FDF will be a file specification containing an embedded file stream representing the PDF file from which the FDF is being submitted.

The set of fields whose names and values are to be submitted is defined by the **Fields** array in the action dictionary (Table 8.76) together with the Include/Exclude and IncludeNoValueFields flags in the **Flags** entry (Table 8.77). Each element of the **Fields** array identifies an interactive form field, either by an indirect reference to its field dictionary or *(PDF 1.3)* by its fully qualified field name (see "Field Names" on page 616). If the Include/Exclude flag is clear, the submission consists of all fields listed in the **Fields** array, along with any descendants of those fields in the field hierarchy. If the Include/Exclude flag is set, the submission consists of all fields in the document's interactive form *except* those listed in the **Fields** array.

***Note:** The NoExport flag in the field dictionary's **Ff** entry (see Tables 8.60 on page 615 and 8.61 on page 616) takes precedence over the action's **Fields** array and Include/Exclude flag. Fields whose NoExport flag is set are* never *included in a submit-form action.*

Field names and values may be submitted in any of the following formats, depending on the settings of the action's ExportFormat, SubmitPDF and XFDF flags (see the Bibliography for references):

- HTML Form format (described in the *HTML 4.01 Specification*)
- Forms Data Format (FDF), which is described in Section 8.6.6, "Forms Data Format"; see also implementation note 110 in Appendix H.
- XFDF, a version of FDF based on XML. XFDF is described in the Adobe technical note *XML Forms Data Format Specification, Version 2.0.* XML is described in the World Wide Web Consortium document *Extensible Markup Language (XML) 1.1*)
- PDF format (in this case, the entire document is submitted rather than individual fields and values).

The name submitted for each field is its fully qualified name (see "Field Names" on page 616), and the value is that specified by the **V** entry in its field dictionary.

***Note:** For pushbutton fields submitted in FDF, the value submitted is that of the **AP** entry in the field's widget annotation dictionary. If the submit-form action dictionary contains no **Fields** entry, such pushbutton fields are not submitted at all.*

Fields with no value (that is, whose field dictionary does not contain a **V** entry) are ordinarily not included in the submission. The submit-form action's IncludeNoValueFields flag overrides this behavior; if this flag is set, such valueless fields are included in the submission by name only, with no associated value.

Reset-Form Actions

A *reset-form action* resets selected interactive form fields to their default values; that is, it sets the value of the **V** entry in the field dictionary to that of the **DV** entry (see Table 8.60 on page 615). If no default value is defined for a field, its **V** entry is removed. For fields that can have no value (such as pushbuttons), the action has no effect. Table 8.78 shows the action dictionary entries specific to this type of action.

The value of the action dictionary's **Flags** entry is an unsigned 32-bit integer containing flags specifying various characteristics of the action. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). At the time of publication, only one flag is defined for this type of action; Table 8.79 shows its meaning. All undefined flag bits are reserved and must be set to 0.

TABLE 8.78 Additional entries specific to a reset-form action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **ResetForm** for a reset-form action.
Fields	array	*(Optional)* An array identifying which fields to reset or which to exclude from resetting, depending on the setting of the Include/Exclude flag in the **Flags** entry (see Table 8.79). Each element of the array is either an indirect reference to a field dictionary or *(PDF 1.3)* a string representing the fully qualified name of a field. Elements of both kinds may be mixed in the same array. If this entry is omitted, the Include/Exclude flag is ignored; all fields in the document's interactive form are reset.
Flags	integer	*(Optional; inheritable)* A set of flags specifying various characteristics of the action (see Table 8.79). Default value: 0.

TABLE 8.79 Flag for reset-form actions

BIT POSITION	NAME	MEANING
1	Include/Exclude	If clear, the **Fields** array (see Table 8.78) specifies which fields to reset. (All descendants of the specified fields in the field hierarchy are reset as well.) If set, the **Fields** array tells which fields to exclude from resetting; all fields in the document's interactive form are reset *except* those listed in the **Fields** array.

Import-Data Actions

An *import-data action* imports Forms Data Format (FDF) data into the document's interactive form from a specified file (see Section 8.6.6, "Forms Data Format"). Table 8.80 shows the action dictionary entries specific to this type of action.

TABLE 8.80 Additional entries specific to an import-data action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **ImportData** for an import-data action.
F	file specification	*(Required)* The FDF file from which to import the data. (See implementation notes 111 and 112 in Appendix H.)

JavaScript Actions

A *JavaScript action (PDF 1.3)* causes a script to be compiled and executed by the JavaScript interpreter. Depending on the nature of the script, this can cause various interactive form fields in the document to update their values or change their visual appearances. Netscape Communications Corporation's *Client-Side JavaScript Reference* and Adobe Technical Note #5431, *Acrobat JavaScript Scripting Reference* (see the Bibliography) give details on the contents and effects of JavaScript scripts. Table 8.81 shows the action dictionary entries specific to this type of action.

TABLE 8.81 Additional entries specific to a JavaScript action

KEY	TYPE	VALUE
S	name	*(Required)* The type of action that this dictionary describes; must be **JavaScript** for a JavaScript action.
JS	string or stream	*(Required)* A string or stream containing the JavaScript script to be executed. ***Note: PDFDocEncoding** or Unicode encoding (the latter identified by the Unicode prefix U+FEFF) is used to encode the contents of the string or stream. (See implementation note 113 in Appendix H.)*

To support the use of parameterized function calls in JavaScript scripts, the **JavaScript** entry in a PDF document's name dictionary (see Section 3.6.3, "Name Dictionary") can contain a name tree mapping name strings to document-level JavaScript actions. When the document is opened, all of the actions in this name tree are executed, defining JavaScript functions for use by other scripts in the document.

Note: *The name strings associated with individual JavaScript actions in the name dictionary serve merely as a convenient means for organizing and packaging scripts. The names are arbitrary and need not bear any relation to the JavaScript name space itself.*

8.6.5 Named Pages

The optional **Pages** entry *(PDF 1.3)* in a document's name dictionary (see Section 3.6.3, "Name Dictionary") contains a name tree that maps name strings to individual pages within the document. Naming a page allows it to be referenced in two different ways:

- An import-data action can add the named page to the document into which FDF is being imported, either as a page or as a button appearance.
- A script executed by a JavaScript action can add the named page to the current document as a regular page.

A named page that is to be visible to the user should be left in the page tree (see Section 3.6.2, "Page Tree"), with a reference to it in the appropriate leaf node of the name dictionary's **Pages** tree. If the page is not to be displayed by the viewer application, it should be referenced from the name dictionary's **Templates** tree instead. Such invisible pages should have an object type of **Template** rather than **Page**, and should have no **Parent** or **B** entry (see Table 3.27 on page 118). Regardless of whether the page is named in the **Pages** or **Templates** tree or whether it is added to a document by an import-data or JavaScript action, the new copy is not itself named.

8.6.6 Forms Data Format

This section describes Forms Data Format (FDF), the file format used for interactive form data *(PDF 1.2)*. FDF is used when submitting form data to a server, receiving the response, and incorporating it into the interactive form. It can also be used to export form data to stand-alone files that can be stored, transmitted electronically, and imported back into the corresponding PDF interactive form. In addition, beginning in PDF 1.3, it can be used to define a container for annotations that are separate from the PDF document to which they apply.

FDF is based on PDF; it uses the same syntax (see Section 3.1, "Lexical Conventions") and basic object types (Section 3.2, "Objects"), and has essentially the same file structure (Section 3.4, "File Structure"). However, it differs from PDF in the following ways:

- The cross-reference table (Section 3.4.3, "Cross-Reference Table") is optional.
- FDF files cannot be updated (see Section 3.4.5, "Incremental Updates"); objects can only be of generation 0, and no two objects can have the same object number.
- The document structure is much simpler than PDF, since the body of an FDF document consists of only one required object.
- The length of a stream may not be specified by an indirect object.

FDF uses the MIME content type application/vnd.fdf. On the Windows and UNIX platforms, FDF files have the extension .fdf; on Mac OS, they have file type 'FDF '.

FDF File Structure

An FDF file is structured in essentially the same way as a PDF file, but need contain only those elements required for the export and import of interactive form and annotation data. It consists of three required elements and one optional one (see Figure 8.6):

- A one-line *header* identifying the version number of the PDF specification to which the file conforms
- A *body* containing the objects that make up the content of the file
- An optional *cross-reference table* containing information about the objects in the file
- A *trailer* giving the location of various objects within the body of the file

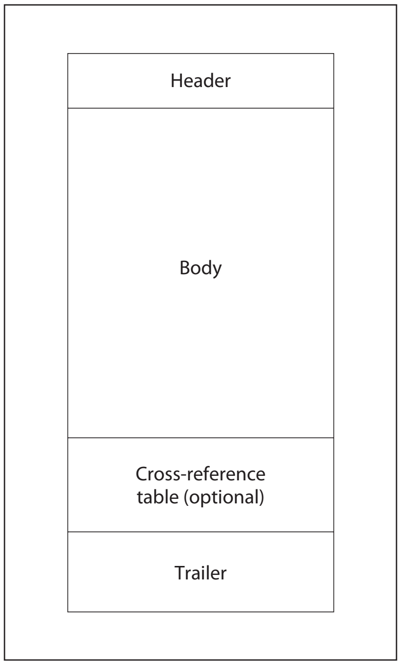


FIGURE 8.6 *FDF file structure*

FDF Header

The first line of an FDF file is a *header*, originally intended to identify the version of the PDF specification to which the file conforms. However, for historical reasons, this version number is now frozen and must read

```
%FDF−1.2
```

The true version number is now given by the **Version** entry in the FDF catalog dictionary (see "FDF Catalog," below; see also implementation note 114 in Appendix H).

FDF Body

The *body* of an FDF file consists of a sequence of indirect objects representing the file's catalog (see "FDF Catalog" on page 649), together with any additional objects that the catalog may reference. The objects are of the same basic types described in Section 3.2, "Objects." Just as in PDF, objects in FDF can be direct or indirect.

FDF Trailer

The *trailer* of an FDF file enables an application reading the file to find significant objects quickly within the body of the file. The last line of the file contains only the end-of-file marker, %%EOF. This is preceded by the *FDF trailer dictionary*, consisting of the keyword **trailer** followed by a series of one or more key-value pairs enclosed in double angle brackets (<<…>>). The only required key is **Root**, whose value is an indirect reference to the file's catalog dictionary (see Table 8.82). The trailer may optionally contain additional entries for objects that are referenced from within the catalog.

TABLE 8.82 Entry in the FDF trailer dictionary

KEY	TYPE	VALUE
Root	dictionary	*(Required; must be an indirect reference)* The catalog object for this FDF file (see "FDF Catalog," below).

Thus the trailer has the following overall structure:

```
trailer
    <<  /Root  c 0 R
        key2  value2
        …
        keyn  valuen
    >>
%%EOF
```

where *c* is the object number of the file's catalog dictionary.

FDF Catalog

The root node of an FDF file's object hierarchy is the *catalog* dictionary, located via the **Root** entry in the file's trailer dictionary (see "FDF Trailer," above). As shown in Table 8.83, the only required entry in the catalog is **FDF**; its value is an *FDF dictionary* (Table 8.84), which in turn contains references to other objects describing the file's contents. The catalog may also contain an optional **Version** entry identifying the version of the PDF specification to which this FDF file conforms.

TABLE 8.83 Entries in the FDF catalog dictionary

KEY	TYPE	VALUE
Version	name	*(Optional; PDF 1.4)* The version of the PDF specification to which this FDF file conforms (for example, 1.4), if later than the version specified in the file's header (see "FDF Header" on page 648). If the header specifies a later version, or if this entry is absent, the document conforms to the version specified in the header. ***Note:*** *The value of this entry is a name object, not a number, and so must be preceded by a slash character (/) when written in the FDF file (for example, /1.4).*
FDF	dictionary	*(Required)* The FDF dictionary for this file (see Table 8.84).
Sig	dictionary	*(Optional; PDF 1.5)* A signature dictionary, indicating that the document is signed using an object digest (see Section 8.7, "Digital Signatures"). This dictionary must contain a signature reference dictionary whose **Data** entry is an indirect reference to the catalog and whose **TransFormMethod** entry is **Identity**.

TABLE 8.84 Entries in the FDF dictionary

KEY	TYPE	VALUE
F	file specification	*(Optional)* The *source file* or *target file*: the PDF document file that this FDF file was exported from or is intended to be imported into.
ID	array	*(Optional)* An array of two strings constituting a file identifier (see Section 10.3, "File Identifiers") for the source or target file designated by **F**, taken from the **ID** entry in the file's trailer dictionary (see Section 3.4.4, "File Trailer").
Fields	array	*(Optional)* An array of FDF field dictionaries (see "FDF Fields" on page 653) describing the root fields (those with no ancestors in the field hierarchy) to be exported or imported. This entry and the **Pages** entry may not both be present.

KEY	TYPE	VALUE
Status	string	*(Optional)* A status string to be displayed indicating the result of an action, typically a submit-form action (see "Submit-Form Actions" on page 639). The string is encoded with **PDFDocEncoding**. (See implementation note 115 in Appendix H.) This entry and the **Pages** entry may not both be present.
Pages	array	*(Optional; PDF 1.3)* An array of FDF page dictionaries (see "FDF Pages" on page 656) describing new pages to be added to a PDF target document. The **Fields** and **Status** entries may not be present together with this entry.
Encoding	name	*(Optional; PDF 1.3)* The encoding to be used for any FDF field value or option (**V** or **Opt** in the field dictionary; see Table 8.87 on page 653) or field name that is a string and does not begin with the Unicode prefix U+FEFF. (See implementation note 116 in Appendix H.) Default value: **PDFDocEncoding**.
Annots	array	*(Optional; PDF 1.3)* An array of FDF annotation dictionaries (see "FDF Annotation Dictionaries" on page 658). The array can include annotations of any of the standard types listed in Table 8.16 on page 570 except **Link**, **Movie**, **Widget**, **PrinterMark**, **Screen** and **TrapNet**.
Differences	stream	*(Optional; PDF 1.4)* A stream containing all the bytes in all incremental updates made to the underlying PDF document since it was opened (see Section 3.4.5, "Incremental Updates"). If a submit-form action submitting the document to a remote server in FDF format has its IncludeAppendSaves flag set (see "Submit-Form Actions" on page 639), the contents of this stream are included in the submission. This allows any digital signatures (see Section 8.7, "Digital Signatures) to be transmitted to the server. An incremental update is automatically performed just before the submission takes place, in order to capture all changes made to the document. Note that the submission always includes the full set of incremental updates back to the time the document was first opened, even if some of them may already have been included in intervening submissions. ***Note:** Although a **Fields** or **Annots** entry (or both) may be present along with **Differences**, there is no guarantee that their contents will be consistent with it. In particular, if **Differences** contains a digital signature, only the values of the form fields given in the **Differences** stream can be considered trustworthy under that signature.*

KEY	TYPE	VALUE
Target	string	*(Optional; PDF 1.4)* The name of a browser frame in which the underlying PDF document is to be opened. This mimics the behavior of the target attribute in HTML <href> tags.
EmbeddedFDFs	array	*(Optional; PDF 1.4)* An array of file specifications (see Section 3.10, "File Specifications") representing other FDF files embedded within this one (Section 3.10.3, "Embedded File Streams").
JavaScript	dictionary	*(Optional; PDF 1.4)* A *JavaScript dictionary* (see Table 8.86) defining document-level JavaScript scripts.

Embedded FDF files specified in the FDF dictionary's **EmbeddedFDFs** entry may optionally be encrypted. Besides the usual entries for an embedded file stream, the stream dictionary representing such an encrypted FDF file must contain the additional entry shown in Table 8.85 to identify the revision number of the FDF encryption algorithm used to encrypt the file. Although the FDF encryption mechanism is separate from the one for PDF file encryption described in Section 3.5, "Encryption," revision 1 (the only one defined at the time of publication) uses a similar RC4 encryption algorithm based on a 40-bit encryption key. The key is computed via an MD5 hash, using a padded user-supplied password as input. The computation is identical to steps 1 and 2 of Algorithm 3.2 on page 99; the first 5 bytes of the result are the encryption key for the embedded FDF file.

TABLE 8.85 Additional entry in an embedded file stream dictionary for an encrypted FDF file

KEY	TYPE	VALUE
EncryptionRevision	integer	*(Required if the FDF file is encrypted; PDF 1.4)* The revision number of the FDF encryption algorithm used to encrypt the file. The only valid value defined at the time of publication is 1.

The **JavaScript** entry in the FDF dictionary holds a *JavaScript dictionary* containing JavaScript scripts that are defined globally at the document level, rather than associated with individual fields. The dictionary can contain scripts defining JavaScript functions for use by other scripts in the document, as well as scripts to be executed immediately before and after the FDF file is imported. Table 8.86 shows the contents of this dictionary.

TABLE 8.86 Entries in the JavaScript dictionary

KEY	TYPE	VALUE
Before	string or stream	*(Optional)* A string or stream containing a JavaScript script to be executed just before the FDF file is imported.
After	string or stream	*(Optional)* A string or stream containing a JavaScript script to be executed just after the FDF file is imported.
Doc	array	*(Optional)* An array defining additional JavaScript scripts to be added to those defined in the **JavaScript** entry of the document's name dictionary (see Section 3.6.3, "Name Dictionary"). The array contains an even number of elements, organized in pairs. The first element of each pair is a name and the second is a string or stream defining the script corresponding to that name. Each of the defined scripts will be added to those already defined in the name dictionary and then executed before the script defined in the **Before** entry is executed. As described in "JavaScript Actions" on page 645, these scripts are used to define JavaScript functions for use by other scripts in the document.

FDF Fields

Each field in an FDF file is described by an *FDF field dictionary.* Table 8.87 shows the contents of this type of dictionary. Most of the entries have the same form and meaning as the corresponding entries in a field dictionary (Tables 8.60 on page 615, 8.62 on page 618, 8.69 on page 632, and 8.71 on page 634) or a widget annotation dictionary (Tables 8.11 on page 560 and 8.35 on page 589). Unless otherwise indicated in the table, importing a field causes the values of the entries in the FDF field dictionary to replace those of the corresponding entries in the field with the same fully qualified name in the target document. (See implementation notes 117–122 in Appendix H.)

TABLE 8.87 Entries in an FDF field dictionary

KEY	TYPE	VALUE
Kids	array	*(Optional)* An array containing the immediate children of this field. ***Note:** Unlike the children of fields in a PDF file, which must be specified as indirect object references, those of an FDF field may be either direct or indirect objects.*
T	text string	*(Required)* The partial field name (see "Field Names" on page 616).

KEY	TYPE	VALUE
V	(various)	*(Optional)* The field's value, whose format varies depending on the field type; see the descriptions of individual field types in Section 8.6.3 for further information.
Ff	integer	*(Optional)* A set of flags specifying various characteristics of the field (see Tables 8.61 on page 616, 8.66 on page 626, 8.68 on page 631, and 8.70 on page 634). When imported into an interactive form, the value of this entry replaces that of the **Ff** entry in the form's corresponding field dictionary. If this field is present, the **SetFf** and **ClrFf** entries, if any, are ignored.
SetFf	integer	*(Optional)* A set of flags to be set (turned on) in the **Ff** entry of the form's corresponding field dictionary. Bits equal to 1 in **SetFf** cause the corresponding bits in **Ff** to be set to 1. This entry is ignored if an **Ff** entry is present in the FDF field dictionary.
ClrFf	integer	*(Optional)* A set of flags to be cleared (turned off) in the **Ff** entry of the form's corresponding field dictionary. Bits equal to 1 in **ClrFf** cause the corresponding bits in **Ff** to be set to 0. If a **SetFf** entry is also present in the FDF field dictionary, it is applied before this entry. This entry is ignored if an **Ff** entry is present in the FDF field dictionary.
F	integer	*(Optional)* A set of flags specifying various characteristics of the field's widget annotation (see Section 8.4.2, "Annotation Flags"). When imported into an interactive form, the value of this entry replaces that of the **F** entry in the form's corresponding annotation dictionary. If this field is present, the **SetF** and **ClrF** entries, if any, are ignored.
SetF	integer	*(Optional)* A set of flags to be set (turned on) in the **F** entry of the form's corresponding widget annotation dictionary. Bits equal to 1 in **SetF** cause the corresponding bits in **F** to be set to 1. This entry is ignored if an **F** entry is present in the FDF field dictionary.
ClrF	integer	*(Optional)* A set of flags to be cleared (turned off) in the **F** entry of the form's corresponding widget annotation dictionary. Bits equal to 1 in **ClrF** cause the corresponding bits in **F** to be set to 0. If a **SetF** entry is also present in the FDF field dictionary, it is applied before this entry. This entry is ignored if an **F** entry is present in the FDF field dictionary.
AP	dictionary	*(Optional)* An appearance dictionary specifying the appearance of a pushbutton field (see "Pushbuttons" on page 626). The appearance dictionary's contents are as shown in Table 8.15 on page 569, except that the values of the **N**, **R**, and **D** entries must all be streams.

KEY	TYPE	VALUE
APRef	dictionary	*(Optional; PDF 1.3)* A dictionary holding references to external PDF files containing the pages to use for the appearances of a pushbutton field. This dictionary is similar to an appearance dictionary (see Table 8.15 on page 569), except that the values of the **N**, **R**, and **D** entries must all be named page reference dictionaries (Table 8.91 on page 658). This entry is ignored if an **AP** entry is present.
IF	dictionary	*(Optional; PDF 1.3; button fields only)* An icon fit dictionary (see Table 8.88) specifying how to display a button field's icon within the annotation rectangle of its widget annotation.
Opt	array	*(Required; choice fields only)* An array of options to be presented to the user. Each element of the array can take either of two forms: • A text string representing one of the available options • A two-element array consisting of a text string representing one of the available options and a default appearance string for constructing the item's appearance dynamically at viewing time (see "Variable Text" on page 617)
A	dictionary	*(Optional)* An action to be performed when this field's widget annotation is activated (see Section 8.5, "Actions").
AA	dictionary	*(Optional)* An additional-actions dictionary defining the field's behavior in response to various trigger events (see Section 8.5.2, "Trigger Events").
RV	text string or text stream	*(Optional; PDF 1.5)* A rich text string, as described in "Rich Text Strings" on page 620.

In an FDF field dictionary representing a button field, the optional **IF** entry holds an *icon fit dictionary (PDF 1.3)* specifying how to display the button's icon within the annotation rectangle of its widget annotation. Table 8.88 shows the contents of this type of dictionary.

TABLE 8.88 Entries in an icon fit dictionary

KEY	TYPE	VALUE
SW	name	*(Optional)* The circumstances under which the icon should be scaled inside the annotation rectangle: A Always scale. B Scale only when the icon is bigger than the annotation rectangle. S Scale only when the icon is smaller than the annotation rectangle. N Never scale. Default value: A.
S	name	*(Optional)* The type of scaling to use: A *Anamorphic scaling*: scale the icon to fill the annotation rectangle exactly, without regard to its original aspect ratio (ratio of width to height). P *Proportional scaling*: scale the icon to fit the width or height of the annotation rectangle while maintaining the icon's original aspect ratio. If the required horizontal and vertical scaling factors are different, use the smaller of the two, centering the icon within the annotation rectangle in the other dimension. Default value: P.
A	array	*(Optional)* An array of two numbers between 0.0 and 1.0 indicating the fraction of left-over space to allocate at the left and bottom of the icon. A value of [0.0 0.0] positions the icon at the bottom-left corner of the annotation rectangle; a value of [0.5 0.5] centers it within the rectangle. This entry is used only if the icon is scaled proportionally. Default value: [0.5 0.5].
FB	boolean	*(Optional; PDF 1.5)* If **true**, indicates that the button appearance should be scaled to fit fully within the bounds of the annotation, without taking into consideration the line width of the border; see implementation note 123 in Appendix H. Default value: **false**.

FDF Pages

The optional **Pages** field in an FDF dictionary (see Table 8.84 on page 650) contains an array of *FDF page dictionaries (PDF 1.3)* describing new pages to be added to the target document. Table 8.89 shows the contents of this type of dictionary.

TABLE 8.89 Entries in an FDF page dictionary

KEY	TYPE	VALUE
Templates	array	*(Required)* An array of *FDF template dictionaries* (see Table 8.90) describing the named pages that serve as templates on the page.
Info	dictionary	*(Optional)* An *FDF page information dictionary* containing additional information about the page. At the time of publication, no entries have been defined for this dictionary.

An *FDF template dictionary* contains information describing a named page that serves as a template. Table 8.90 shows the contents of this type of dictionary.

TABLE 8.90 Entries in an FDF template dictionary

KEY	TYPE	VALUE
TRef	dictionary	*(Required)* A named page reference dictionary (see Table 8.91) specifying the location of the template.
Fields	array	*(Optional)* An array of references to FDF field dictionaries (see Table 8.87 on page 653) describing the root fields to be imported (those with no ancestors in the field hierarchy).
Rename	boolean	*(Optional)* A flag specifying whether fields imported from the template may be renamed in the event of name conflicts with existing fields; see below for further discussion. Default value: **true**.

The names of fields imported from a template may sometimes conflict with those of existing fields in the target document. This can occur, for example, if the same template page is imported more than once or if two different templates have fields with the same names. If the **Rename** flag in the FDF template dictionary is **true**, fields with such conflicting names are renamed to guarantee their uniqueness. If **Rename** is **false**, the fields are not renamed; this results in multiple fields with the same name in the target document. Each time the FDF file provides attributes for a given field name, all fields with that name will be updated. (See implementation notes 124 and 125 in Appendix H.)

The **TRef** entry in an FDF template dictionary holds a *named page reference dictionary* describing the location of external templates or page elements. Table 8.91 shows the contents of this type of dictionary.

TABLE 8.91 Entries in an FDF named page reference dictionary

KEY	TYPE	VALUE
Name	string	*(Required)* The name of the referenced page.
F	file specification	*(Optional)* The file containing the named page. If this entry is absent, it is assumed that the page resides in the associated PDF file.

FDF Annotation Dictionaries

Each annotation dictionary in an FDF file must have a **Page** entry (see Table 8.92) indicating the page of the source document to which the annotation is attached.

TABLE 8.92 Additional entry for annotation dictionaries in an FDF file

KEY	TYPE	VALUE
Page	integer	*(Required for annotations in FDF files)* The ordinal page number on which this annotation should appear, where page 0 is the first page.

8.6.7 XFA Forms

PDF 1.5 introduces support for interactive forms based on Adobe's XML Forms Architecture (XFA). The **XFA** entry in the interactive forms dictionary (see Table 8.58) specifies an *XFA resource*, which is an XML stream containing information, including but not limited to the following:

- The *form template*, which describes the characteristics of the form, including its fields, calculations, validations and formatting.
- The *data*, which represents the state of the form.
- The *configuration information*, which is required to properly process the form template and associated data.

When creating or modifying a PDF file whose interactive forms dictionary has an **XFA** entry:

- Viewer applications should ensure that the field values in the XFA resource are consistent with the corresponding **V** entries of the PDF field objects.
- No **A** or **AA** entries (see Table 8.11) should be present in the annotation dictionaries of fields that also have actions specified by the XFA resource. The behavior of a field whose actions are specified in both ways is undefined.

8.7 Digital Signatures

A digital signature can be used to authenticate the identity of a user and the validity of a document's contents. It stores information about the signer and the state of the document when it was signed. The signature may be purely mathematical, such as a public/private-key encrypted document digest, or it may be a biometric form of identification such as a handwritten signature, fingerprint, or retinal scan. The specific form of authentication used is implemented by a plug-in *signature handler*. Third-party handler writers are encouraged to register their handler names with Adobe; see Appendix E.

Signature information is contained in a *signature dictionary (PDF 1.3)*, whose entries are listed in Table 8.93. Signature handlers are free to use or omit those entries that are marked optional in the table, but are encouraged to use them in a standard way if they are used at all. In addition, signature handlers may add private entries of their own. To avoid name duplication, it is suggested that the keys for all such private entries be prefixed with the registered handler name followed by a period (.).

Signature dictionaries can occur as values of the following dictionary entries:

- The **V** entry of a signature field dictionary (see "Signature Fields" on page 636)
- The **UR** entry of a permissions dictionary (see Section 8.7.3, "Permissions")

  Note: *The **DocMDP** entry of a permissions dictionary also contains an indirect reference to a signature dictionary that is part of a signature field.*
- The **Sig** entry in the catalog of an FDF file (see "FDF Catalog" on page 649)

TABLE 8.93 Entries in a signature dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Sig** for a signature dictionary.
Filter	name	*(Required; inheritable)* The name of the preferred signature handler to use when validating this signature. If the **Prop_Build** entry is not present, it is also the name of the signature handler that was used to create the signature. If **Prop_Build** is present, it can be used to determine the name of the handler that created the signature (which is typically the same as **Filter**, but is not required to be). A viewer application may substitute a different handler when verifying the signature, as long is it supports the specified **SubFilter** format. Example signature handlers are as **Adobe.PPKLite**, **Entrust.PPKEF**, **CICI.SignIt**, and **VeriSign.PPKVS**.
SubFilter	name	*(Optional)* A name that describes the encoding of the signature value and key information in the signature dictionary. A viewer application may use any handler that supports this format to validate the signature. Defined values for public-key cryptographic signatures are **adbe.x509.rsa_sha1**, **adbe.pkcs7.detached** and **adbe.pkcs7.sha1** (see Section 8.7.1, "Signature Interoperability").
Contents	string	*(Required)* The signature value. When **ByteRange** is specified, the value is a hexadecimal string (see "Hexadecimal Strings" on page 32) representing the value of the byte range digest. If **ByteRange** is not specified, the value is an object digest of the signature dictionary, excluding the **Contents** entry. For public-key signatures, **Contents** is commonly either a DER-encoded PKCS#1 binary data object or a DER-encoded PKCS#7 binary data object.
Cert	array or string	*(Required when **SubFilter** is **adbe.x509.rsa_sha1**; not used otherwise)* An array of strings representing the X.509 certificate chain used when signing and verifying signatures that use public-key cryptography, or a string if the chain has only one entry. The signing certificate must appear first in the array; it is used to verify the signature value in **Contents**, and the other certificates are used to verify the authenticity of the signing certificate. If **SubFilter** is **adbe.pkcs7.detached** or **adbe.pkcs7.sha1**, this entry is not used, and the certificate chain must be put in the PKCS#7 envelope in **Contents**.

KEY	TYPE	VALUE
ByteRange	array	*(Required if the signature dictionary is part of a signature field)* An array of pairs of integers (starting byte offset, length in bytes) describing the exact byte range for the digest calculation. Multiple discontiguous byte ranges are used to describe a digest that does not include the signature value (the **Contents** entry) itself.
Reference	array	*(Required if the signature dictionary includes an object digest; PDF 1.5)* An array of signature reference dictionaries (see Table 8.94).
Changes	array	*(Optional)* An array of three integers specifying changes to the document that have been made between the previous signature and this signature: in this order, the number of pages altered, the number of fields altered, and the number of fields filled in. ***Note:** The ordering of signatures is determined by the value of* ***ByteRange****: since each signature results in an incremental save, later signatures have a greater length value.*
Name	text string	*(Optional)* The name of the person or authority signing the document.
M	date	*(Optional)* The time of signing. Depending on the signature handler, this may be a normal unverified computer time or a time generated in a verifiable way from a secure time server.
Location	text string	*(Optional)* The CPU host name or physical location of the signing.
Reason	text string	*(Optional)* The reason for the signing, such as (I agree …).
Location	text string	*(Optional)* The CPU hostname or physical location of the signing.
ContactInfo	text string	*(Optional)* Information provided by the signer to allow a recipient to contact the signer to verify the signature; for example, a phone number.
R	integer	*(Optional)* The version of the signature handler that was used to create the signature. ***Note:** Beginning with PDF 1.5, this entry is deprecated, and the information should be stored in the* ***Prop_Build*** *dictionary.*
V	integer	*(Optional; PDF 1.5)* The version of the signature dictionary format; it corresponds to the usage of the signature dictionary in the context of the value of **SubFilter**. The value is 1 if the **Reference** dictionary is considered critical to the validation of the signature. Default value: 0.

KEY	TYPE	VALUE
Prop_Build	dictionary	*(Optional; PDF 1.5)* A dictionary that can be used by a signature handler to record information that captures the state of the computer environment used for signing, such as the name of the handler used to create the signature, software build date, version, and operating system. Adobe publishes a separate specification, the *PDF Signature Build Dictionary Specification for Acrobat 6.0* that provides implementation guidelines for the use of this dictionary.
Prop_AuthTime	integer	*(Optional; PDF 1.5)* The number of seconds since the signer was last authenticated; it is intended to be used in claims of signature repudiation. It should be omitted if the value is unknown.
Prop_AuthType	name	*(Optional; PDF 1.5)* The method used to authenticate the signer; it is intended to be used in claims of signature repudiation. Possible values include Password, PIN and Fingerprint.

***Note:** The entries in the signature dictionary can be conceptualized as being in different dictionaries; they are in one dictionary for historical and cryptographic reasons. The categories are: signature properties (**R**, **M**, **Name**, **Reason**, **Location**, **Prop_Build**, **Prop_AuthTime**, and **Prop_AuthType**); key information (**Cert** and portions of **Contents** when the signature value is a PKCS#7 object); reference: **Reference** and **ByteRange**); and signature value (**Contents** when the signature value is a PKCS#1 object).*

Signatures are created by computing a *digest* of the data in a document, or part of the data, which is stored in the document. To verify the signature, the digest is recomputed and compared with the one stored in the document. Differences in the digest values indicate that modifications have been made since the document was signed. Therefore, techniques for computing the digest must be reproducible, and the contents of the document must not be able to be derived from the digest itself.

There are two defined techniques for computing a reproducible digest of the contents of all or part of a PDF file:

- A *byte range digest* is computed over a range of bytes in the file, usually the entire file. This range includes the signature dictionary, excluding the signature value itself (the **Contents** entry). When a byte range digest is present, all values in the signature dictionary are required to be direct objects. See implementation note 126 in Appendix H.

- An *object digest (PDF 1.5)* is computed by selectively walking a subtree of objects in memory, beginning with the referenced object, which is typically the root object. The resulting digest, along with information about how it was computed, is placed in a *signature reference dictionary*, whose entries are listed in Table 8.94. The **TransformMethod** entry specifies the general method used to compute the digest and the **TransformParams** entry specifies the variable portion of the computation. Transform methods are described in detail in Section 8.7.2, "Transform Methods."

TABLE 8.94 Entries in a signature reference dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SigRef** for a signature reference dictionary.
TransformMethod	name	*(Required)* The name of the transform method (see Section 8.7.2, "Transform Methods") that was applied to the object prior to computing the digest. Defined values are: **DocMDP** Used to detect modifications to a document relative to a signature field that is signed by the originator of a document; see "DocMDP" on page 666. **UR** Used to detect modifications to a document that would invalidate a signature in a rights-enabled document; see "UR" on page 667. **FieldMDP** Used to detect modifications to a list of form fields specified in **TransformParams**; see "FieldMDP" on page 669. **Identity** Used when signing a single object, specified by the value of **Data** in the signature reference dictionary (see Table 8.94); it is used to support signing of FDF files. See "Identity" on page 670 for details.
TransformParams	dictionary	*(Optional)* A dictionary specifying transform parameters (variable data) for the transform method specified by **TransformMethod**. Each method except **Identity** takes its own set of parameters. See each of the sections specified above for details on the individual transform parameter dictionaries

KEY	TYPE	VALUE
Data	various	*(Required when **TransformMethod** is **FieldMDP** or **Identity**)* An indirect reference to the object in the document over which the digest was computed. For transform methods other than **FieldMDP** and **Identity**, this object is implicitly defined.
DigestMethod	name	*(Optional)* A name identifying the algorithm to be used when computing the digest. Defined values are **MD5** and **SHA1**. Default value: **MD5**.
DigestValue	string	*(Required)* The computed value of the digest.
DigestLocation	array	*(Required when **TransformMethod** is **FieldMDP** or **DocMDP**)* An array of two integers specifying the location in the PDF file of the **DigestValue** string. The integers represent the starting offset and length in bytes, respectively. It is required when **DigestValue** is written directly to the PDF file, by passing any encryption that has been performed on the document. When specified, the values must be used to read **DigestValue** directly from the file during validation.

8.7.1 Signature Interoperability

It is intended that PDF viewer applications allow interoperability between signature handlers; that is, a PDF file signed with a handler from one vendor must be able to be validated with a handler from a different vendor.

The **SubFilter** entry in the signature dictionary specifies the encoding of the signature value and key information, while the **Filter** entry specifies the preferred handler to use to validate the signature. Handlers specify the **SubFilter** encodings they support; therefore, handlers other than the preferred handler can be used to validate the signature if necessary or desired.

There are several defined values for the **SubFilter** entry, all based on public-key cryptographic standards published by RSA Security and also as part of the standards issued by the Internet Engineering Task Force (IETF) Public Key Infrastructure (PKIX) working group; see the Bibliography for references.

PKCS#1 Signatures

The PKCS#1 standard supports several public-key cryptographic algorithms and digest methods, including RSA encryption, DSA signatures, and SHA-1 and MD5 digests (see the Bibliography for references). For signing PDF files using PKCS#1, the only recommended value of **SubFilter** is **adbe.x509.rsa_sha1**, which uses the RSA encryption algorithm and SHA-1 digest method. The certificate chain of the signer is stored in the **Cert** entry.

PKCS#7 Signatures

When PKCS#7 signatures are used, the value of **Contents** is a PKCS#7 signature object. The value of **SubFilter** can be one of the following:

- **adbe.pkcs7.detached**: No data is encapsulated in the PKCS#7 signed-data field.
- **adbe.pkcs7.sha1**: The SHA1 digest is in the signed-data field.

The minimum required certificate included in the PKCS#7 object is the signer's X.509 signing certificate. The PKCS#7 object may optionally contain one or more issuer certifications from the signer's trust chain.

For byte range signatures, **Contents** is a hexadecimal string with "<" and ">" delimiters. It must fit precisely in the space between the ranges specified by **ByteRange**. Since the length of PKCS#7 objects is not entirely predictable, it is often necessary to pad the value of **Contents** with zeros at the end of the string, before the ">" delimiter, before writing the PKCS#7 to the allocated space in the file.

A PKCS#7 object is a wrapper for signing information that, when using PKCS#1, can be found in the signature dictionary. PKCS#7 signatures require more space in a PDF file and can vary in size. PKCS#1 signatures are predictable in size and compact, 131 bytes for a 1024-bit RSA signature, before hex encoding.

It is possible to do an on-the-fly conversion of PKCS#1 signature values to PKCS#7 signature values, while the reverse conversion is not possible. The implication is that PKCS#1 signatures can be validated by handlers that do not directly support PKCS#1, but PKCS#7 signatures cannot be validated by handlers that support only PKCS#1. PKCS#1 signatures are therefore recommended in all cases where the where the added capabilities of PKCS#7 are not required.

8.7.2 Transform Methods

Transform methods specify the general method used to compute an object digest, including which objects are included in and excluded from the digest. Further information is provided by transform parameters. The following sections describe the types of transform methods and their transform parameters. Appendix I, "Computation of Object Digests," describes in detail the algorithm for computing the object digests.

***Note:** All transform methods exclude the signature dictionary from the object digest.*

DocMDP

The **DocMDP** transform method is used to detect modifications relative to a signature field that is signed by the author of a document (the person applying the first signature). There is permitted to be only one signature field per document that contains a **DocMDP** transform method; it must be the first signed field in the document. It allows the author to specify changes to the document that will invalidate the signature.

The term "MDP" stands for *modification detection and prevention.* Object signatures enable *detection* of disallowed changes specified by the author. In addition, disallowed changes can also be *prevented* by the **DocMDP** entry in the permissions dictionary (see Section 8.7.3, "Permissions").

***Note:** When creating an author signature, viewer applications should also create a legal attestation dictionary (see Section 8.7.4, "Legal Content Attestations") that specifies all content that might result in unexpected rendering of the document contents.*

Table 8.95 lists the entries in the **DocMDP** transform parameters dictionary. The **P** entry indicates which changes to the document will invalidate the signature, as specified by the author. It also specifies which changes to the document should be prevented, if this signature is referred to from the **DocMDP** entry in the permissions dictionary. A value of 1 for **P** indicates that the document is intended to be final; that is, any changes invalidate the signature. The values 2 and 3 are used for documents that are intended for form field or comment workflows.

The **DocMDP** object digest is computed over a subset of the PDF objects in the document: that is, the objects that are not modified, directly or indirectly, by permissible operations. Therefore, the digest changes only if a disallowable change has been made. See Appendix I for details on how the **DocMDP** digest is computed.

TABLE 8.95 Entries in the DocMDP transform parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **TransformParams** for a transform parameters dictionary.
P	number	*(Optional)* The access permissions granted for this document. Defined values are: 1 — No changes to the document are permitted; any change to the document invalidates the signature. 2 — Permitted changes are filling in forms, instantiating page templates, and signing; other changes invalidate the signature. 3 — Permitted changes are those allowed by 2, as well as annotation creation, deletion, and modification; other changes invalidate the signature. Default value: 1.
V	number	*(Optional)* The **DocMDP** transform parameters dictionary version number. The value for PDF 1.5 is 1.0. Default: 1.0.

UR

The **UR** transform method is used to detect changes to a document that would invalidate a *usage rights* signature, which is referred to from the **UR** entry in the permissions dictionary (see Section 8.7.3, “Permissions). Usage rights signatures are used to enable additional interactive features that are not available by default in a particular viewer application (such as Adobe Reader). The signature is used to validate that the permissions have been granted by the granting authority. The transform parameters dictionary (see Table 8.96) specifies the additional rights that are enabled if the signature is valid. If the signature is invalid, because the document has been modified in a way that is not permitted or the identity of the signer is not granted the extended permissions, additional rights are not granted.

Adobe Systems grants permissions, for example, to enable additional features in Adobe Reader, using public-key cryptography. It uses certificate authorities to issue public key certificates to document creators with which it has entered into a business relationship. Adobe Reader will verify that the rights-enabling signature uses a certificate from an Adobe-authorized certificate authority. Other PDF viewer applications are free to use this same mechanism for their own purposes.

The **UR** object digest is computed over a subset of the PDF objects in the document: that is, the objects that are not modified, directly or indirectly, by permissible operations, as specified by the transform parameters dictionary. Therefore, the digest changes only if a disallowable change has been made. See Appendix I for details on how the **UR** digest is computed.

TABLE 8.96 Entries in the UR transform parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **TransformParams** for a transform parameters dictionary.
Document	array	*(Optional)* An array of names specifying additional document-wide usage rights for the document. The only defined value is FullSave, which allows a user to save the document along with modified form and/or annotation data.
Form	array	*(Optional)* An array of names specifying additional form-field-related usage rights for the document. Defined values are: Import — Allows the user to import form data files in FDF, XFDF and text (CSV/TSV) formats. Export — Allows the user to export form data files in FDF or XFDF formats. SubmitStandalone — Allows the user to submit form data. SpawnTemplate — Allows the user to instantiate page templates.
Signature	array	*(Optional)* An array of names specifying additional signature-related usage rights for the document. The only defined value is Modify, which allows a user to apply a digital signature to an existing signature form field or clear a signed signature form field. If specified, it implicitly enables a **Document** value of FullSave.
Annots	array	*(Optional)* An array of names specifying additional annotation-related usage rights for the document. Defined values are: Create, Delete, Modify, Copy, Import, and Export, which allow the user to perform the named operation on annotations.
Msg	string	*(Optional)* A string that can be used to specify any arbitrary information, such as the reason for adding usage rights to the document.

KEY	TYPE	VALUE
V	number	*(Optional)* The **UR** transform parameters dictionary version number. The value for PDF 1.5 is 2. Default value: 2.

FieldMDP

The **FieldMDP** transform method computes an object digest over a list of form field objects, and is used to detect changes to the values those form fields. The entries in its transform parameters dictionary are listed in Table 8.97.

TABLE 8.97 Entries in the FieldMDP transform parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **TransformParams** for a transform parameters dictionary.
Action	name	*(Required)* A name that, along with the **Fields** array, describes which form fields are included in the object digest (and hence do not allow changes after the signature is applied). Defined values are: All — All form fields. Include — Only those form fields that are specified in **Fields**. Exclude — Only those form fields that are not specified in **Fields**.
Fields	array	*(Required if **Action** is Include or Exclude)* An array of strings containing field names.
V	number	*(Optional)* The transform parameters dictionary version number. The value for PDF 1.5 is 1.0. Default value: 1.0.

In documents intended for form field workflows:

- The author specifies that form fields can be filled in without invalidating the author's signature; the **P** entry of the **DocMDP** transform parameters dictionary is set to either 2 or 3 (see Table 8.95).
- The author may also specify that any modifications to specific form fields, after a specific recipient has signed the document, should invalidate that recipient's signature. There is a separate signature field for each designated recipient, each having an associated signature field lock dictionary (see Table 8.73) specifying the form fields that should be locked for that user.

- When the recipient signs the field, the signature, signature reference and transform parameters dictionaries are created. The **Action** and **Fields** entries in the transform parameters dictionary are copied from the corresponding fields in the signature field lock dictionary.

  Note: *This copying is done because all objects in a signature dictionary must be direct objects, if the dictionary contains a byte range signature. (Even though* ***FieldMDP*** *signatures are object signatures, any signature dictionary referred to from a signature field must also have a byte range signature.) Therefore the transform parameters dictionary cannot reference the signature field lock dictionary indirectly.*

The object digest is computed over all the form fields specified by the transform parameters dictionary, sorted in alphabetical order (see Appendix I for details). The specified form fields are locked to prevent changes by marking them read-only. Any changes to the form fields can be detected when the recipient's signature is verified.

Identity

The **Identity** transform method is used when computing an object digest that is all-inclusive; that is, no objects are excluded. The entire object tree is walked, starting with the object specified by **Data** in the signature reference dictionary (see Table 8.94). Any changes to the contents of the object invalidate the signature. This method is used to support to support the signing of FDF files; the FDF catalog is the object over which the digest is calculated.

8.7.3 Permissions

The **Perms** entry in the document catalog (see Table 3.25) specifies a *permissions dictionary (PDF 1.5)*. Each entry in this dictionary (see Table 8.98 for the currently defined entries) specifies the name of a permission handler that controls access permissions for the document. These permissions are similar to those defined by security handlers (see Table 3.20 on page 98), but do not require that the document be encrypted. In order for a permission (for example, the ability to fill in form fields) to be actually granted for a document, it must be allowed by each permission handler as well as the security handler.

TABLE 8.98 Entries in a permissions dictionary

KEY	TYPE	VALUE
DocMDP	dictionary	*(Optional)* An indirect reference to a signature dictionary (see Table 8.93). This dictionary must contain a **Reference** entry that is a signature reference dictionary (see Table 8.94) which has a **DocMDP** transform method (see "DocMDP" on page 666) and corresponding transform parameters. Viewer applications should behave as follows: • If this entry is present, the transform parameters indicate the modifications to the document that should be detected and prevented. If the associated signature is invalid, no modifications to the document should be allowed. • If this entry is absent, all operations should be permitted.
UR	dictionary	*(Optional)* A signature dictionary that is used to specify and validate additional capabilities (usage rights) granted for this document; that is, the enabling of interactive features of the viewer application that are not available by default. For example, Adobe Reader does not permit saving documents by default, but Adobe Systems may grant permissions that enable saving in Reader for specific documents. The signature is used to validate that the permissions have been granted by Adobe Systems. The signature dictionary must contain a **Reference** entry that is a signature reference dictionary which has a **UR** transform method (see "UR" on page 667). The transform parameter dictionary for this method indicates which additional permissions should be granted for the document. If the signature is valid, the Adobe Reader allows the specified permissions for the document, in addition to the application's default permissions.

8.7.4 Legal Content Attestations

The PDF language provides a number of capabilities that can make the content of a PDF document vary depending on time or system environment. These capabilities could potentially be used to construct a document that misleads the recipient of a document, intentionally or unintentionally. These situations are relevant when considering the legal implications of a signed PDF document.

Therefore, it is necessary to have a mechanism by which a document recipient can determine whether the document can be trusted. The primary method is to accept only documents that contain author signatures (one that has a **DocMDP** signature that defines what is allowed to change in a document; see "DocMDP" on page 666).

When creating author signatures, viewer applications should also create a *legal attestation dictionary*, whose entries are shown in Table 8.99. This dictionary is the value of the **Legal** entry in the document catalog (see Table 3.25). Its entries specify all content that may result in unexpected rendering of the document contents. The author may provide further clarification of such content by means of the **Attestation** entry. Reviewers should establish for themselves that they trust the author and contents of the document. In the case of a legal challenge to the document, any questionable content can be reviewed in the context of the information in this dictionary.

TABLE 8.99 Entries in a legal attestation dictionary

KEY	TYPE	VALUE
JavaScriptActions	integer	*(Optional)* The number of JavaScript actions found in the document (see "JavaScript Actions" on page 645).
LaunchActions	integer	*(Optional)* The number of launch actions found in the document (see "Launch Actions" on page 600).
URIActions	integer	*(Optional)* The number of URI actions found in the document (see "URI Actions" on page 602).
MovieActions	integer	*(Optional)* The number of movie actions found in the document (see "Movie Actions" on page 605).
SoundActions	integer	*(Optional)* The number of sound actions found in the document (see "Sound Actions" on page 604).
HideAnnotationActions	integer	*(Optional)* The number of hide actions found in the document (see "Hide Actions" on page 606).
GoToRemote	integer	*(Optional)* The number of remote go-to actions found in the document (see "Remote Go-To Actions" on page 599).
AlternateImages	integer	*(Optional)* The number of alternate images found in the document (see "Alternate Images" on page 310)
ExternalStreams	integer	*(Optional)* The number of external streams found in the document.

KEY	TYPE	VALUE
TrueTypeFonts	integer	*(Optional)* The number of TrueType fonts found in the document (see "TrueType Fonts" on page 380).
ExternalRefXobjects	integer	*(Optional)* The number of reference XObjects found in the document (see "Reference XObjects" on page 325).
ExternalOPIdicts	integer	*(Optional)* The number of OPI dictionaries found in the document (see "Open Prepress Interface (OPI)" on page 850).
NonEmbeddedFonts	integer	*(Optional)* The number of non-embedded fonts found in the document (see Section 5.8, "Embedded Font Programs")
DevDepGS_OP	integer	*(Optional)* The number of references to the graphics state parameter **OP** found in the document (see Table 4.8).
DevDepGS_HT	integer	*(Optional)* The number of references to the graphics state parameter **HT** found in the document (see Table 4.8).
DevDepGS_TR	integer	*(Optional)* The number of references to the graphics state parameter **TR** found in the document (see Table 4.8).
DevDepGS_UCR	integer	*(Optional)* The number of references to the graphics state parameter **UCR** found in the document (see Table 4.8).
DevDepGS_BG	integer	*(Optional)* The number of references to the graphics state parameter **BG** found in the document (see Table 4.8).
DevDepGS_FL	integer	*(Optional)* The number of references to the graphics state parameter **FL** found in the document (see Table 4.8).
Annotations	integer	*(Optional)* The number of annotations found in the document (see Section 8.4, "Annotations").
OptionalContent	boolean	*(Optional)* **true** if optional content is found in the document (see Section 4.10, "Optional Content").
Attestation	string	*(Optional)* An attestation, created by the author of the document, explaining the presence of any of the other entries in this dictionary.

CHAPTER 9

Multimedia Features

THIS CHAPTER DESCRIBES those features of PDF that support embedding and playing multimedia content. These capabilities have been greatly enhanced in PDF 1.5, as described in Section 9.1, "Multimedia."

Section 9.2, "Sounds," and Section 9.3, "Movies," describe features that have been supported since PDF 1.2. Section 9.4, "Alternate Presentations," describes a feature that was introduced in PDF 1.4, subsequent to the publication of the third edition of this reference.

9.1 Multimedia

PDF 1.5 introduces a comprehensive set of language constructs to allow the following capabilities:

- Arbitrary media types can be embedded in PDF files. (See implementation note 127 in Appendix H for a list of media types that are recommended for use with Acrobat 6.0 viewers).

- Embedded media, as well as referenced media outside a PDF file, can be played with a variety of player software. (In some situations, the player software may be the viewer application itself.)

  ***Note:** The term "playing" can be used with a wide variety of media, and is not restricted to audio or video. For example, it may be applied to static images such as JPEGs.*

- Media objects may have multiple *renditions*, which can be chosen at play-time based on considerations such as available bandwidth.

- Document authors can control play-time requirements, such as which player software should be used to play a given media object.
- Media objects can be played in various ways; for example, in a floating window as well as in a region on a page.
- Future extensions to the media constructs can be handled in an appropriate manner by current viewer applications; authors can control how old viewers treat future extensions.
- Document authors can adapt the use of multimedia to accessibility requirements.
- On-line media objects can be played efficiently, even when very large.

The following list summarizes the new multimedia features and indicates where each feature is discussed.

- Section 9.1.1, "Viability," describes the rules for determining when media objects are suitable for playing on a particular system.
- Rendition actions (see "Rendition Actions" on page 609) are used to begin the playing of multimedia content.
- A rendition action associates a screen annotation (see "Screen Annotations" on page 588) with a rendition (see Section 9.1.2, "Renditions").
- Renditions are of two varieties: media renditions (see "Media Renditions" on page 682) that define the characteristics of the media to be played, and selector renditions (see "Selector Renditions" on page 683) that enables choosing which of a set of media renditions should be played.
- Media renditions contain entries that specify what should be played (see Section 9.1.3, "Media Clip Objects"), how it should be played (see Section 9.1.4, "Media Play Parameters"), and where it should be played (see Section 9.1.5, "Media Screen Parameters").
- Section 9.1.6, "Other Multimedia Objects," describes several PDF objects that are referenced by the major ones listed above.

Note: *Some of the features described in the following sections have references to corresponding elements in the Synchronized Multimedia Integration Language (SMIL 2.0) standard (see the Bibliography).*

9.1.1 Viability

When playing multimedia content, the viewer application must often make decisions such as which player software and which options (for example, volume and duration) to use. In making these decisions, the viewer must determine the *viability* of the objects used. If an object is considered non-viable, the media should not be played; if it is viable, it should be played, though possibly under less than optimum conditions.

There are several entries in the multimedia object dictionaries whose values have an effect on viability. In particular, some of the object dictionaries define two entries that divide options into one of two categories:

- **MH** *("must honor")*: the options specified by this entry must be honored; otherwise, the containing object is considered non-viable.
- **BE** *("best effort")*: an attempt should be made to honor the options; however, if they cannot be honored, the containing object is still considered viable.

MH and **BE** are both dictionaries, and the same entries are defined for both of them. In any dictionary where these entries are allowed, both may be present, or only one, or neither. For example, the media play parameters dictionary (see Table 9.14) allows the playback volume to be set via the **V** entry in its **MH** and **BE** dictionaries (see Table 9.15). If the specified volume cannot be honored, the object is considered non-viable if **V** is in the **MH** dictionary, and playback should not occur. If **V** is in the **BE** dictionary (and not also in the **MH** dictionary), playback should still occur: the playing software attempts to honor the specified option as best it can.

Using this mechanism, authors can specify minimum requirements (**MH**) and preferred options (**BE**). They can also specify how entries that are added in the future to the multimedia dictionaries will be interpreted by old viewer applications. If an entry that is unrecognized by the viewer is in the **MH** dictionary, the object is considered non-viable; if in a **BE** dictionary, the entry is ignored and viability is unaffected. Unless otherwise stated, an object should be considered non-viable if its **MH** dictionary contains an unrecognized key or an unrecognized value for a recognized key.

The following rules apply to the entries in **MH** and **BE** dictionaries, which behave somewhat differently from other PDF dictionaries:

- If an entry is required, the requirement is met if the entry is present in either the **MH** or **BE** dictionary.
- If an optional entry is not present in either dictionary, it is considered to be present with its default value (if one is defined) in the **BE** dictionary.
- If an instance of the same entry is present in both **MH** and **BE**, the instance in the **BE** dictionary is ignored, unless otherwise specified.
- If the value of an entry in an **MH** or **BE** dictionary is a dictionary or array, it is treated as an atomic unit when determining viability. That is, all entries within the dictionary or array must be honored for the containing object to be viable.

Note: *When determining whether entries can be honored, it is not required that each one be evaluated independently, since they may be dependent on one another. That is, a viewer application or player may examine multiple entries at once (even within different dictionaries) to determine whether their values can be honored.*

The following media objects have **MH** and **BE** dictionaries. They function as described above, except where noted in the individual sections:

- Rendition (Table 9.2)
- Media clip data (Table 9.11)
- Media clip section (Table 9.13)
- Media play parameters (Table 9.15)
- Media screen parameters (Table 9.18)

9.1.2 Renditions

There are two types of *rendition* objects:

- A *media rendition* (see “Media Renditions” on page 682) is a basic media object that specifies what to play, how to play it, and where to play it.
- A *selector rendition* (see “Selector Renditions” on page 683) contains an ordered list of renditions. This list may include other selector renditions, in effect resulting in a tree whose leaves are media renditions. The viewer application

should play the first viable media rendition it encounters in the tree (see Section 9.1.1, “Viability”).

Table 9.1 shows the entries common to all rendition dictionaries. The **N** entry in a rendition dictionary specifies a name that can be used to access the rendition object by means of name tree lookup (see Table 3.28 on page 124). JavaScript actions (see “JavaScript Actions” on page 645), for example, use this mechanism. Since the values referenced by name trees must be indirect objects, it is recommended that all rendition objects be indirect objects.

Note: *A rendition dictionary is not required to have a name tree entry. When it does, the viewer application should ensure that the name specified in the tree is kept the same as the value of the* **N** *entry, for example if the user interface allows the name to be changed. It is recommended (but not required) that a document not contain multiple renditions with the same name.*

The **MH** and **BE** entries are dictionaries whose entries may be present in one or the other of them, as described in Section 9.1.1, “Viability”. For renditions, these dictionaries have a single entry **C** (see Table 9.2), whose value is a *media criteria dictionary*, which specifies a set of criteria that must be met in order for the rendition to be considered viable (see Table 9.3).

The media criteria dictionary behaves somewhat differently than other **MH**/**BE** entries as described in Section 9.1.1. The criteria specified by all of its entries must be met, regardless of whether they are in an **MH** or a **BE** dictionary. The only exception is that if an entry in a **BE** dictionary is *unrecognized* by the viewer application, it does not affect the viability of the object. If a media criteria dictionary is present in both **MH** and **BE**, the entries in both dictionaries are individually evaluated, with **MH** taking precedence (corresponding **BE** entries are ignored).

TABLE 9.1 Entries common to all rendition dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that dictionary describes; if present, must be **Rendition** for a rendition object.
S	name	*(Required)* The type of rendition that this dictionary describes. May be **MR** for media rendition or **SR** for selector rendition. The rendition is considered non-viable if the viewer application does not recognize the value of this entry.

KEY	TYPE	VALUE
N	text string	*(Optional)* A Unicode-encoded text string specifying name of the rendition, for use in a user interface and for name tree lookup by JavaScript actions.
MH	dictionary	*(Optional)* A dictionary whose entries (see Table 9.2) must be honored for the rendition to be considered viable.
BE	dictionary	*(Optional)* A dictionary whose entries (see Table 9.2) need only be honored in a "best effort" sense.

TABLE 9.2 Entries in a rendition MH/BE dictionary

KEY	TYPE	VALUE
C	dictionary	*(Optional)* A *media criteria* dictionary (see Table 9.3). ***Note:*** *The media criteria dictionary behaves somewhat differently than other **MH**/**BE** entries as described in Section 9.1.1, "Viability." The criteria specified by all of its entries must be met, regardless of whether it is in an **MH** or a **BE** dictionary. The only exception is that if an entry in a **BE** dictionary is* unrecognized *by the viewer application, it does not affect the viability of the object.*

TABLE 9.3 Entries in a media criteria dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaCriteria** for a media criteria dictionary.
A	boolean	*(Optional)* If specified, the value of this entry must match the user's preference for whether or not to hear audio descriptions in order for this object to be viable. Equivalent to SMIL's *systemAudioDesc* attribute.
C	boolean	*(Optional)* If specified, the value of this entry must match the user's preference for whether or not to see text captions in order for this object to be viable. Equivalent to SMIL's *systemCaptions* attribute.
O	boolean	*(Optional)* If specified, the value of this entry must match the user's preference for whether or not to hear audio overdubs in order for this object to be viable.
S	boolean	*(Optional)* If specified, the value of this entry must match the user's preference for whether or not to see subtitles in order for this object to be viable.
R	integer	*(Optional)* If specified, the system's bandwidth (in bits per second) must be greater than or equal to the value of this entry in order for this object to be viable. Equivalent to SMIL's *systemBitrate* attribute.

KEY	TYPE	VALUE
D	dictionary	*(Optional)* A dictionary (see Table 9.4) specifying the minimum bit depth required in order for this object to be viable. Equivalent to SMIL's *systemScreenDepth* attribute.
Z	dictionary	*(Optional)* A dictionary (see Table 9.5) specifying the minimum screen size required in order for this object to be viable. Equivalent to SMIL's *systemScreenSize* attribute.
V	array	*(Optional)* An array of software identifier objects (see "Software Identifier Dictionary" on page 700). If this entry is present and non-empty, the viewer application must be identified by one or more of the objects in the array in order for this object to be viable.
P	array	*(Optional)* An array containing one or two name objects specifying a minimum and optionally a maximum PDF language version, in the same format as the **Version** entry in the document catalog (see Table 3.25). If this entry is present and non-empty, the version of multimedia constructs fully supported by the viewer application must be within the specified range in order for this object to be viable.
L	array	*(Optional)* An array of language identifiers (see "Language Identifiers" on page 807). If this entry is present and non-empty, the language in which the viewer application is running must exactly match a language identifier, or consist only of a primary code that matches the primary code of an identifier, in order for this object to be viable. Equivalent to SMIL's *systemLanguage* attribute.

TABLE 9.4 Entries in a minimum bit depth dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MinBitDepth** for a minimum bit depth dictionary.
V	integer	*(Required)* A positive integer (0 or greater) specifying the minimum screen depth (in bits) of the monitor for the rendition to be viable. A negative value is not allowed.
M	integer	*(Optional)* A monitor specifier (see Table 9.28) that specifies which monitor the value of **V** should be tested against. If the value is unrecognized, the object is not viable. Default value: 0.

TABLE 9.5 Entries in a minimum screen size dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MinScreenSize** for a rendition object.
V	array	*(Required)* An array containing two non-negative integers. The width and height (in pixels) of the monitor specified by M must be greater than or equal to the values of the first and second integers in the array, respectively, in order for this object to be viable.
M	integer	*(Optional)* A monitor specifier (see Table 9.28) that specifies which monitor the value of **V** should be tested against. If the value is unrecognized, the object is not viable. Default value: 0.

Media Renditions

Table 9.6 lists the entries in a media rendition dictionary. Its entries specify what media should be played (**C**), how (**P**) and where (**SP**) it should be played. A media rendition object is viable if and only if the objects referenced by its **C**, **P** and **SP** entries are viable.

C can be omitted only in cases where a referenced player takes no meaningful input. This requires that **P** is present and that its referenced media play parameters dictionary (see Table 9.14) contains a **PL** entry, whose referenced media players dictionary (see "Media Players Dictionary" on page 697) has a non-empty **MU** array or a non-empty **A** array.

TABLE 9.6 Additional entries in a media rendition dictionary

KEY	TYPE	VALUE
C	dictionary	*(Optional)* A *media clip* dictionary (see Section 9.1.3, "Media Clip Objects") that specifies what should be played when the media rendition object is played.
P	dictionary	*(Required if **C** is not present, otherwise optional)* A *media play parameters* dictionary (see Section 9.1.4, "Media Play Parameters") that specifies how the media rendition object should be played. Default value: a media play parameters dictionary whose entries (see Table 9.14) all contain their default values.

KEY	TYPE	VALUE
SP	dictionary	*(Optional)* A *media screen parameters* dictionary (see Section 9.1.5, "Media Screen Parameters") that specifies where the media rendition object should be played. Default value: a media screen parameters dictionary whose entries (see Table 9.17) all contain their default values.

Selector Renditions

A selector rendition specifies an array of rendition objects in its **R** entry (see Table 9.7). The renditions in this array should be ordered by preference, with the most preferred rendition first. At play-time, the renditions in the array are evaluated and the first viable media rendition, if any, is played. If one of the renditions is itself a selector, that selector is evaluated in turn, yielding the equivalent of a depth-first tree search. Note, however, that a selector rendition itself may be non-viable; in this case, none of its associated media renditions are evaluated (in effect, this branch of the tree is skipped).

This mechanism may be used, for example, to specify that a large video clip should be used on high-bandwidth machines and a smaller clip should be used on low-bandwidth machines.

TABLE 9.7 Additional entries specific to a selector rendition dictionary

KEY	TYPE	VALUE
R	array	*(Required)* An array of rendition objects. The first viable media rendition object found in the array, or nested within a selector rendition in the array, should be used. An empty array is legal.

9.1.3 Media Clip Objects

There are two types of media clip objects, determined by the subtype **S**, which can be either **MCD** for media clip data (see "Media Clip Data" on page 684) or **MCS** for media clip section (see "Media Clip Section" on page 687). The entries common to all media clip dictionaries are listed in Table 9.8.

TABLE 9.8 Entries common to all media clip dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaClip** for a media clip dictionary.
S	name	*(Required)* The subtype of media clip that this dictionary describes. May be **MCD** for media clip data (see "Media Clip Data" on page 684)or **MCS** for a media clip section (see "Media Clip Section" on page 687). The media clip is considered non-viable if the viewer application does not recognize the value of this entry.
N	text string	*(Optional)* The name of the media clip, for use in the user interface.

Media Clip Data

A media clip data dictionary defines the data for a media object that can be played. For example, it may reference a URL to a streaming video presentation or a movie embedded in the PDF file. Its entries are listed in Table 9.9.

TABLE 9.9 Additional entries in a media clip data dictionary

KEY	TYPE	VALUE
D	file specification or stream	*(Required)* A full file specification or form XObject that specifies the actual media data.
CT	string	*(Optional; not allowed for form XObjects)* A string identifying the type of data in **D**. The string should conform to the content type specification described in Internet RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* (see the Bibliography).
P	dictionary	*(Optional)* A *media permissions dictionary* (see Table 9.10) containing permissions that control the use of the media data. Default value: a media permissions dictionary containing default values.
Alt	array	*(Optional)* An array that provides alternate text descriptions for the media clip data in case it cannot be played; see "Multi-language Text Arrays" on page 811.
PL	dictionary	*(Optional)* A *media players dictionary* (see "Media Players Dictionary" on page 697) that identifies, among other things, players that are legal and not legal for playing the media. ***Note:** If the media players dictionary is non-viable, the media clip data is non-viable.*

KEY	TYPE	VALUE
MH	dictionary	*(Optional)* A dictionary whose entries (see Table 9.11) must be honored for the media clip data to be considered viable.
BE	dictionary	*(Optional)* A dictionary whose entries (see Table 9.11) need only be honored in a "best effort" sense.

The media clip data object must be considered non-viable if the object referenced by the **D** entry does not contain a **Type** entry, the **Type** entry is unrecognized, or the referenced object is not a dictionary or stream. Note that this excludes the use of simple file specifications (see Section 3.10, "File Specifications").

If **D** references a file specification that has an embedded file stream (see Section 3.10.3, "Embedded File Streams"), the embedded file stream's **Subtype** entry is ignored if present; the media clip data dictionary's **CT** entry identifies the type of data.

If **D** references a form XObject, the associated player is implicitly the viewer application, and the form XObject should be rendered as if it were any other data type. For example, the **F** and **D** entries in the media play parameters dictionary (see Table 9.14) apply to a form XObject just as they do to a QuickTime movie.

For media other than form XObjects, the media clip object must provide enough information to allow a viewer application to locate an appropriate player. This can be done by providing one or both of the following:

- A **CT** entry that specifies the content type of the media (the preferred method). If this entry is present, any player that is selected must support this content type.
- A **PL** entry that specifies one or more players that can be used to play the referenced media. It is highly recommended if **CT** is present. However, see implementation note 128 in Appendix H.

The **P** entry specifies a *media permissions dictionary* (see Table 9.10), which specifies the manner in which the data referenced by the media may be used by a viewer application. These permissions allow authors control over how their data is exposed to operations that could allow it to be copied. If the dictionary contains unrecognized entries or entries with unrecognized values, it should be considered non-viable, and the viewer application should not play the media.

TABLE 9.10 Entries in a media permissions dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaPermissions** for a media permissions dictionary.
TF	string	*(Optional)* A string indicating the circumstances under which it is acceptable to write a temporary file in order to play a media clip. Possible values are: (TEMPNEVER) Never allowed. (TEMPEXTRACT) Allowed only if the document permissions allow content extraction; for example, when bit 5 of the user access permissions (see Table 3.20) is set. (TEMPACCESS) Allowed only if the document permissions allow content extraction, including for accessibility purposes; for example, when bits 5 or 10 of the user access permissions (see Table 3.20) are set, or both. (TEMPALWAYS) Always allowed. Default value: (TEMPNEVER). An unrecognized value is treated as (TEMPNEVER).

The **BU** entry in the media clip data **MH** and **BE** dictionaries (see Table 9.11) specifies a base URL for the media data. Relative URLs in the media (which point to auxiliary files or are used for hyperlinking, for example) should be resolved with respect to the value of **BU**. The following should be noted about the **BU** entry:

- If **BU** is in the **MH** dictionary and the base URL is not honored (for example, the player does not accept base URLs), the media clip data is non-viable.
- Determining the viability of the object does not require checking whether the base URL is valid (for example, that the target host exists).
- Absolute URLs within the media are not affected.
- If the media itself contains a base URL (for example, the <BASE> element is defined in HTML), that value is used in preference to **BU**.
- **BU** is completely independent of and unrelated to the value of the **URI** entry in the document catalog (see Section 3.6.1, "Document Catalog").

TABLE 9.11 Entries in a media clip data MH/BE dictionary

KEY	TYPE	VALUE
BU	string	*(Optional)* An absolute URL to be used as the base URL in resolving any relative URLs found within the media data.

Media Clip Section

A *media clip section* dictionary (see Table 9.12) defines a continuous section of another media clip object (known as the *next-level* media clip object). For example, a media clip section could define a 15-minute segment of a media clip data object representing a two-hour movie. The next-level media clip object, specified by the **D** entry, can be either a media clip data object or another media clip section object. However, the linked list formed by the **D** entries of media clip sections must terminate in a media clip data object. If the next-level media object is non-viable, the media clip section is also non-viable.

TABLE 9.12 Additional entries in a media clip section dictionary

KEY	TYPE	VALUE
D	dictionary	*(Required)* The media clip section or media clip data object (the next-level media object) of which this media clip section object defines a continuous section.
Alt	array	*(Optional)* An array that provides alternate text descriptions for the media clip section in case it cannot be played; see "Multi-language Text Arrays" on page 811.
MH	dictionary	*(Optional)* A dictionary whose entries (see Table 9.13) must be honored for the media clip section to be considered viable.
BE	dictionary	*(Optional)* A dictionary whose entries (see Table 9.13) need only be honored in a "best effort" sense.

The **B** and **E** entries in the media clip section's **MH** and **BE** dictionaries (see Table 9.13) define a subsection of the next-level media object referenced by **D**, by specifying beginning and ending offsets into it. Depending on the media type, the offsets may be specified by time, frames, or markers (see "Media Offset Dictionary" on page 696); **B** and **E** are not required to specify the same type of offset.

The following rules apply to these offsets:

- For media types where an offset makes no sense (such as JPEG images), **B** and **E** are ignored, with no effect on viability.
- When **B** or **E** are specified by time or frames, their value is considered to be relative to the start of the next-level media clip. However, if **E** specifies an offset beyond the end of the next-level media clip, the end value is used instead, and there is no effect on viability.
- When **B** or **E** are specified by markers, there is a corresponding absolute offset into the underlying media clip data object. If this offset is not within the range defined by the next-level media clip (if any), or if the marker is not present in the underlying media clip, the existence of the entry is ignored, and there is no effect on viability.
- If the absolute offset derived from the values of all **B** entries in a media clip section chain is greater than or equal to the absolute offset derived from the values of all **E** entries, an empty range is defined. An empty range is legal.
- Any **B** or **E** entry in a media clip section's **MH** dictionary must be honored at play-time in order for the media clip section to be considered viable. (The entry might not be honored if its value was not viable or if the player did not support its value; for example, the player did not support markers.)
- If a **B** or **E** entry is in a media clip section's **MH** dictionary, all **B** or **E** entries, respectively, at deeper levels (closer to the media clip data), are evaluated as if they were in an **MH** dictionary (even if they are actually within **BE** dictionaries).
- If **B** or **E** entry in a **BE** dictionary cannot be supported, it may be ignored at play-time.

TABLE 9.13 Entries in a media clip section MH/BE dictionary

KEY	TYPE	VALUE
B	dictionary	*(Optional)* A *media offset dictionary* (see "Media Offset Dictionary" on page 696) that specifies the offset into the next-level media object at which the media clip section begins. Default: the start of the next-level media object.
E	dictionary	*(Optional)* A *media offset dictionary* (see "Media Offset Dictionary" on page 696) that specifies the offset into the next-level media object at which the media clip section ends. Default: the end of the next-level media object.

9.1.4 Media Play Parameters

A media play parameters dictionary specifies how a media object should be played. It is referenced from a media rendition (see “Media Renditions” on page 682).

TABLE 9.14 Entries in a media play parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaPlayParams** for a media play parameters dictionary.
PL	dictionary	*(Optional)* A media players dictionary (see “Media Players Dictionary” on page 697) that identifies, among other things, players that are legal and not legal for playing the media. ***Note:*** *If this object is non-viable, the media play parameters dictionary is considered non-viable.*
MH	dictionary	*(Optional)* A dictionary whose entries (see Table 9.13) must be honored for the media play parameters to be considered viable.
BE	dictionary	*(Optional)* A dictionary whose entries (see Table 9.13) need only be honored in a “best effort” sense.

TABLE 9.15 Entries in a media play parameters MH/BE dictionary

KEY	TYPE	VALUE
V	integer	*(Optional)* An integer that specifies the desired volume level as a percentage of recorded volume level. A zero value is equivalent to mute; negative values are illegal. Default value: 100.
C	boolean	*(Optional)* A flag specifying whether to display a player-specific controller user interface (for example, play/pause/stop controls) when playing. Default value: **false**

KEY	TYPE	VALUE
F	integer‘	*(Optional)* The manner in which the player should treat a visual media type that does not exactly fit the rectangle in which it plays. 0 The media's width and height are scaled while preserving the aspect ratio, so the media and play rectangles have the greatest possible intersection while still displaying all media content. Same as "meet" value of SMIL's *fit* attribute. 1 The media's width and height are scaled while preserving the aspect ratio, so the play rectangle is entirely filled, and the amount of media content that does not fit within the play rectangle is minimized. Same as "slice" value of SMIL's *fit* attribute. 2 The media's width and height are scaled independently so the media and play rectangles are the same; the aspect ratio is not necessarily preserved. Same as "fill" value of SMIL's *fit* attribute. 3 The media is not scaled. A scrolling user interface is provided if the media rectangle is wider or taller than the play rectangle. Same as "scroll" value of SMIL's *fit* attribute. 4 The media is not scaled. Only the portions of the media rectangle that intersect the play rectangle are displayed. Same as "hidden" value of SMIL's *fit* attribute. 5 Use the player's default setting (author doesn't care). Default value: 5. An unrecognized value should be treated as the default value if in a **BE** dictionary; if in an **MH** dictionary, the object should be considered non-viable.
D	dictionary	*(Optional)* A media duration dictionary (see Table 9.16). Default value: a dictionary specifying the intrinsic duration (see below).
A	boolean	*(Optional)* If **true**, the media should automatically play when activated. If **false**, the media should be initially paused when activated (for example, the first frame is displayed). Relevant only for media that may be paused. Default value: **true**.
RC	number	*(Optional)* Specifies the number of iterations of the duration **D** to repeat; similar to SMIL's *repeatCount* attribute. Zero means repeat forever. Negative values are illegal; non-integral values are legal. Default value: 1.0.

The value of the **D** entry is a *media duration dictionary*, whose entries are shown in Table 9.16. It specifies a temporal duration (which corresponds to the notion of a *simple duration* in SMIL). The duration may be a specific amount of time; it may be infinity; or it may be the media's *intrinsic duration* (for example, the intrinsic duration of a two-hour QuickTime movie is two hours). The intrinsic duration may be modified when a media clip section (see “Media Clip Section” on page 687) is used: the intrinsic duration is the difference between the absolute begin and end offsets. For a media type having no notion of time (such as a JPEG image), the duration is considered to be infinity.

If the simple duration is longer than the intrinsic duration, the player should freeze the media in its final state until the simple duration has elapsed. For visual media types, the last appearance (frame) would be displayed. For aural media types, the media is logically frozen, but should not continue to produce sound.

Note: *In this case, the* ***RC*** *entry, which specifies a repeat count, applies to the simple duration, so that the entire play-pause sequence is repeated* ***RC*** *times.*

TABLE 9.16 Entries in a media duration dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaDuration** for a media duration dictionary.
S	name	*(Required)* The subtype of media duration dictionary. Allowed values are: **I** The duration is the intrinsic duration of the associated media **F** The duration is infinity **T** The duration is specified by the **T** entry The media duration dictionary is considered non-viable if the viewer application does not recognize the value of this entry.
T	dictionary	*(Required if the value of* ***S*** *is* ***T****; otherwise ignored)* A timespan dictionary specifying an explicit duration (see Table 9.24). A negative duration is illegal.

9.1.5 Media Screen Parameters

A media screen parameters dictionary (see Table 9.17) specifies where a media object should be played. It contains **MH** and **BE** dictionaries (see Table 9.18) which function as discussed in Section 9.1.1, “Viability.” All media clips that are being played are associated with a particular document and must be stopped when the document is closed.

Note: *It is recommended that viewer applications disallow floating windows and full-screen windows unless specifically allowed by the user. The reason is that document-based security attacks are possible if windows containing arbitrary media con-*

tent can be displayed without indicating to the user that the window is merely hosting a media object. This recommendation may be relaxed if it is possible to communicate the nature of such windows to the user, for example with text in a floating window's title bar.

TABLE 9.17 Entries in a media screen parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaScreenParams** for a media screen parameters dictionary.
MH	dictionary	*(Optional)* A dictionary whose entries (see Table 9.18) must be honored for the media screen parameters to be considered viable.
BE	dictionary	*(Optional)* A dictionary whose entries (see Table 9.18) need only be honored in a "best effort" sense.

TABLE 9.18 Entries in a media screen parameters MH/BE dictionary

KEY	TYPE	VALUE
W	integer	*(Optional)* The type of window that the media object should play in: 0 A floating window 1 A full-screen window that obscures all other windows 2 A hidden window 3 The rectangle occupied by the screen annotation (see "Screen Annotations" on page 588) associated with the media rendition. Default value: 3. Unrecognized value in **MH**: object is non-viable; in **BE**: treat as default value.

KEY	TYPE	VALUE
B	array	*(Optional)* An array of 3 numbers in the range 0.0 to 1.0 specifying the components in the **DeviceRGB** color space of the background color for the rectangle in which the media is being played. This color is used if the media object does not entirely cover the rectangle or if it has transparent sections. Ignored for hidden windows. Default value: implementation-defined. The viewer application should choose a reasonable value based on the value of **W**; for example, a system default background color for floating windows or a user-preferred background color for full-screen windows. ***Note:*** *If a media format has an intrinsic background color,* ***B*** *does not override it. However, the* ***B*** *color is visible if the media has transparent areas or otherwise does not cover the entire window.*
O	number	*(Optional)* A number in the range 0.0 to 1.0 specifying the constant opacity value to be used in painting the background color specified by **B**. A value below 1.0 means the window is transparent; for example, windows behind a floating window will show through if the media does not cover the entire floating window. A value of 0.0 indicates full transparency and makes **B** irrelevant. Ignored for full-screen and hidden windows. Default value: 1.0 (fully opaque).
M	integer	*(Optional)* A monitor specifier (see Table 9.28) that specifies which monitor in a multi-monitor system a floating or full-screen window should appear on. Ignored for other types. Default value: 0 (document monitor). Unrecognized value in **MH**: object is non-viable; in **BE**: treat as default value.
F	dictionary	*(Required if the value of* ***W*** *is 0; otherwise ignored)* A floating window parameters dictionary (see Table 9.19) specifying the size, position and options used in displaying floating windows.

The **F** entry in the media screen parameters **MH**/**BE** dictionaries is a floating window parameters dictionary, whose entries are listed in Table 9.19. The entries in the floating window parameters dictionary are treated as if they were present in the **MH** or **BE** dictionaries that they are referenced from. That is, the contained entries are individually evaluated for viability, rather than the dictionary being evaluated as a whole. (There may be an **F** entry in both **MH** and **BE**. In such a case, if a given entry is present in both floating window parameters dictionaries, the one in the **MH** dictionary takes precedence.)

The **D**, **P** and **RT** entries are used to specify the rectangle that the floating window occupies. Once created, the floating window's size and position are not tied to any other window, even if the initial size or position was computed relative to other windows.

Unrecognized values for the **R**, **P**, **RT** and **O** entries are handled as follows: if they are nested within an **MH** dictionary, the floating window parameters object (and hence the media screen parameters object) must be considered non-viable; if they are nested within a **BE** dictionary, they should be considered to have their default values.

TABLE 9.19 Entries in a floating window parameters dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **FWParams** for a floating window parameters dictionary.
D	array	*(Required)* An array containing two non-negative integers representing the floating window's width and height, in pixels, respectively. These values correspond to the dimensions of the rectangle in which the media will play, not including such items as title bar and resizing handles.
RT	integer	*(Optional)* The window relative to which the floating window should be positioned: 0 The document window 1 The application window 2 The full virtual desktop 3 The monitor specified by **M** in the media screen parameters **MH** or **BE** dictionary (see 9.22) Default value: 0.

KEY	TYPE	VALUE
P	integer	*(Optional)* The location where the floating window (including such items as title bar and resizing handles) should be positioned relative to the window specified by **RT**: 0 Top left corner 1 Top center 2 Top right corner 3 Center left 4 Center 5 Center right 6 Bottom left corner 7 Bottom center 8 Bottom right corner Default value: 4.
O	integer	*(Optional)* Specifies what should occur if the floating window is positioned totally or partially offscreen (that is, not visible on any physical monitor): 0 Take no special action 1 Move and/or resize the window so that it is onscreen 2 Consider the object to be non-viable Default value: 1
T	boolean	*(Optional)* If **true**, the floating window should have a title bar. Default value: **true**.
UC	boolean	*(Optional; meaningful only if **T** is **true**)* If **true**, the floating window should include user interface elements that allow a user to close a floating window. Default value: **true**
R	integer	*(Optional)* Specifies whether the floating window may be resized by a user: 0 May not be resized 1 May be resized only if aspect ratio is preserved. 2 May be resized without preserving aspect ratio. Default value: 0.
TT	array	*(Optional; meaningful only if **T** is **true**)* An array providing text to display on the floating window's title bar. See "Multi-language Text Arrays" on page 811. If this entry is not present, the viewer application may provide default text.

Media Offset Dictionary

A *media offset dictionary* (Table 9.20) specifies an offset into a media object. The **S** (subtype) entry indicates how the offset is specified: in terms of time (for example, "10 seconds"), frames (for example, "frame 20") or markers (for example, "Chapter One"). Different media types support different types of offsets.

TABLE 9.20 Entries common to all media offset dictionaries

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaOffset** for a media offset dictionary.
S	name	*(Required)* The subtype of media offset dictionary. Possible values are: T — A media offset time dictionary (see Table 9.21) F — A media offset frame dictionary (see Table 9.22) M — A media offset marker dictionary (see Table 9.23) The rendition is considered non-viable if the viewer application does not recognize the value of this entry.

TABLE 9.21 Additional entries in a media offset time dictionary

KEY	TYPE	VALUE
T	dictionary	*(Required)* A timespan dictionary (see Table 9.24) that specifies a temporal offset into a media object. Negative timespans are not allowed in this context. The media offset time dictionary is non-viable if its timespan dictionary is non-viable.

TABLE 9.22 Additional entries in a media offset frame dictionary

KEY	TYPE	VALUE
F	integer	*(Required)* Specifies a frame within a media object. Frame numbers begin at 0; negative frame numbers are not allowed.

TABLE 9.23 Additional entries in a media offset marker dictionary

KEY	TYPE	VALUE
M	text string	*(Required)* A text string that identifies a named offset within a media object.

Timespan Dictionary

A *timespan* dictionary specifies a length of time; its entries are shown in Table 9.24.

TABLE 9.24 Entries in a timespan dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Timespan** for a timespan dictionary.
S	name	*(Required)* The subtype of timespan dictionary. The only value currently allowed is **S** (simple timespan). The rendition is considered non-viable if the viewer application does not recognize the value of this entry.
V	number	*(Required)* The number of seconds in the timespan. Non-integral values are allowed. Negative values are allowed, but may be disallowed in some contexts (all situations defined in PDF 1.5 disallow negative values). ***Note:*** *This entry is required only if the value of the* ***S*** *entry is* ***S****. Subtypes defined in the future need not use this entry.*

9.1.6 Other Multimedia Objects

This section defines several dictionary types that are referenced by the previous sections.

Media Players Dictionary

A media players dictionary can be referenced by media clip data (see "Media Clip Data" on page 684) and media play parameters (see Section 9.1.4, "Media Play Parameters") dictionaries, and allows them to specify which players may or may not be used to play the associated media. The media players dictionary references *media player info dictionaries* (see "Media Player Info Dictionary," below) which provide specific information about each player.

TABLE 9.25 Entries in a media players dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaPlayers** for a media players dictionary.
MU	array	*(Optional)* An array of media player info objects (see Table 9.26) that specify a set of players, one of which *must* be used in playing the associated media object. ***Note:*** *Any players specified in* ***NU*** *are effectively removed from* ***MU****. (For example, if* ***MU*** *specifies versions 1 through 5 of a player, while* ***NU*** *specifies versions 1 and 2 of the same player,* ***MU*** *is effectively versions 3 through 5.)*
A	array	*(Optional)* An array of media player info objects (see Table 9.26) that specify a set of players, any of which *may* be used in playing the associated media object. If **MU** is also present and non-empty, **A** is ignored.
NU	array	*(Optional)* An array of media player info objects (see Table 9.26) that specify a set of players that *must not* be used in playing the associated media object (even if they are also specified in **MU**).

The **MU**, **A** and **NU** entries each specify one or more media player info objects. (An empty array is treated as if it is not present.) The media player info objects are allowed to specify overlapping player ranges (for example, **MU** could contain a media player info dictionary describing versions 1 to 10 of "Player X" and another describing versions 3 through 5 of "Player X").

If a non-viable media player info object is referenced by **MU**, **NU** or **A**, it is treated as if it were not present in its original array, and a media player info object containing the same software identifier dictionary (see "Software Identifier Dictionary" on page 700) is logically considered to be present in **NU**. The same rule applies to a media player info object that contains a "partially unrecognized" software identifier dictionary.

Since both media clip data and media play parameters dictionaries can be employed in a play operation, and each can reference a media players dictionary, there is a potential for conflict between the contents of the two media players dictionaries. At play-time, the viewer should use the following algorithm to determines whether a player present on the machine can be employed. The player cannot be used if any of the following conditions are true:

Algorithm 9.1

1. The content type is known and the player does not support the type.
2. The player is found in the **NU** array of either dictionary.
3. Both dictionaries have non-empty **MU** arrays and the player is not found in both of them, or if only one of the dictionaries has a non-empty **MU** array and the player is not found in it.
4. Neither dictionary has a non-empty **MU** array, the content type is not known and the player is not found in the **A** array of either dictionary.

If none of the conditions are true, the player can be used.

Note: *A player is "found" in the **NU**, **MU** or **A** arrays if it matches the information found in the PID entry of one of the entries, as described by Algorithm 9.2.*

Media Player Info Dictionary

A media player info dictionary provides a variety of information regarding a specific media player. Its entries (see Table 9.26) allow information to be associated with a particular version of a player, or with a range of versions. As of PDF 1.5, only the **PID** entry provides information about the player, as described in the next section, "Software Identifier Dictionary".

TABLE 9.26 Entries in a media player info dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **MediaPlayerInfo** for a media player info dictionary.
PID	dictionary	*(Required)* A software identifier object (see "Software Identifier Dictionary," below) that specifies the player name, versions and operating systems to which this media player info object applies.
MH	dictionary	*(Optional)* A dictionary containing entries that must be honored for this object to be considered viable ***Note:*** *Currently, there are no defined entries for this dictionary*
BE	dictionary	*(Optional)* A dictionary containing entries that need only be honored in a "best effort" sense. ***Note:*** *Currently, there are no defined entries for this dictionary*

Software Identifier Dictionary

A software identifier dictionary allows software to be identified by name, range of versions and operating systems; its entries are listed in Table 9.27. A viewer application uses this information to determine whether a given media player can be used in a given situation. If the dictionary contains keys that are unrecognized by the viewer application, it is considered to be “partially recognized.” The viewer application may or may not decide to treat the software identifier as viable, depending on the context in which it is used.

The following procedure is used to determine whether a piece of software is considered to match a software identifier object:

Algorithm 9.2

1. The software name must match the name specified by the **U** entry (see “Software URIs,” below).
2. The software version must be within the interval specified by the **L**, **H**, **LI** and **H1** entries (see “Version arrays,” below).
3. The machine’s operating system name must be an exact match for one present in the **OS** array. If the array is not present or empty, a match is also considered to exist.

TABLE 9.27 Entries in a software identifier dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SoftwareIdentifier** for a software identifier dictionary.
U	string	*(Required)* A URI that identifies a piece of software (see “Software URIs,” below).
L	array	*(Optional)* The lower bound of the range of software versions that this software identifier object specifies (see “Version arrays,” below). Default value: the array [0].
LI	boolean	*(Optional)* If **true**, the lower bound of the interval defined by **L** and **H** is inclusive; that is, the software version must be greater than or equal to **L** (see “Version arrays,” below). If **false**, it is not inclusive. Default value: **true**.
H	array	*(Optional)* The upper bound of the range of software versions that this software identifier object specifies (see “Version arrays,” below). Default value: an empty array [].

KEY	TYPE	VALUE
HI	boolean	*(Optional)* If **true**, the upper bound of the interval defined by **L** and **H** is inclusive; that is, the software version must be less than or equal to **H** (see "Version arrays," below).If **false**, it is not inclusive. Default value: **true**.
OS	array	*(Optional)* An array of strings representing operating system identifiers that indicate which operating systems this object applies to. The defined values are the same as those defined for SMIL 2.0's *systemOperatingSystem* attribute. There may not be multiple copies of the same identifier in the array. An empty array is considered to represent all operating systems. Default value: an empty array.

Software URIs

The **U** entry is a URI (universal resource identifier) that identifies a piece of software. It is interpreted according to its scheme; the only presently defined scheme is *vnd.adobe.swname*. The scheme name is case-insensitive; if is not recognized by the viewer application, the software must be considered a non-match. The syntax of URIs of this scheme is:

"vnd.adobe.swname:" *software_name*

where *software_name* is equivalent to *reg_name* as defined in Internet RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*; see the Bibliography. *software_name* is considered to be a sequence of UTF-8-encoded characters that have been escaped with one pass of URL escaping (see "URL Strings" on page 821). That is, to recover the original software name, *software_name* must be unescaped and then treated as a sequence of UTF-8 characters. The actual software names must be compared in a case-sensitive fashion.

Software names are second-class names (see Appendix E). For example, the URI for Adobe Acrobat is

vnd.adobe.swname:ADBE_Acrobat

Version arrays

The **L**, **H**, **LI** and **HI** entries are used to specify a range of software versions. **L** and **H** are *version arrays* containing zero or more non-negative integers representing sub-version numbers; the first integer is the major version numbers, and subsequent integers are increasingly minor. **H** must be greater than or equal to **L**, according to the following rules for comparing version arrays:

Algorithm 9.3 ***Comparing version arrays***

1. An empty version array is treated as infinity; that is, it is considered greater than any other version array except another empty array. Two empty arrays are equal.
2. When comparing arrays that contain different numbers of elements, the smaller array is implicitly padded with zero-valued integers to make the number of elements equal. For example, when comparing [5 1 2 3 4] to [5], the latter is treated as [5 0 0 0 0].
3. The corresponding elements of the arrays are compared, starting with the first. When a difference is found, the array containing the larger element is considered to have the larger version number. If no differences are found, the versions are equal.

Note: *If a version array contains negative numbers, it is considered non-viable, as is the enclosing software identifier.*

Monitor Specifier

A monitor specifier is an integer that identifies a physical monitor attached to a system. It can have one of the following values:

TABLE 9.28 Monitor specifier values

VALUE	DESCRIPTION
0	The monitor containing the largest section of the document window
1	The monitor containing the smallest section of the document window
2	Primary monitor. If no monitor is considered primary, use case 0
3	Monitor with the greatest color depth
4	Monitor with the greatest area (in pixels squared)
5	Monitor with the greatest height (in pixels)
6	Monitor with the greatest width (in pixels)

For some of these values, it is possible have a "tie" at play-time; for example, two monitors might have the same color depth. Ties are broken in an implementation-dependent manner.

9.2 Sounds

A *sound object (PDF 1.2)* is a stream containing sample values that define a sound to be played through the computer's speakers. The **Sound** entry in a sound annotation or sound action dictionary (see Tables 8.32 on page 587 and 8.50 on page 604) identifies a sound object representing the sound to be played when the annotation is activated.

Since a sound object is a stream, it can contain any of the standard entries common to all streams, as described in Table 3.4 on page 38. In particular, if it contains an **F** (file specification) entry, then the sound is defined in an external file. This must be a self-describing sound file, containing all information needed to render the sound; no additional information need be present in the PDF file.

Note: *The AIFF, AIFF-C (Mac OS), RIFF (.wav), and snd (.au) file formats are all self-describing.*

If no **F** entry is present, the sound object itself contains the sample data and all other information needed to define the sound. Table 9.29 shows the additional dictionary entries specific to a sound object.

TABLE 9.29 Additional entries specific to a sound object

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **Sound** for a sound object.
R	number	*(Required)* The sampling rate, in samples per second.
C	integer	*(Optional)* The number of sound channels. Default value: 1. (See implementation note 129 in Appendix H.)
B	integer	*(Optional)* The number of bits per sample value per channel. Default value: 8.

KEY	TYPE	VALUE
E	name	*(Optional)* The encoding format for the sample data: Raw Unspecified or unsigned values in the range 0 to $2^{B} - 1$ Signed Twos-complement values muLaw μ-law–encoded samples ALaw A-law–encoded samples Default value: Raw.
CO	name	*(Optional)* The sound compression format used on the sample data. (Note that this is separate from any stream compression specified by the sound object's **Filter** entry; see Table 3.4 on page 38 and Section 3.3, "Filters.") If this entry is absent, then no sound compression has been used; the data contains sampled waveforms to be played at **R** samples per second per channel.
CP	(various)	*(Optional)* Optional parameters specific to the sound compression format used. ***Note:** At the time of publication, no standard values have been defined for the **CO** and **CP** entries.*

Sample values are stored in the stream with the most significant bits first ("big-endian" order for samples larger than 8 bits). Samples that are not a multiple of 8 bits are packed into consecutive bytes, starting at the most significant end. If a sample extends across a byte boundary, the most significant bits are placed in the first byte, followed by less significant bits in subsequent bytes. For dual-channel stereophonic sounds, the samples are stored in an interleaved format, with each sample value for the left channel (channel 1) preceding the corresponding sample for the right (channel 2).

To maximize the portability of PDF documents containing embedded sounds, Adobe recommends that PDF viewer applications and plug-in extensions support at least the following formats (assuming the platform has sufficient hardware and OS support to play sounds at all):

R	8000, 11,025, or 22,050 samples per second
C	1 or 2 channels
B	8 or 16 bits per channel
E	Raw, Signed, or muLaw encoding

If the encoding (**E**) is Raw or Signed, then **R** must be 11,025 or 22,050 samples per channel. If the encoding is muLaw, then **R** must be 8000 samples per channel, **C** must be 1 channel, and **B** must be 8 bits per channel. Sound players should be prepared to convert between formats, downsample rates, and combine channels as necessary to render sound on the target platform.

9.3 Movies

***Note:** The features described in this section are obsolescent and their use is no longer recommended. They are superseded by the general multimedia framework described in Section 9.1, "Multimedia."*

PDF includes the ability to embed *movies* within a document by means of movie annotations (see "Movie Annotations" on page 587). Despite the name, a movie may consist entirely of sound with no visible images to be displayed on the screen. The **Movie** and **A** (activation) entries in the movie annotation dictionary refer, respectively, to a *movie dictionary* (Table 9.30) describing the static characteristics of the movie and a *movie activation dictionary* (Table 9.31) specifying how it should be presented.

TABLE 9.30 Entries in a movie dictionary

KEY	TYPE	VALUE
F	file specification	*(Required)* A file specification identifying a self-describing movie file. ***Note:** The format of a "self-describing movie file" is left unspecified, and there is no guarantee of portability.*
Aspect	array	*(Optional)* The width and height of the movie's bounding box, in pixels, specified as [*width height*]. This entry should be omitted for a movie consisting entirely of sound with no visible images. See implementation note 130 in Appendix H
Rotate	integer	*(Optional)* The number of degrees by which the movie is rotated clockwise relative to the page. The value must be a multiple of 90. Default value: 0.
Poster	boolean or stream	*(Optional)* A flag or stream specifying whether and how to display a *poster image* representing the movie. If this value is a stream, it contains an image XObject (see Section 4.8, "Images") to be displayed as the poster. If it is the boolean value **true**, the poster image should be retrieved from the movie file itself; if it is **false**, no poster should be displayed. See implementation note 131 in Appendix H. Default value: **false**.

TABLE 9.31 Entries in a movie activation dictionary

KEY	TYPE	VALUE
Start	(various)	*(Optional)* The starting time of the movie segment to be played. Movie time values are expressed in units of time based on a *time scale*, which defines the number of units per second; the default time scale is defined in the movie data itself. The starting time is nominally a non-negative 64-bit integer, specified as follows: • If it is representable as an integer (subject to the implementation limit for integers, as described in Appendix C), it should be specified as such. • If it is not representable as an integer, it should be specified as an 8-byte string representing a 64-bit twos-complement integer, most significant byte first. • If it is expressed in a time scale different from that of the movie itself, it is represented as an array of two values: an integer or string denoting the starting time, as above, followed by an integer specifying the time scale in units per second. If this entry is omitted, the movie is played from the beginning.
Duration	(various)	*(Optional)* The duration of the movie segment to be played, specified in the same form as **Start**. If this entry is omitted, the movie is played to the end.
Rate	number	*(Optional)* The initial speed at which to play the movie. If the value of this entry is negative, the movie is played backward with respect to **Start** and **Duration**. Default value: 1.0.
Volume	number	*(Optional)* The initial sound volume at which to play the movie, in the range –1.0 to 1.0. Higher values denote greater volume; negative values mute the sound. Default value: 1.0.
ShowControls	boolean	*(Optional)* A flag specifying whether to display a movie controller bar while playing the movie. Default value: **false**.
Mode	name	*(Optional)* The *play mode* for playing the movie: Once — Play once and stop. Open — Play and leave the movie controller bar open. Repeat — Play repeatedly from beginning to end until stopped. Palindrome — Play continuously forward and backward until stopped. Default value: Once.

KEY	TYPE	VALUE
Synchronous	boolean	*(Optional)* A flag specifying whether to play the movie synchronously or asynchronously. If this value is **true**, the movie player will retain control until the movie is completed or dismissed by the user; if **false**, it will return control to the viewer application immediately after starting the movie. Default value: **false**.
FWScale	array	*(Optional)* The magnification (zoom) factor at which to play the movie. The presence of this entry implies that the movie is to be played in a floating window; if the entry is absent, it will be played in the annotation rectangle. The value of the entry is an array of two positive integers, [*numerator denominator*], denoting a rational magnification factor for the movie. (See implementation note 132 in Appendix H.) The final window size, in pixels, is (*numerator* ÷ *denominator*) × **Aspect** where the value of **Aspect** is taken from the movie dictionary (see Table 9.30).
FWPosition	array	*(Optional)* For floating play windows, the relative position of the window on the screen. The value is an array of two numbers [*horiz vert*] each in the range 0.0 to 1.0, denoting the relative horizontal and vertical position of the movie window with respect to the screen. For example, the value [0.5 0.5] centers the window on the screen. See implementation note 133 in Appendix H. Default value: [0.5 0.5].

9.4 Alternate Presentations

Beginning with PDF 1.4, a PDF document may contain *alternate presentations*, which specify alternate ways in which the document may be viewed. The optional **AlternatePresentations** entry *(PDF 1.4)* in a document's name dictionary (see Table 3.28) contains a name tree that maps name strings to the alternate presentations available for the document.

***Note:** Since PDF viewers are not required to support alternate presentations, authors of documents containing alternate presentations should define the files such that something useful and meaningful can be displayed and printed. For example, if the document contains an alternate presentation slideshow of a sequence of photographs, the photographs should be viewable in a static form by viewers that are not capable of playing the slideshow.*

As of PDF 1.5, the only type of alternate presentation is a *slideshow*. Slideshows are typically invoked by means of JavaScript actions (see "JavaScript Actions" on page 645") initiated by user action on an interactive form element (see Section 8.6, "Interactive Forms"). Implementation note 134 in Appendix H describes Acrobat's implementation of slideshows.

The following table shows the entries in a slideshow dictionary.

TABLE 9.32 Entries in a slideshow dictionary

KEY	TYPE	DESCRIPTION
Type	name	*(Required; PDF 1.4)* The type of PDF object that this dictionary describes; must be **SlideShow** for a slideshow dictionary.
Subtype	name	*(Required; PDF 1.4)* The subtype of the PDF object that this dictionary describes; must be **Embedded** for a slideshow dictionary.
Resources	name tree	*(Required; PDF 1.4)* A name tree that maps name strings to objects referenced by the alternate presentation. ***Note:*** *Even though PDF treats the strings in the name tree as strings without a specified encoding, the slideshow interprets them as UTF-8 encoded Unicode.*
StartResource	string	*(Required; PDF 1.4)* A string that must match one of the strings in the **Resources** entry. It defines the root object for the slideshow presentation.

The **Resources** name tree represents a virtual file system to the slideshow. It associates strings ("filenames") with PDF objects that represent resources used by the slideshow. For example, a root stream might reference a filename, which would be looked up in the **Resources** name tree, and the corresponding object would be loaded as the file. (This virtual file system is flat; that is, there is no way to reference subfolders.)

Typically, images are stored in the document as image XObjects (see Section 4.8.4, "Image Dictionaries"), thereby allowing them to be shared between the standard PDF representation and the slideshow; other media objects are stored or embedded file streams (see Section 3.10.3, "Embedded File Streams"). Also, see Implementation note 135 in Appendix H.

To allow viewers to verify content against the supported features in a particular viewer, it is a requirement that all referenced objects include a **Type** entry in their dictionary, even when the **Type** entry is normally optional for a given object.

The following example illustrates the use of alternate presentation slideshows.

Example 9.1

```
1 0 obj
    <</Type /Catalog
        /Pages 2 0 R
        /Names 3 0 R             % Indirect reference to name dictionary
    >>
...
3 0 obj                          % The name dictionary
    <</AlternatePresentations 4 0 R >>
endobj
4 0 obj                          % The alternate presentations name tree
    <</Names [(MySlideShow) 5 0 R]>>
endobj
5 0 obj                          % The slideshow definition
    <</Type /SlideShow
        /Subtype Embedded
        /Resources <</Names [ (mysvg.svg) 31 0R
            (abc0001.jpg) 35 0 R (abc0002.jpg) 36 0 R
            (mysvg.js) 61 0 R (mymusic.mp3) 65 0 R ]>>
        /StartResource (mysvg.svg)
    >>
...
31 0 obj
    <</Type /Filespec            % The root object, which
        /F (mysvg.svg)           % points to an embedded file stream
        /EF <</F 32 0 R>>
    >>
endobj
32 0 obj                                 % The embedded file stream
    <</Type /EmbeddedFile
        /Subtype /image#2Fsvg+xml
        /Length 72
    >>
    stream
        <?xml version="1.0" standalone="no"?>
        <svg><!-- Some SVG goes here--></svg>
    endstream
endobj

% ... other objects not shown
```

CHAPTER 10

Document Interchange

THE FEATURES DESCRIBED in this chapter do not affect the final appearance of a document. Rather, they allow it to include higher-level information that is useful for the interchange of documents among applications. They include:

- *Procedure sets* (Section 10.1) that define the implementation of PDF operators
- *Metadata* (Section 10.2) consisting of general information about a document or a component of a document, such as its title, author, and creation and modification dates
- *File identifiers* (Section 10.3) for reliable reference from one PDF file to another
- *Page-piece dictionaries* (Section 10.4) allowing an application to embed private data in a PDF document for its own use
- *Marked-content* operators (Section 10.5) for identifying portions of a content stream and associating them with additional properties or externally specified objects
- *Logical structure* facilities (Section 10.6) for imposing a hierarchical organization on the content of a document
- *Tagged PDF* (Section 10.7), a set of conventions for using the marked content and logical structure facilities to facilitate the extraction and reuse of a document's content for other purposes
- Various ways of increasing the *accessibility* of a document to disabled users (Section 10.8), including the identification of the natural language in which it is written (such as English or Spanish) for the benefit of a text-to-speech engine
- The *Web Capture* plug-in extension (Section 10.9), which creates PDF files from Internet-based or locally resident HTML, PDF, GIF, JPEG, and ASCII text files

- Facilities supporting prepress production workflows (Section 10.10), such as the specification of *page boundaries* and the generation of *printer's marks*, *color separations*, *output intents*, *traps*, and low-resolution *proxies* for high-resolution images

10.1 Procedure Sets

The PDF operators used in content streams are grouped into categories of related operators called *procedure sets* (see Table 10.1). Each procedure set corresponds to a named resource containing the implementations of the operators in that procedure set. The **ProcSet** entry in a content stream's resource dictionary (see Section 3.7.2, "Resource Dictionaries") holds an array consisting of the names of the procedure sets used in that content stream. These procedure sets are used only when the content stream is printed to a PostScript output device; the names identify PostScript procedure sets that must be sent to the device to interpret the PDF operators in the content stream. Each element of this array must be one of the predefined names shown in Table 10.1. (See implementation note 136 in Appendix H.)

TABLE 10.1 Predefined procedure sets

NAME	CATEGORY OF OPERATORS
PDF	Painting and graphics state
Text	Text
ImageB	Grayscale images or image masks
ImageC	Color images
ImageI	Indexed (color-table) images

Note: *Beginning with PDF 1.4, this feature is considered obsolete. For compatibility with existing viewer applications, PDF producer applications should continue to specify procedure sets (preferably all of those listed in Table 10.1, unless it is known that fewer are needed). However, viewer applications should not depend on the correctness of this information.*

10.2 Metadata

A PDF document may include general information such as the document's title, author, and creation and modification dates. Such global information about the document itself (as opposed to its content or structure) is called *metadata*, and is intended to assist in cataloguing and searching for documents in external databases. A document's metadata may also be added or changed by users or plug-in extensions (see implementation note 137 in Appendix H). Beginning with PDF 1.4, metadata can also be specified for individual components of a document.

Metadata can be stored in a PDF document in either of the following ways:

- In a *document information dictionary* associated with the document (Section 10.2.1)
- In a *metadata stream (PDF 1.4)* associated with the document or a component of the document (Section 10.2.2)

10.2.1 Document Information Dictionary

The optional **Info** entry in the trailer of a PDF file (see Section 3.4.4, "File Trailer") can hold a *document information dictionary* containing metadata for the document; Table 10.2 shows its contents. Any entry whose value is not known should be omitted from the dictionary, rather than included with an empty string as its value.

Some plug-in extensions may choose to permit searches on the contents of the document information dictionary. To facilitate browsing and editing, all keys in the dictionary are fully spelled out, not abbreviated. New keys should be chosen with care so that they make sense to users.

The value associated with any key not specifically mentioned in Table 10.2 must be a text string.

Note: *Although viewer applications can store custom metadata in the document information dictionary, it is inappropriate to store private content or structural information there; such information should be stored in the document catalog instead (see Section 3.6.1, "Document Catalog").*

TABLE 10.2 Entries in the document information dictionary

KEY	TYPE	VALUE
Title	text string	*(Optional; PDF 1.1)* The document's title.
Author	text string	*(Optional)* The name of the person who created the document.
Subject	text string	*(Optional; PDF 1.1)* The subject of the document.
Keywords	text string	*(Optional; PDF 1.1)* Keywords associated with the document.
Creator	text string	*(Optional)* If the document was converted to PDF from another format, the name of the application (for example, Adobe FrameMaker®) that created the original document from which it was converted.
Producer	text string	*(Optional)* If the document was converted to PDF from another format, the name of the application (for example, Acrobat Distiller) that converted it to PDF.
CreationDate	date	*(Optional)* The date and time the document was created, in human-readable form (see Section 3.8.3, "Dates").
ModDate	date	*(Required if **PieceInfo** is present in the document catalog; otherwise optional; PDF 1.1)* The date and time the document was most recently modified, in human-readable form (see Section 3.8.3, "Dates").
Trapped	name	*(Optional; PDF 1.3)* A name object indicating whether the document has been modified to include trapping information (see Section 10.10.5, "Trapping Support"): False — The document has not yet been trapped; any desired trapping must still be done. (Note that this is the name False, not the boolean value **false**.) Unknown — Either it is unknown whether the document has been trapped or it has been partly but not yet fully trapped; some additional trapping may still be needed. Default value: Unknown. The value of this entry may be set automatically by the software creating the document's trapping information or may be known only to a human operator and entered manually.

Example 10.1 shows a typical document information dictionary.

Example 10.1

```
1 0 obj
   << /Title (PostScript Language Reference, Third Edition)
        /Author (Adobe Systems Incorporated)
        /Creator (Adobe® FrameMaker® 5.5.3 for Power Macintosh®)
        /Producer (Acrobat® Distiller™ 3.01 for Power Macintosh)
        /CreationDate (D:19970915110347-08'00')
        /ModDate (D:19990209153925-08'00')
   >>
endobj
```

10.2.2 Metadata Streams

Metadata, both for an entire document and for components within a document, can be stored in PDF streams called *metadata streams (PDF 1.4)*. The advantages of metadata streams over the document information dictionary include the following:

- PDF-based workflows often embed metadata-bearing artwork as components within larger documents. Metadata streams provide a standard way of preserving the metadata of these components for examination downstream. PDF-aware applications should be able to derive a list of all metadata-bearing document components from the PDF document itself.
- PDF documents are often made available on the World Wide Web or in other environments, where many tools routinely examine, catalog, and classify documents. These tools should be able to understand the self-contained description of the document even if they do not understand PDF.

Besides the usual entries common to all stream dictionaries (see Table 3.4 on page 38), the metadata stream dictionary contains the additional entries listed in Table 10.3.

The contents of a metadata stream is the metadata represented in Extensible Markup Language (XML). This information will be visible as plain text to tools that are not PDF-aware only if the metadata stream is both unfiltered and unencrypted.

TABLE 10.3 Additional entries in a metadata stream dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **Metadata** for a metadata stream.
Subtype	name	*(Required)* The type of metadata stream that this dictionary describes; must be **XML**.

The format of the XML representing the metadata is defined as part of a framework called the Extensible Metadata Platform (XMP) and described in the Adobe document *XMP: Extensible Metadata Platform* (see the Bibliography). This framework provides a way to use XML to represent metadata describing documents and their components, and is intended to be adopted by a wider class of applications than just those that process PDF. It includes a method to embed XML data within non-XML data files in a platform-independent format that can be easily located and accessed by simple scanning rather than requiring the document file to be parsed.

A metadata stream can be attached to a document through the **Metadata** entry in the document catalog (see Chapter 3.6.1, “Document Catalog,” and also see implementation note 138 in Appendix H). In addition, most PDF document components represented as a stream or dictionary can have a **Metadata** entry (see Table 10.4).

TABLE 10.4 Additional entry for components having metadata

KEY	TYPE	VALUE
Metadata	stream	*(Optional; PDF 1.4)* A metadata stream containing metadata for the component.

In general, a PDF stream or dictionary can have metadata attached to it as long as the stream or dictionary represents an actual information resource, as opposed to serving as an implementation artifact. Some PDF constructs are considered implementational, and hence cannot have associated metadata.

For the remaining PDF constructs, there is sometimes ambiguity about exactly which stream or dictionary should bear the **Metadata** entry. Such cases are to be resolved so that the metadata is attached as close as possible to the object that actually stores the data resource described. For example, metadata describing a tiling pattern should be attached to the pattern stream’s dictionary, but a shading should have metadata attached to the shading dictionary itself, rather than to the shading pattern dictionary that refers to it. Similarly, metadata describing an **ICCBased** color space should be attached to the ICC profile stream describing it, and metadata for fonts should be attached to font file streams rather than to font dictionaries.

In tables describing document components in this book, the **Metadata** entry is listed only for those in which it is most likely to be used. Keep in mind, however, that this entry may appear in other components represented as streams or dictionaries.

In addition, metadata can also be associated with marked content within a content stream. This association is created by including an entry in the property list dictionary whose key is **Metadata** and whose value is the metadata stream dictionary. Because this construct refers to an object outside the content stream, the property list must be referred to indirectly as a named resource (see Section 10.5.1, "Property Lists").

10.3 File Identifiers

PDF files may contain references to other PDF files (see Section 3.10, "File Specifications"). Simply storing a file name, however, even in a platform-independent format, does not guarantee that the file can be found. Even if the file still exists and its name has not been changed, different server software applications may identify it in different ways. For example, servers running on DOS platforms must convert all file names to 8 characters and a 3-character extension; different servers may use different strategies for converting longer file names to this format.

External file references can be made more reliable by including a *file identifier (PDF 1.1)* in the file itself and using it in addition to the normal platform-based file designation. Matching the identifier in the file reference with the one in the file itself confirms whether the desired file was found.

File identifiers are defined by the optional **ID** entry in a PDF file's trailer dictionary (see Section 3.4.4, "File Trailer"; see also implementation note 139 in Appendix H). The value of this entry is an array of two strings. The first string is a permanent identifier based on the contents of the file at the time it was originally created, and does not change when the file is incrementally updated. The second string is a changing identifier based on the file's contents at the time it was last updated. When a file is first written, both identifiers are set to the same value.

If both identifiers match when a file reference is resolved, it is very likely that the correct file has been found; if only the first identifier matches, then a different version of the correct file has been found.

To help ensure the uniqueness of file identifiers, it is recommend that they be computed using a message digest algorithm such as MD5 (described in Internet RFC 1321, *The MD5 Message-Digest Algorithm*; see the Bibliography), using the following information (see implementation note 140 in Appendix H):

- The current time
- A string representation of the file's location, usually a pathname
- The size of the file in bytes
- The values of all entries in the file's document information dictionary (see Section 10.2.1, "Document Information Dictionary")

10.4 Page-Piece Dictionaries

A *page-piece dictionary (PDF 1.3)* can be used to hold private application data. The data can be associated with a page or form XObject, by means of the optional **PieceInfo** entry in the page object (see Table 3.27 on page 118) or form dictionary (see Table 4.42 on page 321). Starting with PDF 1.4, private data may also be associated with the PDF document itself, by means of the **PieceInfo** entry in the document catalog (see Table 3.25 on page 113).

Applications can use this dictionary as a place to store any private data they wish in connection with that document, page or form. Such private data can convey information meaningful to the application that produces it (such as information on object grouping for a graphics editor or the layer information used by Adobe Photoshop®), but is typically ignored by general-purpose PDF viewer applications.

As Table 10.5 shows, a page-piece dictionary may contain any number of entries, each keyed by the name of a distinct application or of a "well-known" data type recognized by a family of applications. The value associated with each key is an *application data dictionary* containing the private data to be used by the application. The **Private** entry may have a value of any data type, but typically it will be a dictionary containing all of the private data needed by the application other than the actual content of the document, page or form.

TABLE 10.5 Entries in a page-piece dictionary

KEY	TYPE	VALUE
any application name or well-known data type	dictionary	An application data dictionary (see Table 10.6).

TABLE 10.6 Entries in an application data dictionary

KEY	TYPE	VALUE
LastModified	date	*(Required)* The date and time when the contents of the document, page or form were most recently modified by this application.
Private	(any)	*(Optional)* Any private data appropriate to the application, typically in the form of a dictionary.

The **LastModified** entry indicates when this application last altered the content of the page or form. If the page-piece dictionary contains several application data dictionaries, their modification dates can be compared with those in the corresponding entry of the page object or form dictionary (see Tables 3.27 on page 118 and 4.42 on page 321) or the **ModDate** entry of the document information dictionary (see Table 10.2) to ascertain which application data dictionary corresponds to the current content of the page or form. Because some platforms may use only an approximate value for the date and time or may not deal correctly with differing time zones, modification dates are compared only for equality and not for sequential ordering.

***Note:** It is possible for two or more application data dictionaries to have the same modification date. Applications can use this capability to define multiple or extended versions of the same data format. For example, suppose that earlier versions of an application use an application data dictionary named **PictureEdit**, while later versions of the same application extend the data to include additional items not previously used. The original data could continue to be kept in the **PictureEdit** dictionary, with the additional items placed in a new dictionary named **PictureEditExtended**. This allows the earlier versions of the application to continue to work as before, while later versions are able to locate and use the extended data items.*

10.5 Marked Content

Marked-content operators (PDF 1.2) identify a portion of a PDF content stream as a *marked-content element* of interest to a particular application or PDF plug-in extension. Marked-content elements and the operators that mark them fall into two categories:

- The **MP** and **DP** operators designate a single *marked-content point* in the content stream.
- The **BMC**, **BDC**, and **EMC** operators bracket a *marked-content sequence* of objects within the content stream. Note that this is not simply a sequence of bytes in the content stream, but of complete graphics objects. Each object is fully qualified by the parameters of the graphics state in which it is rendered.

A graphics application, for example, might use marked content to identify a set of related objects as a "group" to be processed as a single unit. A text-processing application might use it to maintain a connection between a footnote marker in the body of a document and the corresponding footnote text at the bottom of the page. Table 10.7 summarizes the marked-content operators.

All marked-content operators except **EMC** take a *tag* operand indicating the role or significance of the marked-content element to the processing application. All such tags must be registered with Adobe Systems (see Appendix E) to avoid conflicts between different applications marking the same content stream. In addition to the tag operand, the **DP** and **BDC** operators specify a *property list* containing further information associated with the marked content. Property lists are discussed further in Section 10.5.1, "Property Lists."

Marked-content operators may appear only *between* graphics objects in the content stream; they may not occur within a graphics object or between a graphics state operator and its operands. Marked-content sequences may be nested one within another, but each sequence must be entirely contained within a single content stream; it may not cross page boundaries, for example.

Note: *The* **Contents** *entry of a page object (see "Page Objects" on page 118), which may be either a single stream or an array of streams, is considered a single stream with respect to marked-content sequences.*

TABLE 10.7 Marked-content operators

OPERANDS	OPERATOR	DESCRIPTION
tag	**MP**	Designate a marked-content point. *tag* is a name object indicating the role or significance of the point.
tag properties	**DP**	Designate a marked-content point with an associated property list. *tag* is a name object indicating the role or significance of the point; *properties* is either an inline dictionary containing the property list or a name object associated with it in the **Properties** subdictionary of the current resource dictionary (see Section 10.5.1, "Property Lists").
tag	**BMC**	Begin a marked-content sequence terminated by a balancing **EMC** operator. *tag* is a name object indicating the role or significance of the sequence.
tag properties	**BDC**	Begin a marked-content sequence with an associated property list, terminated by a balancing **EMC** operator. *tag* is a name object indicating the role or significance of the sequence; *properties* is either an inline dictionary containing the property list or a name object associated with it in the **Properties** subdictionary of the current resource dictionary (see Section 10.5.1, "Property Lists").
—	**EMC**	End a marked-content sequence begun by a **BMC** or **BDC** operator.

When the marked-content operators **BMC**, **BDC**, and **EMC** are combined with the text object operators **BT** and **ET** (see Section 5.3, "Text Objects"), each pair of matching operators (**BMC**…**EMC**, **BDC**…**EMC**, or **BT**…**ET**) must be properly (separately) nested. That is, the sequences

```
BMC                          BT
   BT                           BMC
      …          and               …
   ET                           EMC
EMC                          ET
```

are valid, but

```
BMC                              BT
   BT                               BMC
      …          and                   …
   EMC                              ET
BT                               EMC
```

are not.

10.5.1 Property Lists

The marked-content operators **DP** and **BDC** associate a *property list* with a marked-content element within a content stream. This is a dictionary containing private information meaningful to the program (application or plug-in extension) creating the marked content. Although the dictionary may contain any entries the program wishes to place there, it is suggested that any particular program use the entries in a consistent way; for example, the values associated with a given key should always be of the same type (or small set of types).

If all of the values in a property list dictionary are direct objects, the dictionary may be written inline in the content stream as a direct object. If any of the values are indirect references to objects outside the content stream, the property list dictionary must instead be defined as a named resource in the **Properties** subdictionary of the current resource dictionary (see Section 3.7.2, "Resource Dictionaries") and then referenced by name as the *properties* operand of the **DP** or **BDC** operator.

10.5.2 Marked Content and Clipping

Some PDF path and text objects are defined purely for their effect on the current clipping path, without themselves actually being painted on the page. This occurs when a path object is defined using the operator sequence **W n** or **W* n** (see Section 4.4.3, "Clipping Path Operators") or when a text object is painted in text rendering mode 7 (see Section 5.2.5, "Text Rendering Mode"). Such clipped, unpainted path or text objects are called *clipping objects*. When a clipping object falls within a marked-content sequence, it is not considered part of the sequence unless the entire sequence consists only of clipping objects. In Example 10.2, for instance, the marked-content sequence tagged Clip includes the text string (Clip me), but not the rectangular path that defines the clipping boundary.

Example 10.2

```
/Clip  BMC
    100  100  10  10  re  W  n                 % Clipping path
    (Clip me)  Tj                              % Object to be clipped
EMC
```

Only when a marked-content sequence consists entirely of clipping objects are the clipping objects considered part of the sequence. In this case, the sequence is known as a *marked clipping sequence*. Such sequences may be nested. In Example 10.3, for instance, multiple lines of text are used to clip a subsequent graphics object (in this case, a filled path). Each line of text is bracketed within a separate marked clipping sequence, tagged Pgf; the entire series is bracketed in turn by an outer marked clipping sequence, tagged Clip. Note, however, that the marked-content sequence tagged ClippedText is *not* a marked clipping sequence, since it contains a filled rectangular path that is not a clipping object. The clipping objects belonging to the Clip and Pgf sequences are therefore not considered part of the ClippedText sequence.

Example 10.3

```
/ClippedText  BMC
    /Clip  <<…>>
        BDC
            BT
                7  Tr                          % Begin text clip mode
                /Pgf  BMC
                    (Line 1)  Tj
                EMC
                /Pgf  BMC
                    (Line)  '
                    ( 2)  Tj
                EMC
            ET                                 % Set current text clip
        EMC
    100  100  10  10  re  f                    % Filled path
EMC
```

The precise rules governing marked clipping sequences are as follows:

- A *clipping object* is a path object ended by the operator sequence **W n** or **W* n** or a text object painted in text rendering mode 7.
- An *invisible graphics object* is a path object ended by the operator **n** only (with no preceding **W** or **W***) or a text object painted in text rendering mode 3.
- A *visible graphics object* is a path object ended by any operator other than **n**, a text object painted in any text rendering mode other than 3 or 7, or any XObject invoked by the **Do** operator.
- An *empty marked-content element* is a marked-content point or a marked-content sequence that encloses no graphics objects.
- A *marked clipping sequence* is a marked-content sequence that contains at least one clipping object and no visible graphics objects.
- Clipping objects and marked clipping sequences are considered part of an enclosing marked-content sequence only if it is a marked clipping sequence.
- Invisible graphics objects and empty marked-content elements are always considered part of an enclosing marked-content sequence, regardless of whether it is a marked clipping sequence.
- The **q** (save) and **Q** (restore) operators may not occur within a marked clipping sequence.

Example 10.4 illustrates the application of these rules. Marked-content sequence S4 is a marked clipping sequence, because it contains a clipping object (clipping path 2) and no visible graphics objects. Clipping path 2 is therefore considered part of sequence S4. Marked-content sequences S1, S2, and S3 are *not* marked clipping sequences, since they each include at least one visible graphics object. Thus clipping paths 1 and 2 are not part of any of these three sequences.

Example 10.4

```
/S1  BMC
    /S2  BMC
        /S3  BMC
            0  0  m
            100  100  l
            0  100  l  W  n                      % Clipping path 1

            0  0  m
            200  200  l
            0  100  l  f                          % Filled path
        EMC

        /S4  BMC
            0  0  m
            300  300  l
            0  100  l  W  n                      % Clipping path 2
        EMC
    EMC
    100  100  10  10  re  f                      % Filled path
EMC
```

In Example 10.5, marked-content sequence S1 is a marked clipping sequence, because the only graphics object it contains is a clipping path. Thus the empty marked-content sequence S3 and the marked-content point P1 are both part of sequence S2, and S2, S3, and P1 are all part of sequence S1.

Example 10.5

```
/S1  BMC
    … Clipping path …
    /S2  BMC
        /S3  BMC
        EMC
        /P1  DP
    EMC
EMC
```

In Example 10.6, marked-content sequences S1 and S4 are marked clipping sequences, because the only object they contain is a clipping path. Hence the clipping path is part of sequences S1 and S4; S3 is part of S2; and S2, S3, and S4 are all part of S1.

Example 10.6

```
/S1  BMC
    /S2  BMC
        /S3  BMC
        EMC
    EMC

    /S4  BMC
        …Clipping path…
    EMC
EMC
```

10.6 Logical Structure

PDF's *logical structure* facilities *(PDF 1.3)* provide a mechanism for incorporating structural information about a document's content into a PDF file. Such information might include, for example, the organization of the document into chapters and sections or the identification of special elements such as figures, tables, and footnotes. The logical structure facilities are extensible, allowing applications that produce PDF files to choose what structural information to include and how to represent it, while enabling PDF consumers to navigate a file without knowing the producer's structural conventions.

PDF logical structure shares basic features with standard document markup languages such as HTML, SGML, and XML. A document's logical structure is expressed as a hierarchy of *structure elements*, each represented by a dictionary object. Like their counterparts in other markup languages, PDF structure elements can have content and attributes. Their content can consist of references to document content, references to other structure elements, or both. In PDF, however, rendered document content takes over the role occupied by text in HTML, SGML, and XML.

A PDF document's logical structure is stored separately from its visible content, with pointers from each to the other. This separation allows the ordering and nesting of logical elements to be entirely independent of the order and location of graphics objects on the document's pages.

10.6.1 Structure Hierarchy

The logical structure of a document is described by a hierarchy of objects called the *structure hierarchy* or *structure tree*. At the root of the hierarchy is a dictionary object called the *structure tree root*, located via the **StructTreeRoot** entry in the document catalog (see Section 3.6.1, "Document Catalog"). The remainder of the hierarchy is formed of intermediate nodes called *structure elements*. At the leaves of the tree are individual *content items* associated with structure elements; these may be marked-content sequences, complete PDF objects, or other structure elements.

Tables 10.8 and 10.9 show the contents of the structure tree root and a structure element, respectively.

TABLE 10.8 Entries in the structure tree root

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **StructTreeRoot** for a structure tree root.
K	dictionary or array	*(Optional)* The immediate child or children of the structure tree root in the structure hierarchy. The value may be either a dictionary representing a single structure element or an array of such dictionaries.
IDTree	name tree	*(Required if any structure elements have element identifiers)* A name tree that maps element identifiers (see Table 10.9) to the structure elements they denote.

KEY	TYPE	VALUE
ParentTree	number tree	*(Required if any structure element contains PDF objects or marked-content sequences as content items)* A number tree (see Section 3.8.6, "Number Trees") used in finding the structure elements to which content items belong. Each integer key in the number tree corresponds to a single page of the document or to an individual object (such as an annotation or an XObject) that is a content item in its own right. The integer key is given as the value of the **StructParent** or **StructParents** entry in that object (see "Finding Structure Elements from Content Items" on page 739). The form of the associated value depends on the nature of the object: • For an object that is a content item in its own right, the value is an indirect reference to the object's parent element (the structure element that contains it as a content item). • For a page object or content stream containing marked-content sequences that are content items, the value is an array of references to the parent elements of those marked-content sequences. See "Finding Structure Elements from Content Items" on page 739 for further discussion.
ParentTreeNextKey	integer	*(Optional)* An integer greater than any key in the parent tree, to be used as a key for the next entry added to the tree.
RoleMap	dictionary	*(Optional)* A dictionary mapping the names of structure types used in the document to their approximate equivalents in the set of standard structure types (see Section 10.7.4, "Standard Structure Types").
ClassMap	dictionary	*(Optional)* A dictionary mapping name objects designating attribute classes to the corresponding attribute objects or arrays of attribute objects (see "Attribute Classes" on page 744).

TABLE 10.9 Entries in a structure element dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **StructElem** for a structure element.
S	name	*(Required)* The *structure type*, a name object identifying the nature of the structure element and its role within the document, such as a chapter, paragraph, or footnote (see Section 10.6.2, "Structure Types"). Names of structure types must conform to the guidelines described in Appendix E.

KEY	TYPE	VALUE
P	dictionary	*(Required; must be an indirect reference)* The structure element that is the immediate parent of this one in the structure hierarchy.
ID	string	*(Optional)* The *element identifier*, a string designating this structure element. The string must be unique among all elements in the document's structure hierarchy. The **IDTree** entry in the structure tree root (see Table 10.8) defines the correspondence between element identifiers and the structure elements they denote.
Pg	dictionary	*(Optional; must be an indirect reference)* A page object representing a page on which some or all of the content items designated by the **K** entry are rendered.
K	(various)	*(Optional)* The contents of this structure element, which may consist of one or more marked-content sequences, PDF objects, and other structure elements. The value of this entry may be any of the following: • An integer marked-content identifier denoting a marked-content sequence • A marked-content reference dictionary denoting a marked-content sequence • An object reference dictionary denoting a PDF object • An array, each of whose elements is one of the objects listed above ***Note:*** *If the value of* ***K*** *is a dictionary containing no* ***Type*** *entry, it is assumed to be a structure element dictionary.* See Section 10.6.3, "Structure Content" for further discussion of each of these forms of representation.
A	(various)	*(Optional)* The attribute object or objects, if any, associated with this structure element. Each attribute object is either a dictionary or a stream; the value of this entry may be either a single attribute object or an array of such objects together with their revision numbers (see Section 10.6.4, "Structure Attributes," and "Attribute Revision Numbers" on page 745).

KEY	TYPE	VALUE
C	name or array	*(Optional)* The attribute class or classes, if any, to which this structure element belongs. The value of this entry may be either a single class name or an array of class names together with their revision numbers (see "Attribute Classes" on page 744 and "Attribute Revision Numbers" on page 745). ***Note:** If both the* **A** *and* **C** *entries are present and a given attribute is specified by both, the one specified by the* **A** *entry takes precedence.*
R	integer	*(Optional)* The current revision number of this structure element (see "Attribute Revision Numbers" on page 745). The value must be a nonnegative integer. Default value: 0.
T	text string	*(Optional)* The title of the structure element, a text string representing it in human-readable form. The title should characterize the specific structure element, such as Chapter 1, rather than merely a generic element type, such as Chapter.
Lang	text string	*(Optional; PDF 1.4)* A *language identifier* specifying the natural language for all text in the structure element except where overridden by language specifications for nested structure elements or marked content (see Section 10.8.1, "Natural Language Specification"). If this entry is absent, the language (if any) specified in the document catalog applies.
Alt	text string	*(Optional)* An alternate description of the structure element and its children in human-readable form, useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.2, "Alternate Descriptions").
E	text string	*(Optional; PDF 1.5)* The expanded form of an abbreviation.
ActualText	text string	*(Optional; PDF 1.4)* Text that is an exact replacement for the structure element and its children. This replacement text (which should apply to as small a piece of content as possible) is useful when extracting the document's contents in support of accessibility to disabled users or for other purposes (see Section 10.8.3, "Replacement Text").

10.6.2 Structure Types

Every structure element has a *structure type*, a name object that identifies the nature of the structure element and its role within the document (such as a chapter, paragraph, or footnote). To facilitate the interchange of content among PDF applications, Adobe has defined a set of standard structure types; see Section

10.7.4, "Standard Structure Types." Applications are not required to adopt them, however, but may use any names they wish for their structure types.

Where names other than the standard ones are used, a *role map* may be provided in the structure tree root, mapping the structure types used in the document to their nearest equivalents in the standard set. For example, a structure type named Section used in the document might be mapped to the standard type Sect. The equivalence need not be exact; the role map merely indicates an approximate analogy between types, allowing applications other than the one creating a document to handle its nonstandard structure elements in a reasonable way.

Note: *The same structure type may occur as both a key and a value in the role map, and circular chains of association are explicitly permitted. A single role map can thus define a bidirectional mapping. An application using the role map should follow the chain of associations until it either finds a structure type it recognizes or returns to one it has already encountered.*

Note: *Prior to PDF 1.5, standard element types were never remapped. Beginning with PDF 1.5, an element name is always mapped to its corresponding name in the role map, if there is one, even if the original name is one of the standard types. This is done to allow the element, for example, to represent a tag with the same name as a standard role, even though its use differs from the standard role.*

10.6.3 Structure Content

The content of a structure element may consist of one or more *content items* of any of the following kinds:

- Marked-content sequences embedded within content streams
- Complete PDF objects such as annotations and XObjects
- Other structure elements

The **K** entry in a structure element dictionary can have as its value either a single object denoting one of these items or an array of such objects. Items of any or all three kinds may be mixed in the same content array. The following sections describe how each kind of content item is denoted in a structure element's **K** entry.

Marked-Content Sequences as Content Items

For a sequence of graphics operators in a content stream to be included in the content of a structure element, they must be bracketed as a marked-content sequence between **BDC** and **EMC** operators (see Section 10.5, "Marked Content"). Furthermore, the marked-content sequence must have a property list (see Section 10.5.1, "Property Lists") containing an **MCID** entry. This entry defines an integer *marked-content identifier* that uniquely identifies the marked-content sequence within its content stream. The structure element can then refer to the sequence by specifying its content stream and its marked-content identifier within the stream.

Note: *Although the tag associated with a marked-content sequence is not directly related to the document's logical structure, it should be the same as the structure type of the associated structure element.*

In the general case, a structure element refers to a marked-content sequence by means of a dictionary object called a *marked-content reference*. Table 10.10 shows the contents of this type of dictionary.

TABLE 10.10 Entries in a marked-content reference dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **MCR** for a marked-content reference.
Pg	dictionary	*(Optional; must be an indirect reference)* The page object representing the page on which the graphics objects in the marked-content sequence are rendered. This entry overrides any **Pg** entry in the structure element containing the marked-content reference; it is required if the structure element has no such entry.
Stm	stream	*(Optional; must be an indirect reference)* The content stream containing the marked-content sequence. This entry should be present only if the marked-content sequence resides in a content stream other than the content stream for the page—for example, in a form XObject (see Section 4.9, "Form XObjects") or an annotation's appearance stream (Section 8.4.4, "Appearance Streams"). If this entry is absent, the marked-content sequence is contained in the content stream of the page identified by **Pg** (either in the marked-content reference dictionary or in the parent structure element).

KEY	TYPE	VALUE
StmOwn	(any)	*(Optional; must be an indirect reference)* The PDF object owning the stream identified by **Stm**—for example, the annotation to which an appearance stream belongs.
MCID	integer	*(Required)* The marked-content identifier of the marked-content sequence within its content stream.

Example 10.7 illustrates the use of a marked-content reference to refer to a marked-content sequence within the content stream of a page.

***Note:** This and the following examples omit required **StructParents** entries in the objects used as content items (see "Finding Structure Elements from Content Items" on page 739).*

Example 10.7

```
1 0 obj                              % Structure element
   << /Type /StructElem
       /S /P                         % Structure type
       /P …                          % Parent in structure hierarchy
       /K << /Type /MCR
              /Pg 2 0 R              % Page containing marked-content sequence
              /MCID 0                % Marked-content identifier
          >>
   >>
endobj

2 0 obj                              % Page object
   << /Type /Page
       /Contents 3 0 R               % Content stream
       …
   >>
endobj
```

```
3  0  obj                                % Page's content stream
   << /Length  …  >>
stream
   …
   /P  << /MCID  0 >>                    % Start of marked-content sequence
      BDC
         …
         (Here is some text)  Tj
         …
      EMC                                % End of marked-content sequence
   …
endstream
endobj
```

Marked-content sequences can be used to incorporate the content of a form XObject into a structure element in either of two ways:

- In the content stream that paints the form XObject with the **Do** operator, enclose the invocation of **Do** within a marked-content sequence and associate that sequence with the desired structure element (see Example 10.8). The entire content of the form XObject is considered to be part of the structure element's content, as if it were inserted into the marked-content sequence at the point of the **Do** operator. The form XObject has no internal substructure of its own and cannot in turn contain any marked-content sequences associated with this or other structure elements.
- In the content stream of the form XObject itself, enclose the desired content in a marked-content sequence associated with the structure element (see Example 10.9). The **Do** operator that paints the form XObject should *not* be part of any logical structure content item. However, in this case the form XObject itself can have arbitrary substructure, containing any number of marked-content sequences associated with logical structure elements.

Note: *A form XObject that is painted with multiple invocations of the* **Do** *operator can be incorporated into the document's logical structure only by the first method, with each invocation of* **Do** *individually associated with a structure element.*

Example 10.8

```
1 0 obj                                   % Structure element
   << /Type /StructElem
      /S /P                               % Structure type
      /P …                                % Parent in structure hierarchy
      /Pg 2 0 R                           % Page containing marked-content sequence
      /K 0                                % Marked-content identifier
   >>
endobj

2 0 obj                                                     % Page object
   << /Type /Page
      /Resources << /XObject << /Fm4 4 0 R >>               % Resource dictionary
                 >>                                         %    containing form XObject
      /Contents 3 0 R                                       % Content stream
      …
   >>
endobj

3 0 obj                                   % Page's content stream
   << /Length … >>
stream
   …
   /P << /MCID 0 >>                       % Start of marked-content sequence
      BDC
         /Fm4 Do                          % Paint form XObject
      EMC                                 % End of marked-content sequence
   …
endstream
endobj

4 0 obj                                   % Form XObject
   << /Type /XObject
      /Subtype /Form
      /Length …
   >>
stream
   …
   (Here is some text) Tj
   …
endstream
endobj
```

Example 10.9

```
1  0  obj                              % Structure element
   << /Type  /StructElem
      /S  /P                           % Structure type
      /P  …                            % Parent in structure hierarchy
      /K << /Type  /MCR
            /Pg  2 0 R                 % Page containing marked-content sequence
            /Stm  4 0 R                % Stream containing marked-content sequence
            /MCID  0                   % Marked-content identifier
         >>
   >>
endobj

2  0  obj                                                  % Page object
   << /Type  /Page
      /Resources  <<  /XObject  << /Fm4  4 0 R >>         % Resource dictionary
                  >>                                       %    containing form XObject
      /Contents  3 0 R                                     % Content stream
      …
   >>
endobj

3 0 obj                                % Page's content stream
   << /Length  …  >>
stream
   …
   /Fm4  Do                            % Paint form XObject
   …
endstream
endobj

4  0  obj                              % Form XObject
   << /Type  /XObject
      /Subtype  /Form
      /Length  …
   >>
```

```
stream
   …
   /P  << /MCID  0 >>                     % Start of marked-content sequence
      BDC
         …
         (Here is some text)  Tj
         …
      EMC                                 % End of marked-content sequence
   …
endstream
endobj
```

In the common case where all or most of the marked-content sequences in a structure element's content are on the same page, the **Pg** entry identifying the page can be supplied in the structure element dictionary itself (see Table 10.9 on page 728), rather than in a separate marked-content reference for each sequence. Often, this allows the marked-content reference to be dispensed with entirely and the sequence identified in the structure element's **K** entry by simply giving its integer marked-content identifier directly, as in Example 10.8 above.

PDF Objects as Content Items

When a structure element's content includes an entire PDF object, such as an XObject or an annotation, that is associated with a page but not directly included in the page's content stream, the object is identified in the structure element's **K** entry by an *object reference dictionary* (see Table 10.11). Note that this form of reference is used only for entire objects; if the referenced content forms only part of the object's content stream, it is instead handled as a marked-content sequence, as described in the preceding section.

TABLE 10.11 Entries in an object reference dictionary

KEY	TYPE	VALUE
Type	name	*(Required)* The type of PDF object that this dictionary describes; must be **OBJR** for an object reference.
Pg	dictionary	*(Optional; must be an indirect reference)* The page object representing the page on which the object is rendered. This entry overrides any **Pg** entry in the structure element containing the object reference; it is required if the structure element has no such entry.
Obj	(any)	*(Required; must be an indirect reference)* The referenced object.

Note:** If the referenced object is rendered on multiple pages, each rendering requires a separate object reference; but if it is rendered multiple times on the same page, just a single object reference suffices to identify all of them. (If it is important to distinguish between multiple renditions of the same XObject on the same page, they should be accessed via marked-content sequences enclosing particular invocations of the* ***Do *operator, rather than via object references.)*

Structure Elements as Content Items

One structure element can contain another as a content item simply by referring to the other structure element in its **K** entry. Example 10.10 shows a structure element containing other structure elements as content items, along with a marked-content sequence from a page's content stream.

Example 10.10

```
1 0 obj                          % Containing structure element
   << /Type /StructElem
       /S /MixedContainer        % Structure type
       /P …                      % Parent in structure hierarchy
       /Pg 2 0 R                 % Page containing marked-content sequence
       /K [ 4 0 R                % Three content items: a structure element
             0                   %    a marked-content sequence
             5 0 R               %    another structure element
           ]
   >>
endobj

2 0 obj                          % Page object
   << /Type /Page
       /Contents 3 0 R           % Content stream
       …
   >>
endobj
```

```
3 0 obj                                 % Page's content stream
    << /Length … >>
stream
    …
    /P << /MCID 0 >>                    % Start of marked-content sequence
        BDC
            (Here is some text) Tj
            …
        EMC                             % End of marked-content sequence
    …
endstream
endobj

4 0 obj                                 % First contained structure element
    << /Type /StructElem
        …
    >>
endobj

5 0 obj                                 % Second contained structure element
    << /Type /StructElem
        …
    >>
endobj
```

Finding Structure Elements from Content Items

Because a stream cannot contain object references, there is no way for content items that are marked-content sequences to refer directly back to their parent structure elements (the ones to which they belong as content items). Instead, a different mechanism, the *structural parent tree*, is provided for this purpose. For consistency, content items that are entire PDF objects, such as XObjects, also use the parent tree to refer to their parent structure elements.

The parent tree is a number tree (see Section 3.8.6, "Number Trees"), accessed via the **ParentTree** entry in a document's structure tree root (Table 10.8 on page 727). The tree contains an entry for each object that is a content item of at least one structure element and for each content stream containing at least one marked-content sequence that is a content item. The key for each entry is an integer given as the value of the **StructParent** or **StructParents** entry in the object (see below). The values of these entries are as follows:

- For an object identified as a content item by means of an object reference (see "PDF Objects as Content Items" on page 737), the value is an indirect reference to the parent structure element.
- For a content stream containing marked-content sequences that are content items, the value is an array of indirect references to the sequences' parent structure elements. The array element corresponding to each sequence is found by using the sequence's marked-content identifier as a zero-based index into the array.

Note: *Because marked-content identifiers serve as indices into an array in the structural parent tree, their assigned values should be as small as possible to conserve space in the array.*

The **ParentTreeNextKey** entry in the structure tree root holds an integer value greater than any that is currently in use as a key in the structural parent tree. Whenever a new entry is added to the parent tree, it uses the current value of **ParentTreeNextKey** as its key; the value is then incremented to prepare for the next new entry to be added.

To locate the relevant parent tree entry, each object or content stream that is represented in the tree must contain a special dictionary entry, **StructParent** or **StructParents** (see Table 10.12). Depending on the type of content item, this entry may appear in the page object of a page containing marked-content sequences, in the stream dictionary of a form or image XObject, in an annotation dictionary, or in any other type of object dictionary that is included as a content item in a structure element. Its value is the integer key under which the entry corresponding to the object is to be found in the structural parent tree.

TABLE 10.12 Additional dictionary entries for structure element access

KEY	TYPE	VALUE
StructParent	integer	*(Required for all objects that are structural content items; PDF 1.3)* The integer key of this object's entry in the structural parent tree.
StructParents	integer	*(Required for all content streams containing marked-content sequences that are structural content items; PDF 1.3)* The integer key of this object's entry in the structural parent tree. ***Note:*** *At most one of these two entries may be present in a given object. An object can be either a content item in its entirety or a container for marked-content sequences that are content items, but not both.*

For a content item identified by an object reference, the parent structure element can thus be found by using the value of the **StructParent** entry in the item's object dictionary as a retrieval key in the structural parent tree (found in the **ParentTree** entry of the structure tree root). The corresponding value retrieved from the parent tree is a reference to the parent structure element (see Example 10.11).

Example 10.11

```
1 0 obj                                 % Parent structure element
   << /Type /StructElem
      …
      /K << /Type /OBJR                 % Object reference
            /Pg 2 0 R                   % Page containing form XObject
            /Obj 4 0 R                  % Reference to form XObject
         >>
   >>
endobj

2 0 obj                                 % Page object
   << /Type /Page
      /Resources << /XObject << /Fm4 4 0 R >>     % Resource dictionary
                 >>                               %    containing form XObject
      /Contents 3 0 R                   % Content stream
      …
   >>
endobj
```

```
3 0 obj                                 % Page's content stream
    << /Length … >>
stream
    …
    /Fm4 Do                             % Paint form XObject
    …
endstream
endobj

4 0 obj                                 % Form XObject
    << /Type /XObject
        /Subtype /Form
        /Length …
        /StructParent 6                 % Parent tree key
    >>
stream
    …
endstream
endobj

100 0 obj                               % Parent tree (accessed from structure tree root)
    << /Nums [ 0 101 0 R
               1 102 0 R
               …
               6 1 0 R                  % Entry for page object 2; points back
               …                        %     to parent structure element
             ]
    >>
endobj
```

For a content item that is a marked-content sequence, the retrieval method is similar but slightly more complicated. Because a marked-content sequence is not an object in its own right, its parent tree key is found in the **StructParents** entry of the page object or other content stream in which the sequence resides. The value retrieved from the parent tree is not a reference to the parent structure element itself, but rather an array of such references—one for each marked-content sequence contained within that content stream. The parent structure element for the given sequence is found by using the sequence's marked-content identifier as an index into this array (see Example 10.12).

Example 10.12

```
1 0 obj                                 % Parent structure element
    << /Type /StructElem
        …
        /Pg 2 0 R                       % Page containing marked-content sequence
        /K 0                            % Marked-content identifier
    >>
endobj

2 0 obj                                 % Page object
    << /Type /Page
        /Contents 3 0 R                 % Content stream
        /StructParents 6                % Parent tree key
        …
    >>
endobj

3 0 obj                                 % Page's content stream
    << /Length … >>
stream
    …
    /P << /MCID 0 >>                    % Start of marked-content sequence
        BDC
            (Here is some text) TJ
            …
        EMC                             % End of marked-content sequence
    …
endstream
endobj

100 0 obj                               % Parent tree (accessed from structure tree root)
    << /Nums [ 0 101 0 R
                1 102 0 R
                …
                6 [1 0 R]               % Entry for page object 2; array element at index 0
                …                       %    points back to parent structure element
              ]
    >>
endobj
```

10.6.4 Structure Attributes

An application or plug-in extension that processes logical structure can attach additional information, called *attributes*, to any structure element. The attribute information is held in an *attribute object* associated with the structure element. Any dictionary or stream can serve as an attribute object by including an **O** entry (see Table 10.13) identifying the application or plug-in that owns the attribute information; the owner can then add any additional entries it wishes to the object to hold the attributes. To facilitate the interchange of content among PDF applications, Adobe has defined a set of standard structure attributes, identified by specific standard owners; see Section 10.7.5, "Standard Structure Attributes."

TABLE 10.13 Entry common to all attribute objects

KEY	TYPE	VALUE
O	name	*(Required)* The name of the application or plug-in extension owning the attribute data. The name must conform to the guidelines described in Appendix E.

Any application can attach attributes to any structure element, even one created by another application. Multiple applications can attach attributes to the same structure element; the **A** entry in the structure element dictionary (see Table 10.9 on page 728) can hold either a single attribute object or an array of such objects, together with *revision numbers* for coordinating attributes created by different owner (see "Attribute Revision Numbers" on page 745). An application creating or destroying the second attribute object for a structure element is responsible for converting the value of the **A** entry from a single object to an array or vice versa, as well as for maintaining the integrity of the revision numbers. No inherent order is defined for the attribute objects in an **A** array, but it is considered good practice to add new objects at the end of the array so that the first array element is the one belonging to the application that originally created the structure element.

Attribute Classes

If many structure elements share the same set of attribute values, they can be defined as an *attribute class* sharing the identical attribute object. Structure elements refer to the class by name; the association between class names and attribute objects is defined by a dictionary called the *class map*, kept in the

ClassMap entry of the structure tree root (see Table 10.8 on page 727). Each key in the class map is a name object denoting the name of a class; the corresponding value is an attribute object or an array of such objects.

Note: *Despite the name, PDF attribute classes are unrelated to the concept of a class in object-oriented programming languages such as Java and C++. Attribute classes are strictly a mechanism for storing attribute information in a more compact form; they have no inheritance properties like those of true object-oriented classes.*

The **C** entry in a structure element dictionary (see Table 10.9 on page 728) contains a class name or an array of class names (typically accompanied by revision numbers as well; see "Attribute Revision Numbers," below). For each class named in the **C** entry, the corresponding attribute object or objects are considered to be attached to the given structure element along with those identified in the element's **A** entry. (If both the **A** and **C** entries are present and a given attribute is specified by both, the one specified by the **A** entry takes precedence.)

Attribute Revision Numbers

When an application modifies a structure element or its contents, the change may affect the validity of attribute information attached to that structure element by other applications. A system of *revision numbers* allows applications to detect such changes made by other applications and update their own attribute information accordingly.

A structure element's revision number is kept in the **R** entry in the structure element dictionary (see Table 10.9 on page 728). Initially, the revision number is 0 (the default value if no **R** entry is present); each time an application modifies the structure element or any of its content items, it must signal the change by incrementing the revision number. Note that the revision number is not the same thing as the generation number associated with an indirect object (see Section 3.2.9, "Indirect Objects"); the two are completely separate concepts.

As described below, each attribute object attached to a structure element carries an associated revision number. Each time an attribute object is created or modified, its revision number is set equal to the current value of the structure element's **R** entry. By comparing the attribute object's revision number with that of the structure element, an application can tell whether the contents of the attribute object are still current or whether they have been outdated by more recent changes in the underlying structure element.

Because a single attribute object may be associated with more than one structure element (whose revision numbers may differ), the revision number is not stored directly in the attribute object itself, but rather in the array that associates the attribute object with the structure element. This allows the same attribute object to carry different revision numbers with respect to different structure elements. In the general case, each attribute object in a structure element's **A** array is represented by a pair of array elements, the first containing the attribute object itself and the second the integer revision number associated with it in this structure element. Similarly, the structure element's **C** array contains a pair of elements for each attribute class, the first containing the class name and the second the associated revision number. (Revision numbers equal to 0 can be omitted from both the **A** and **C** arrays; an attribute object or class name that is not followed by an integer array element is understood to have a revision number of 0.)

Note: *A structure element's revision number changes only when the structure element itself or any of its content items is modified. Changes in an attached attribute object do* not *change the structure element's revision number (though they may cause the attribute object's revision number to be updated to match it).*

Occasionally, an application may make such extensive changes to a structure element that they are likely to invalidate all previous attribute information associated with it. In this case, instead of incrementing the structure element's revision number, the application may choose to delete all unknown attribute objects from its **A** and **C** arrays. These two actions are mutually exclusive: the application should *either* increment the structure element's revision number *or* remove its attribute objects, but not both. Note that any application creating attribute objects must be prepared for the possibility that they may be deleted at any time by another application.

10.6.5 Example of Logical Structure

Example 10.13 shows portions of a PDF file with a simple document structure. The structure tree root (object 300) contains elements with structure types **Chap** (object 301) and **Para** (object 304). The **Chap** element, titled Chapter 1, contains elements with types **Head1** (object 302) and **Para** (object 303).

These elements are mapped to the standard structure types specified in Tagged PDF (see Section 10.7.4, “Standard Structure Types”) by means of the role map specified in the structure tree root. Objects 302 through 304 have attached attributes (see Section 10.6.4, “Structure Attributes”and Section 10.7.5, “Standard Structure Attributes”).

The example also illustrates the structure of a parent tree (object 400) mapping content items back to their parent structure elements, and an ID tree (object 403) mapping element identifiers to the structure elements they denote.

Example 10.13

```
1  0  obj                                      % Document catalog
   << /Type  /Catalog
      /Pages  100 0 R                          % Page tree
      /StructTreeRoot  300 0 R                 % Structure tree root
   >>
endobj

100  0  obj                                    % Page tree
   << /Type  /Pages
      /Kids  [  101 1 R                        % First page object
                102 0 R                        % Second page object
             ]
      /Count  2                                % Page count
   >>
endobj

101  1  obj                                    % First page object
   << /Type  /Page
      /Parent  100 0 R                         % Parent is the page tree
      /Resources  << /Font  << /F1  6 0 R      % Font resources
                               /F12  7 0 R
                            >>
                     /ProcSet  [/PDF  /Text]   % Procedure sets
                  >>
      /MediaBox  [0  0  612  792]              % Media box
      /Contents  201 0 R                       % Content stream
      /StructParents  0                        % Parent tree key
   >>
endobj
```

```
201  0  obj                                  % Content stream for first page
    <<  /Length  …  >>
stream
    1  1  1  rg
    0  0  612  792  re  f
    BT                                       % Start of text object

        /Head1  << /MCID  0 >>               % Start of marked-content sequence 0
            BDC
                0  0  0  rg
                /F1  1  Tf
                30  0  0  30  18  732  Tm
                (This is a first level heading. Hello world: )  Tj
                1.1333  TL
                T*
                (goodbye universe.)  Tj
            EMC                              % End of marked-content sequence 0

        /Para  << /MCID  1 >>                % Start of marked-content sequence 1
            BDC
                /F12  1  Tf
                14  0  0  14  18  660.8  Tm
                (This is the first paragraph, which spans pages. It has four fairly short and \
concise sentences. This is the next to last )  Tj
            EMC                              % End of marked-content sequence 1

    ET                                       % End of text object
endstream
endobj

102  0  obj                                  % Second page object
    <<  /Type  /Page
        /Parent  100 0 R                     % Parent is the page tree
        /Resources   <<  /Font   <<  /F1   6 0 R   % Font resources
                                         /F12  7 0 R
                                     >>
                            /ProcSet  [/PDF  /Text]  % Procedure sets
                         >>
        /MediaBox  [0  0  612  792]          % Media box
        /Contents  202 0 R                   % Content stream
        /StructParents  1                    % Parent tree key
    >>
endobj
```

```
202 0 obj                                   % Content stream for second page
   << /Length … >>
stream
   1 1 1 rg
   0 0 612 792 re f
   BT                                       % Start of text object

      /Para << /MCID 0 >>                   % Start of marked-content sequence 0
         BDC
            0 0 0 rg
            /F12 1 Tf
            14 0 0 14 18 732 Tm
            (sentence. This is the very last sentence of the first paragraph.) Tj
         EMC                                % End of marked-content sequence 0

      /Para << /MCID 1 >>                   % Start of marked-content sequence 1
         BDC
            /F12 1 Tf
            14 0 0 14 18 570.8 Tm
            (This is the second paragraph. It has four fairly short and concise sentences. \
This is the next to last ) Tj
         EMC                                % End of marked-content sequence 1

      /Para << /MCID 2 >>                   % Start of marked-content sequence 2
         BDC
            1.1429 TL
            T*
            (sentence. This is the very last sentence of the second paragraph.) Tj
         EMC                                % End of marked-content sequence 2

   ET                                       % End of text object
endstream
endobj

300 0 obj                                   % Structure tree root
   << /Type /StructTreeRoot
      /K [ 301 0 R                          % Two children: a chapter
           304 0 R                          % and a paragraph
         ]
      /RoleMap << /Chap /Sect               % Mapping to standard structure types
                  /Head1 /H
                  /Para /P
               >>
      /ClassMap << /Normal 305 0 R >>       % Class map containing one attribute class
```

```
        /ParentTree  400 0 R                 % Number tree for parent elements
        /ParentTreeNextKey  2                % Next key to use in parent tree
        /IDTree  403 0 R                     % Name tree for element identifiers
    >>
endobj

301  0  obj                                  % Structure element for a chapter
    <<  /Type  /StructElem
        /S  /Chap
        /ID  (Chap1)                         % Element identifier
        /T  (Chapter 1)                      % Human-readable title
        /P  300 0 R                          % Parent is the structure tree root
        /K  [  302 0 R                       % Two children: a section head
               303 0 R                       %     and a paragraph
            ]
    >>
endobj

302  0  obj                                  % Structure element for a section head
    <<  /Type  /StructElem
        /S  /Head1
        /ID  (Sec1.1)                        % Element identifier
        /T  (Section 1.1)                    % Human-readable title
        /P  301 0 R                          % Parent is the chapter
        /Pg  101 1 R                         % Page containing content items
        /A  << /O  /Layout                   % Attribute owned by Layout
                /SpaceAfter 25
                /SpaceBefore 0
                /TextIndent 12.5
            >>
        /K  0                                % Marked-content sequence 0
    >>
endobj

303  0  obj                                  % Structure element for a paragraph
    <<  /Type  /StructElem
        /S  /Para
        /ID  (Para1)                         % Element identifier
        /P  301 0 R                          % Parent is the chapter
        /Pg  101 1 R                         % Page containing first content item
        /C  /Normal                          % Class containing this element's attributes
        /K  [  1                             % Marked-content sequence 1
                << /Type  /MCR               % Marked-content reference to 2nd item
```

```
                         /Pg  102 0 R                  % Page containing second item
                         /MCID  0                      % Marked-content sequence 0
                       >>
                    ]
      >>
    endobj

    304  0  obj                                        % Structure element for another paragraph
      <<  /Type  /StructElem
          /S  /Para
          /ID  (Para2)                                 % Element identifier
          /P  300 0 R                                  % Parent is the structure tree root
          /Pg  102 0 R                                 % Page containing content items
          /C  /Normal                                  % Class containing this element's attributes
          /A  << /O  /Layout
                 /TextAlign  /Justify                  % Overrides attribute provided by classmap
              >>
          /K  [1  2]                                   % Marked-content sequences 1 and 2
      >>
    endobj
    305  0  obj                                        % Attribute class
      <<  /O  /Layout                                  % Owned by Layout
          /EndIndent 0
          /StartIndent 0
          /WritingMode /LrTb
          /TextAlign /Start
      >>
    endobj

    400  0  obj                                        % Parent tree
      <<  /Nums  [ 0  401 0 R                          % Parent elements for first page
                   1  402 0 R                          % Parent elements for second page
                 ]
      >>
    endobj

    401  0  obj                                        % Array of parent elements for first page
      [  302 0 R                                       % Parent of marked-content sequence 0
         303 0 R                                       % Parent of marked-content sequence 1
      ]
    endobj
```

```
402  0  obj                                        % Array of parent elements for second page
   [  303 0 R                                      % Parent of marked-content sequence 0
      304 0 R                                      % Parent of marked-content sequence 1
      304 0 R                                      % Parent of marked-content sequence 2
   ]
endobj

403  0  obj                                        % ID tree root node
   <<  /Kids  [404 0 R]  >>                        % Reference to leaf node
endobj

404  0  obj                                        % ID tree leaf node
   <<  /Limits   [  (Chap1)  (Sec1.3)  ]           % Least and greatest keys in tree
       /Names   [  (Chap1)  301 0 R                % Mapping from element identifiers
                   (Sec1.1)  302 0 R               %     to structure elements
                   (Sec1.2)  303 0 R
                   (Sec1.3)  304 0 R
                ]
   >>
endobj
```

10.7 Tagged PDF

Tagged PDF (PDF 1.4) is a stylized use of PDF that builds on the logical structure framework described in Section 10.6, "Logical Structure." It defines a set of standard structure types and attributes that allow page content (text, graphics, and images) to be extracted and reused for other purposes. It is intended for use by tools that perform operations such as:

- Simple extraction of text and graphics for pasting into other applications
- Automatic reflow of text and associated graphics to fit a page of a different size than was assumed for the original layout
- Processing text for such purposes as searching, indexing, and spell-checking
- Conversion to other common file formats (such as HTML, XML, and RTF) with document structure and basic styling information preserved
- Making content accessible to the visually impaired (see Section 10.8, "Accessibility Support)

A tagged PDF document conforms to the following conventions:

- *Page content* (Section 10.7.2, "Tagged PDF and Page Content"). Tagged PDF defines a set of rules for representing text in the page content, so that characters, words, and text order can be determined reliably. All text is represented in a form that can be converted to Unicode. Word breaks are represented explicitly. Actual content is distinguished from artifacts of layout and pagination. Content is given in an order related to its appearance on the page, as determined by the authoring application.
- A *basic layout model* (Section 10.7.3, "Basic Layout Model"). A set of rules for describing the arrangement of structure elements on the page.
- *Structure types* (Section 10.7.4, "Standard Structure Types"). A set of standard structure types define the meaning of structure elements such as paragraphs, headings, articles, and tables.
- *Structure attributes* (Section 10.7.5, "Standard Structure Attributes"). Standard structure attributes preserve styling information used by the authoring application in laying out content on the page.

***Note:** The types and attributes defined for Tagged PDF are intended to provide a set of standard fallback roles and minimum guaranteed attributes, to enable consumer applications to perform operations such as those mentioned above. Producer applications are free to define additional structure types, so long as they also provide a role mapping to the nearest equivalent standard types, as described in Section 10.6.2, "Structure Types." Likewise, producer applications are free to define additional structure attributes using any of the available extension mechanisms.*

10.7.1 Mark Information Dictionary

Tagged PDF documents must have a **MarkInfo** entry in their document catalog (see Section 3.6.1, "Document Catalog"). The value of this entry is a *mark information dictionary* containing a single entry, **Marked**, as shown in Table 10.14. Its value is a boolean flag indicating whether the document conforms to Tagged PDF conventions. For a document to be recognized as a Tagged PDF document, the value of the **Marked** flag must be **true**.

TABLE 10.14 Entry in the mark information dictionary

KEY	TYPE	VALUE
Marked	boolean	*(Optional)* A flag indicating whether the document conforms to Tagged PDF conventions. Default value: **false**.

10.7.2 Tagged PDF and Page Content

Like all PDF documents, a Tagged PDF document consists of a sequence of self-contained pages, each of which is described by one or more page content streams (including any subsidiary streams such as form XObjects and annotation appearances). Tagged PDF defines some further conventions for organizing and marking content streams so that additional information can be derived from them. These conventions include:

- Distinguishing between the author's original content and artifacts of the layout process (see "Real Content and Artifacts" on page 754).
- Specifying a content order to guide the layout process if the page content must be reflowed (see "Page Content Order" on page 758).
- Representing text in a form from which a Unicode representation and information about font characteristics can be unambiguously derived (see "Extraction of Character Properties" on page 760).
- Representing word breaks unambiguously (see "Identifying Word Breaks" on page 763).
- Marking text with information for making it accessible to the visually impaired (see Section 10.8,"Accessibility Support).

Real Content and Artifacts

The graphics objects in a document can be divided into two classes:

- The *real content* of a document comprises objects representing material originally introduced by the document's author.
- *Artifacts* are graphics objects that are not part of the author's original content but rather are generated by the PDF producer application in the course of pagination, layout, or other strictly mechanical processes.

The document's logical structure encompasses all graphics objects making up the real content and describes how those objects relate to one another; it does not include graphics objects that are mere artifacts of the layout and production process.

A document's real content includes not only the page content stream and subsidiary form XObjects, but also any associated annotations meeting all of the following conditions:

- The annotation has an appearance stream (see Section 8.4.4, "Appearance Streams") containing a normal (**N**) appearance.
- The annotation's Hidden flag (see Section 8.4.2, "Annotation Flags") is not set.
- The annotation is included in the document's logical structure (see Section 10.6, "Logical Structure").

Specification of Artifacts

An artifact can be explicitly distinguished from real content by enclosing it in a marked-content sequence with the tag Artifact:

```
/Artifact                              /Artifact  propertyList
    BMC                                    BDC
        …               or                     …
    EMC                                    EMC
```

The first form is used to identify a generic artifact, the second for those that have an associated property list. Table 10.15 shows the properties that can be included in such a property list.

***Note:** To aid in text reflow, it is recommended that artifacts be defined with property lists whenever possible. Artifacts lacking a specified bounding box are likely to be discarded during reflow.*

TABLE 10.15 Property list entries for artifacts

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of artifact that this property list describes; if present, must be one of the names **Pagination**, **Layout**, or **Page**.
BBox	rectangle	*(Optional)* An array of four numbers in default user space units giving the coordinates of the left, bottom, right, and top edges, respectively, of the artifact's bounding box (the rectangle that completely encloses its visible extent).
Attached	array	*(Optional; pagination artifacts only)* An array of name objects containing one to four of the names Top, Bottom, Left, and Right, specifying the edges of the page, if any, to which the artifact is logically attached. Page edges are defined by the page's crop box (see Section 10.10.1, "Page Boundaries"). The ordering of names within the array is immaterial. Including both Left and Right or both Top and Bottom indicates a full-width or full-height artifact, respectively.

The following types of artifact can be specified by the **Type** entry:

- *Pagination artifacts*. Ancillary page features such as running heads and folios (page numbers).
- *Layout artifacts*. Purely cosmetic typographical or design elements such as footnote rules or background screens.
- *Page artifacts*. Production aids extraneous to the document itself, such as cut marks and color bars.

Tagged PDF consumer applications may have their own ideas about what page content to consider relevant. A text-to-speech engine, for instance, probably should not speak running heads or page numbers when the page is turned. In general, consumer applications are allowed to:

- Disregard elements of page content (for example, specific types of artifact) that are not of interest
- Treat some page elements as *terminals* that are not to be examined further (for example, to treat an illustration as a unit for reflow purposes)
- Replace an element with alternate text (see Section 10.8.2, "Alternate Descriptions")

Depending on their specific goals, different consumer applications may make different decisions in this regard. The purpose of Tagged PDF is not to prescribe what the consumer application should do, but to provide sufficient declarative and descriptive information to allow it to make its own appropriate choices about how to process the content.

Note: *To support consumer applications in providing accessibility to disabled users, Tagged PDF documents should use the natural language specification (**Lang**), alternate description (**Alt**), replacement text (**ActualText**), and abbreviation expansion text (**E**) facilities described in Section 10.8, "Accessibility Support."*

Incidental Artifacts

In addition to objects that are explicitly marked as artifacts and excluded from the document's logical structure, the running text of a page may contain other elements and relationships that are not logically part of the document's real content, but merely incidental results of the process of laying out that content into a document. They may include the following:

- *Hyphenation*. Among the artifacts introduced by text layout is the hyphen marking the incidental division of a word at the end of a line. In Tagged PDF, such an incidental word division must be represented by a *soft hyphen* character, which the Unicode mapping algorithm (see "Unicode Mapping in Tagged PDF" on page 760) translates to the Unicode value U+00AD. (This is distinct from an ordinary *hard hyphen*, whose Unicode value is U+002D.) The producer of a Tagged PDF document must distinguish explicitly between soft and hard hyphens, so that the consumer does not have to guess which type a given character represents.

  Note: *In some languages, the situation is more complicated: there may be multiple hyphen characters, and hyphenation may change the spelling of words. See Example 10.21 on page 814.*

- *Text discontinuities*. The running text of a page, as expressed in page content order (see "Page Content Order," below), may contain places where the normal progression of text suffers a discontinuity. For example, the page may contain the beginnings of two separate articles (see Section 8.3.2, "Articles"), each of which is continued onto a later page of the document; the last words of the first article appearing on the page should not be run together with the first words of the second. Consumer applications can recognize such discontinuities by examining the document's logical structure.

- *Hidden page elements*. For a variety of reasons, elements of a document's logical content may be invisible on the page: they may be clipped, their color may match the background, or they may be obscured by other, overlapping objects. Consumer applications must still be able to recognize and process such hidden elements; for example, formerly invisible elements may become visible when a page is reflowed, or a text-to-speech engine may wish to speak text that is not visible to a sighted reader. For the purposes of Tagged PDF, page content is considered to include all text and illustrations in their entirety, whether or not they are visible when the document is displayed or printed.

Page Content Order

When dealing with material on a page-by-page basis, some Tagged PDF consumer applications may wish to process elements in *page content order*, determined by the sequencing of graphics objects within a page's content stream and of characters within a text object, rather than in the *logical structure order* defined by a depth-first traversal of the page's logical structure hierarchy. The two orderings are logically distinct and may or may not coincide; in particular, any artifacts the page may contain are included in the page content order but not in the logical structure order, since they are not considered part of the document's logical structure. The creator of a Tagged PDF document is responsible for establishing both an appropriate page content order for each page and an appropriate logical structure hierarchy for the entire document.

Because the primary requirement for page content order is to enable reflow to maintain elements in proper reading sequence, it should normally (for Western writing systems) proceed from top to bottom (and, in a multiple-column layout, from column to column), with artifacts in their correct relative places. In general, all parts of an article that appear on a given page should be kept together, even if it flows to scattered locations on the page. Illustrations or footnotes may be interspersed with the text of the associated article or may appear at the end of its content (or, in the case of footnotes, at the end of the entire page's logical content).

Sequencing of Annotations

Annotations associated with a page are not interleaved within the page's content stream, but are placed in the **Annots** array in its page object (see "Page Objects" on page 118). Consequently, the correct position of an annotation in the page content order is not readily apparent, but is determined from the document's logical structure.

Both page content (marked-content sequences) and annotations can be treated as content items that are referenced from structure elements (see Section 10.6.3, "Structure Content"). Structure elements of type Annot (*PDF 1.5*), Link or Form (see "Inline-Level Structure Elements" on page 775 and "Illustration Elements" on page 782) explicitly specify the association between a marked-content sequence and a corresponding annotation. In other cases, if the structure element corresponding to an annotation immediately precedes or follows (in the logical structure order) a structure element corresponding to a marked-content sequence, the annotation is considered to precede or follow the marked-content sequence, respectively, in the page content order.

Note: *If necessary, a Tagged PDF producer may introduce an empty marked-content sequence solely to serve as a structure element for the purpose of positioning adjacent annotations in the page content order.*

Reverse-Order Show Strings

In writing systems that are read from right to left (such as Arabic or Hebrew), one might expect that the glyphs in a font would have their origins at the lower right and their widths (rightward horizontal displacements) specified as negative. For various technical and historical reasons, however, many such fonts follow the same conventions as those designed for Western writing systems, with glyph origins at the lower left and positive widths, as shown in Figure 5.4 on page 356. Consequently, showing text in such right-to-left writing systems requires either positioning each glyph individually (which is tedious and costly) or representing text with show strings (see "Organization and Use of Fonts" on page 350) whose character codes are given in reverse order. When the latter method is used, the character codes' correct page content order is the reverse of their order within the show string.

The marked-content tag ReversedChars informs the Tagged PDF consumer application that show strings within a marked-content sequence contain characters in the reverse of page content order. If the sequence encompasses multiple show strings, only the individual characters within each string are reversed; the strings themselves are in natural reading order. For example, the sequence

```
/ReversedChars
    BMC
        ( olleH) Tj
        −200 0 Td
        (.dlrow) Tj
    EMC
```

represents the text

```
Hello world.
```

The show strings may have a space character at the beginning or end to indicate a word break (see “Identifying Word Breaks” on page 763), but may not contain interior spaces. This is not a serious limitation, since a space provides an opportunity to realign the typography without visible effect; and it serves the valuable purpose of limiting the scope of reversals for word-processing consumer applications.

Extraction of Character Properties

It is a requirement of Tagged PDF that character codes can be unambiguously converted into Unicode values representing the information content of the text. There are several methods for doing this; a Tagged PDF document must conform to at least one of them (see “Unicode Mapping in Tagged PDF,” below).

In addition, Tagged PDF documents must allow some characteristics of the associated fonts to be deduced (see “Font Characteristics” on page 761). These Unicode values and font characteristics can then be used for such operations as cut-and-paste editing, searching, text-to-speech conversion, and exporting to other applications or file formats.

Unicode Mapping in Tagged PDF

Tagged PDF requires that every character code in a document can be mapped to a corresponding Unicode value. Unicode defines scalar values for most of the characters used in the world’s languages and writing systems, as well as providing a *private use area* for application-specific characters. Information about Unicode can be found in the *Unicode Standard*, by the Unicode Consortium (see the Bibliography).

The methods for mapping a character code to a Unicode value are described in Section 5.9.1, "Mapping Character Codes to Unicode Values." Tagged PDF producers should ensure that the PDF file contains enough information to map all character codes to Unicode by one of the methods described there.

An **Alt**, **ActualText** or **E** entry specified in a structure element dictionary or a marked-content property list (see Sections 10.8.2, "Alternate Descriptions," 10.8.3, "Replacement Text," and 10.8.4, "Expansion of Abbreviations and Acronyms") may affect the character stream that some Tagged PDF consumers actually use. For example, some consumers may choose to use the ***Alt*** or ***ActualText*** value and ignore all text and other content associated with the structure element and its descendants.

Some uses of Tagged PDF require characters that may not be available in all fonts, such as the soft hyphen (see "Incidental Artifacts" on page 757). Such characters can be represented either by adding them to the font's encoding or CMap and using ***ToUnicode*** to map them to appropriate Unicode values, or by using an ***ActualText*** entry in the associated structure element to provide substitute characters.

Font Characteristics

In addition to a Unicode value, each character code in a content stream has an associated set of font characteristics. These characteristics are useful when exporting text to another application or file format that has a limited repertoire of available fonts.

Table 10.16 lists a common set of font characteristics corresponding to those used in CSS and XSL; more information can be found in the World Wide Web Consortium document *Extensible Stylesheet Language (XSL) 1.0* (see the Bibliography). Each of the characteristics can be derived from information available in the font descriptor's **Flags** entry (see Section 5.7.1, "Font Descriptor Flags").

TABLE 10.16 Derivation of font characteristics

CHARACTERISTIC	TYPE	DERIVATION
Serifed	boolean	The value of the Serif flag in the font descriptor's **Flags** entry.
Proportional	boolean	The complement of the FixedPitch flag in the font descriptor's **Flags** entry.
Italic	boolean	The value of the Italic flag in the font descriptor's **Flags** entry.
Smallcap	boolean	The value of the SmallCap flag in the font descriptor's **Flags** entry.

Note: *The characteristics shown in the table apply only to character codes contained in show strings within content streams; they do not exist for alternate description text (**Alt**), replacement text (**ActualText**), or abbreviation expansion text (**E**).*

Note: *For the standard 14 Type 1 fonts, the font descriptor may be missing; the well-known values for those fonts are used.*

Tagged PDF in PDF 1.5 defines a wider set of font characteristics, which provide information needed when converting PDF to other files formats such as RTF, HTML, XML and OEB, and also improve accessibility and reflow of tables. Table 10.17 lists these *font selector attributes* and shows how their values are derived.

Note: *If the FontFamily, FontWeight and FontStretch fields are not present in the font descriptor, these values are derived from the font name in an implementation-defined manner.*

TABLE 10.17 Font Selector Attributes

ATTRIBUTE	DESCRIPTION
FontFamily	A string specifying the preferred font family name. Derived from the **FontFamily** entry in the font descriptor (see Table 5.19 on page 418).
GenericFontFamily	A general font classification, used if FontFamily is not found. The following values are supported; with two exceptions, they can be derived from the font descriptor's **Flags** entry: • Serif: chosen if the Serif flag is set and the FixedPitch and Script flags are not set. • SansSerif: chosen if the FixedPitch, Script and Serif flags are all not set. • Cursive: chosen if the Script flag is set and the FixedPitch flag is not set. • Monospace: chosen if the FixedPitch flag is set. • Decorative: cannot be derived. • Symbol: cannot be derived.
FontSize	The size of the font: a positive fixed-point number specifying the height of the typeface in points. It is derived from the *a*, *b*, *c* and *d* fields of the current text matrix.
FontStretch	The stretch value of the font. It can be derived from **FontStretch** in the font descriptor (see Table 5.19 on page 418).
FontStyle	The italicization value of the font. It is set to Italic if the Italic flag is set in the **Flags** field of the font descriptor. Otherwise, it is set to Normal.
FontVariant	The small-caps value of the font. It is set to SmallCaps if the SmallCap flag is set in the **Flags** field of the font descriptor. Otherwise, it is set to Normal.
FontWeight	The weight (thickness) value of the font. It can be derived from **FontWeight** in the font descriptor (see Table 5.19 on page 418). ***Note:*** *The ForceBold flag and the* ***StemV*** *field should not be used to set this attribute.*

Identifying Word Breaks

A document's text stream defines not only the characters in a page's text, but also the words. Unlike a character, the notion of a word is not precisely defined, but depends on the purpose for which the text is being processed. A reflow tool needs to know where it can break the running text into lines; a text-to-speech engine needs to identify the words to be vocalized; spelling checkers and other applications all have their own ideas of what constitutes a word. It is not important for a Tagged PDF document to identify the words within the text stream according to a single, unambiguous definition that will satisfy all of these clients; what is important is that there be enough information available for each client to make that determination for itself.

The consumer of a Tagged PDF document finds words by sequentially examining the Unicode character stream, perhaps augmented by replacement text specified with **ActualText** (see Section 10.8.3, "Replacement Text"). It does not need to guess about word breaks based on information such as glyph positioning on the page, font changes, or glyph sizes. The main consideration is to ensure that the spacing characters that would be present to separate words in a pure text representation are also present in the Tagged PDF.

Note that the identification of what constitutes a word is unrelated to how the text happens to be grouped into show strings. The division into show strings has no semantic significance; in particular, a space or other word-breaking character is still needed even if a word break happens to fall at the end of a show string.

***Note:** Some applications may identify words by simply separating them at every space character; others may be slightly more sophisticated and treat punctuation marks such as hyphens or em dashes as word separators as well. Still other applications may wish to identify possible line-break opportunities by using an algorithm similar to the one in Unicode Standard Annex #29,* Text Boundaries, *available from the Unicode Consortium (see the Bibliography).*

10.7.3 Basic Layout Model

Tagged PDF's standard structure types and attributes are interpreted in the context of a basic layout model that describes the arrangement of structure elements on the page. This model is designed to capture the general intent of the document's underlying structure, and does not necessarily correspond to the one actually used for page layout by the application creating the document. (The PDF content stream itself specifies the exact appearance.) The goal is to provide sufficient information for Tagged PDF consumers to make their own layout decisions while preserving the authoring application's intent as closely as their own layout models allow.

***Note:** The Tagged PDF layout model resembles the ones used in markup languages such as HTML, CSS, XSL, and RTF, but does not correspond exactly to any of them. The model is deliberately defined loosely in order to allow reasonable latitude in the interpretation of structure elements and attributes when converting to other document formats; some degree of variation in the resulting layout from one format to another is to be expected.*

The basic layout model begins with the notion of a *reference area*. This is a rectangular region used by the layout application as a frame or guide in which to place the document's content. Some of the standard structure attributes, such as **StartIndent** and **EndIndent** (see "Layout Attributes for BLSEs" on page 792), are measured from the boundaries of the reference area. Reference areas are not specified explicitly, but are inferred from context; those of interest are generally the column area or areas in a general text layout, the outer bounding box of a table and those of its component cells, and the bounding box of an illustration or other floating element.

The standard structure types are divided into four main categories according to the roles they play in page layout:

- *Grouping elements* (see "Grouping Elements" on page 768) group other elements into sequences or hierarchies, but hold no content directly and have no direct effect on layout.
- *Block-level structure elements* (*BLSE*s) (see "Block-Level Structure Elements" on page 770) describe the overall layout of content on the page, proceeding in the *block-progression direction*.
- *Inline-level structure elements* (*ILSE*s) (see "Inline-Level Structure Elements" on page 775) describe the layout of content within a BLSE, proceeding in the *inline-progression direction*.
- *Illustration elements* (see "Illustration Elements" on page 782) are compact sequences of content, in page content order, that are considered to be unitary objects with respect to page layout. An illustration can be treated as either a BLSE or an ILSE.

The meaning of the terms *block-progression direction* and *inline-progression direction* depends on the writing system in use, as specified by the standard attribute **WritingMode** (see "General Layout Attributes" on page 787). In Western writing systems, the block direction is from top to bottom and the inline direction from left to right; other writing systems use different directions for laying out content.

Because the progression directions can vary depending on the writing system, edges of areas and directions on the page must be identified by terms that are neutral with respect to the progression order rather than by familiar terms such as *up*, *down*, *left*, and *right*. Block layout proceeds from *before* to *after*, inline from *start* to *end*. Thus, for example, in Western writing systems the before and after edges of a reference area are at the top and bottom, respectively, and the start and end edges are at the left and right. Another term, *shift direction* (the direction of shift for a superscript), refers to the direction opposite that for block progression—that is, from after to before (in Western writing systems, from bottom to top).

BLSEs are *stacked* within a reference area in block-progression order. In general, the first BLSE is placed against the before edge of the reference area; subsequent BLSEs are then stacked against preceding ones, progressing toward the after edge, until no more BLSEs will fit in the reference area. If the overflowing BLSE allows itself to be split—such as a paragraph that can be split between lines of text—a portion of it may be included in the current reference area and the remainder carried over to a subsequent reference area (either elsewhere on the same page or on another page of the document). Once the amount of content that fits in a reference area is determined, the placements of the individual BLSEs may be adjusted to bias the placement toward the before edge, the middle, or the after edge of the reference area, or the spacing within or between BLSEs may be adjusted to fill the full extent of the reference area.

***Note:** BLSEs may be nested, with child BLSEs stacked within a parent BLSE in the same manner as BLSEs within a reference area. Except in a few instances noted below (the* BlockAlign *and* InlineAlign *elements), such nesting of BLSEs does not result in the nesting of reference areas; a single reference area prevails for all levels of nested BLSEs.*

Within a BLSE, child ILSEs are *packed* into *lines*. (*Direct content items*—those that are immediate children of a BLSE rather than contained within a child ILSE—are implicitly treated as ILSEs for packing purposes.) Each line is treated as a synthesized BLSE and is stacked within the parent BLSE. Lines may be intermingled with other BLSEs within the parent area. This line-building process is analogous to the stacking of BLSEs within a reference area, except that it proceeds in the inline-progression rather than the block-progression direction: a line is packed with ILSEs beginning at the start edge of the containing BLSE and continuing until the end edge is reached and the line is full. The overflowing ILSE may allow itself to be broken at linguistically determined or explicitly marked break points (such as hyphenation points within a word), and the remaining fragment is then carried over to the next line.

Note: *Certain values of an element's* **Placement** *attribute remove the element from the normal stacking or packing process and allow it instead to "float" to a specified edge of the enclosing reference area or parent BLSE; see "General Layout Attributes" on page 787 for further discussion.*

Two enclosing rectangles are associated with each BLSE and ILSE (including direct content items that are treated implicitly as ILSEs):

- The *content rectangle* is derived from the shape of the enclosed content and defines the bounds used for the layout of any included child elements.
- The *allocation rectangle* includes any additional borders or spacing surrounding the element, affecting how it will be positioned with respect to adjacent elements and the enclosing content rectangle or reference area.

The definitions of these rectangles are determined by layout attributes associated with the structure element; see "Content and Allocation Rectangles" on page 801 for further discussion.

10.7.4 Standard Structure Types

Tagged PDF's *standard structure types* characterize the role of a content element within the document and, in conjunction with the standard structure attributes (described in Section 10.7.5, "Standard Structure Attributes"), how that content is laid out on the page. As discussed in Section 10.6.2, "Structure Types," the structure type of a logical structure element is specified by the **S** entry in its structure element dictionary. To be considered a standard structure type, this value must be either:

- One of the standard structure type names described below
- An arbitrary name that is mapped to one of the standard names by the document's role map (see Section 10.6.2, "Structure Types"), possibly via multiple levels of mapping

***Note:** Beginning with PDF 1.5, an element name is always mapped to its corresponding name in the role map, if there is one, even if the original name is one of the standard types. This is done to allow the element, for example, to represent a tag with the same name as a standard role, even though its use differs from the standard role.*

Ordinarily, structure elements having standard structure types will be processed the same way whether the type is expressed directly or is determined indirectly via the role map. However, some consumer applications may ascribe additional semantics to nonstandard structure types, even though the role map associates them with standard ones. For instance, the actual values of the **S** entries may be used when exporting to a tagged representation such as XML, while the corresponding role-mapped values are used when converting to presentation formats such as HTML or RTF, or for purposes such as reflow or accessibility to disabled users.

***Note:** Most of the standard element types are designed primarily for laying out text; the terminology reflects this usage. However, a layout can in fact include any type of content, such as path or image objects. The content items associated with a structure element are laid out on the page as if they were blocks of text (for a BLSE) or characters within a line of text (for an ILSE).*

Grouping Elements

Grouping elements are used solely to group other structure elements; they are not directly associated with content items. Table 10.18 describes the standard structure types for elements in this category.

TABLE 10.18 Standard structure types for grouping elements

STRUCTURE TYPE	DESCRIPTION
Document	(Document) A complete document. This is the root element of any structure tree containing multiple parts or multiple articles.
Part	(Part) A large-scale division of a document. This type of element is appropriate for grouping articles or sections.

STRUCTURE TYPE	DESCRIPTION
Art	(Article) A relatively self-contained body of text constituting a single narrative or exposition. Articles should be disjoint; that is, they should not contain other articles as constituent elements.
Sect	(Section) A container for grouping related content elements. For example, a section might contain a heading, several introductory paragraphs, and two or more other sections nested within it as subsections.
Div	(Division) A generic block-level element or group of elements.
BlockQuote	(Block quotation) A portion of text consisting of one or more paragraphs attributed to someone other than the author of the surrounding text.
Caption	(Caption) A brief portion of text describing a table or figure.
TOC	(Table of contents) A list made up of table of contents items (structure type TOCI; see below) and/or other TOC elements. A table of contents is thus potentially hierarchical; it is desirable for the hierarchy to correspond in structure to the structure hierarchy of the main body of the document. ***Note:*** *Lists of figures and tables, as well as bibliographies, can be treated as tables of contents for purposes of the standard structure types.*
TOCI	(Table of contents item) An individual member of a table of contents. Its children may include a label (structure type Lbl; see "List Elements" on page 772), references to the title and the page number (structure type Reference; see "Inline-Level Structure Elements" on page 775), NonStruct elements (for wrapping a leader artifact, for example; see "Grouping Elements" on page 768) and descriptive text (structure type P; see "Paragraphlike Elements" on page 771).
Index	(Index) A sequence of entries containing identifying text accompanied by reference elements (structure type Reference; see "Inline-Level Structure Elements" on page 775) that point out occurrences of the specified text in the main body of a document.

STRUCTURE TYPE	DESCRIPTION
NonStruct	(Nonstructural element) A grouping element having no inherent structural significance; it serves solely for grouping purposes. This type of element differs from a division (structure type Div; see above) in that it is not interpreted or exported to other document formats; however, its descendants are to be processed normally.
Private	(Private element) A grouping element containing private content belonging to the application producing it. The structural significance of this type of element is unspecified, and is determined entirely by the producer application. Neither the Private element nor any of its descendants are to be interpreted or exported to other document formats.

For most content extraction formats, the document must be a tree with a single top-level element; the structure tree root (identified by the **StructTreeRoot** entry in the document catalog) must have only one child in its **K** (kids) array. If the PDF file contains a complete document, the structure type Document is recommended for this top-level element in the logical structure hierarchy; if the file contains a well-formed document fragment, one of the structure types Part, Art, Sect, or Div may be used instead.

Block-Level Structure Elements

A *block-level structure element (BLSE)* is any region of text or other content that is laid out in the block-progression direction, such as a paragraph, heading, list item, or footnote. A structure element is a BLSE if its structure type (after role mapping, if any) is one of those listed in Table 10.19. All other standard structure types are treated as ILSEs, with the following exceptions:

- TR (Table row), TH (Table header), TD (Table data), THead (Table head), TBody (Table body), and TFoot (Table footer), which are used to group elements within a table and are considered neither BLSEs nor ILSEs.
- Elements with a **Placement** attribute (see "General Layout Attributes" on page 787) other than the default value of Inline

TABLE 10.19 Block-level structure elements

CATEGORY	STRUCTURE TYPES		
Paragraphlike elements	P	H1	H4
	H	H2	H5
		H3	H6
List elements	L	Lbl	
	LI	LBody	
Table element	Table		

In many cases, a BLSE appears as one compact, contiguous piece of page content; in other cases, it is discontiguous. Examples of the latter include a BLSE that extends across a page boundary or is interrupted in the page content order by another, nested BLSE or a directly included footnote. When necessary, Tagged PDF consumer applications can recognize such fragmented BLSEs from the logical structure and use this information to reassemble them and properly lay them out.

Paragraphlike Elements

Table 10.20 describes structure types for *paragraphlike elements* that consist of running text and other content laid out in the form of conventional paragraphs (as opposed to more specialized layouts such as lists and tables).

TABLE 10.20 Standard structure types for paragraphlike elements

STRUCTURE TYPE	DESCRIPTION
H	(Heading) A label for a subdivision of a document's content. It should be the first child of the division that it heads.
H1–H6	Headings with specific levels, for use in applications that cannot hierarchically nest their sections and thus cannot determine the level of a heading from its level of nesting.
P	(Paragraph) A low-level division of text.

List Elements

The structure types described in Table 10.21 are used for organizing the content of lists.

TABLE 10.21 Standard structure types for list elements

STRUCTURE TYPE	DESCRIPTION
L	(List) A sequence of items of like meaning and importance. Its immediate children should be an optional caption (structure type Caption; see “Grouping Elements” on page 768) followed by one or more list items (structure type LI; see below).
LI	(List item) An individual member of a list. Its children may be one or more labels, list bodies, or both (structure types Lbl or LBody; see below).
Lbl	(Label) A name or number that distinguishes a given item from others in the same list or other group of like items. In a dictionary list, for example, it contains the term being defined; in a bulleted or numbered list, it contains the bullet character or the number of the list item and associated punctuation.
LBody	(List body) The descriptive content of a list item. In a dictionary list, for example, it contains the definition of the term. It can either contain the content directly or have other BLSEs, perhaps including nested lists, as children.

Table Elements

The structure types described in Table 10.22 are used for organizing the content of tables.

Note: *Strictly speaking, the Table element is a BLSE, while the others in this table are neither BLSEs or ILSEs.*

TABLE 10.22 Standard structure types for table elements

STRUCTURE TYPE	DESCRIPTION
Table	(Table) A two-dimensional layout of rectangular data cells, possibly having a complex substructure. It contains either one or more table rows (structure type TR; see below) as children; or an optional table head (structure type THead; see below) followed by one or more table body elements (structure type TBody; see below) and an optional table footer (structure type TFoot; see below). In addition, a table may have an optional caption (structure type Caption; see "Grouping Elements" on page 768) as its first or last child.
TR	(Table row) A row of headings or data in a table. It may contain table header cells and table data cells (structure types TH and TD; see below).
TH	(Table header cell) A table cell containing header text describing one or more rows or columns of the table.
TD	(Table data cell) A table cell containing data that is part of the table's content.
THead	(Table header row group; *PDF 1.5)* A group of rows that constitute the header of a table. If the table is split across multiple pages, these rows may be re-drawn at the top of each table fragment (although there is only one THead element.
TBody	(Table body row group; *PDF 1.5)* A group of rows that constitute the main body portion of a table. If the table is split across multiple pages, the body area may be broken apart on a row boundary. A table may have multiple TBody elements, to allow for the drawing of a border or background for a set of rows.
TFoot	***Note:*** (Table footer row group; *PDF 1.5)* A group of rows that constitute the footer of a table. If the table is split across multiple pages, these rows may be re-drawn at the bottom of each table fragment (although there is only one TFoot element.

Note: *The association of headers with rows and columns of data is typically determined heuristically by applications. Such heuristics may fail for complex tables, so the standard attributes for tables shown in Table 10.32 can be used to make the association explicit.*

Usage Guidelines for Block-Level Structure

Because different consumer applications use PDF's logical structure facilities in different ways, Tagged PDF does not enforce any strict rules regarding the order and nesting of elements using the standard structure types. Furthermore, each export format has its own conventions for logical structure. However, adhering to certain general guidelines will help to achieve the most consistent and predictable interpretation among different Tagged PDF consumers.

As described under "Grouping Elements" on page 768, a Tagged PDF document can have one or more levels of grouping elements, such as Document, Part, Art (Article), Sect (Section), and Div (Division). The descendants of these are BLSEs, such as H (Heading), P (Paragraph), and L (List), that hold the actual content. Their descendants, in turn, are the content items themselves or ILSEs that further describe the content.

***Note:** As noted earlier, elements with structure types that would ordinarily be treated as ILSEs can have a **Placement** attribute (see "General Layout Attributes" on page 787) that causes them to be treated as BLSEs instead. Such elements may be included as BLSEs in the same manner as headings and paragraphs.*

The block-level structure can follow one of two principal paradigms:

- *Strongly structured*. The grouping elements nest to as many levels as necessary to reflect the organization of the material into articles, sections, subsections, and so on. At each level, the children of the grouping element consist of a heading (H), one or more paragraphs (P) for content at that level, and perhaps one or more additional grouping elements for nested subsections.
- *Weakly structured*. The document is relatively flat, having perhaps only one or two levels of grouping elements, with all the headings, paragraphs, and other BLSEs as their immediate children. In this case, the organization of the material is not reflected in the logical structure; however, it can be expressed by the use of headings with specific levels (H1–H6).

The strongly structured paradigm is used by some rich document models based on XML; the weakly structured paradigm is typical of documents represented in HTML.

Lists and tables should be organized using the specific structure types described under "List Elements" on page 772 and "Table Elements" on page 772. Likewise, tables of contents and indexes should be structured as described for the TOC and Index structure types under "Grouping Elements" on page 768.

Inline-Level Structure Elements

An *inline-level structure element (ILSE)* contains a portion of text or other content having specific styling characteristics or playing a specific role in the document. Within a paragraph or other block defined by a containing BLSE, consecutive ILSEs—possibly intermixed with other content items that are direct children of the parent BLSE—are laid out consecutively in the inline-progression direction (left to right in Western writing systems). The resulting content may then be broken into multiple *lines*, which in turn are stacked in the block-progression direction. It is possible for an ILSE in turn to contain a BLSE, which is then treated as a unitary item of layout in the inline direction. Table 10.23 lists the standard structure types for ILSEs.

TABLE 10.23 Standard structure types for inline-level structure elements

STRUCTURE TYPE	DESCRIPTION
Span	(Span) A generic inline portion of text having no particular inherent characteristics. It can be used, for example, to delimit a range of text with a given set of styling attributes. ***Note:*** *Not all inline style changes need to be identified as a span. Text color and font changes (including modifiers such as bold, italic, and small caps) need not be so marked, since these can be derived from the PDF content (see "Font Characteristics" on page 761). However, it is necessary to use a span in order to apply explicit layout attributes such as* ***LineHeight****,* ***BaselineShift****, or* ***TextDecorationType*** *(see "Layout Attributes for ILSEs" on page 797).* ***Note:*** *Marked-content sequences having the tag Span are also used to carry certain accessibility properties (****Alt****,* ***ActualText****,* ***Lang*** *and* ***E****; see Section 10.8, "Accessibility Support"). Such sequences lack an* ***MCID*** *property and are not associated with any structure element. This use of the Span marked-content tag is distinct from its use as a structure type.*

STRUCTURE TYPE	DESCRIPTION
Quote	(Quotation) An inline portion of text attributed to someone other than the author of the surrounding text. ***Note:*** *The quoted text is contained inline within a single paragraph; this differs from the block-level element* BlockQuote *(see "Grouping Elements" on page 768), which consists of one or more complete paragraphs (or other elements presented as if they were complete paragraphs).*
Note	(Note) An item of explanatory text, such as a footnote or an endnote, that is referred to from within the body of the document. It may have a label (structure type Lbl; see "List Elements" on page 772) as a child. The note itself may be included as a child of the structure element in the body text that refers to it, or it may be included elsewhere (such as in an endnotes section) and accessed via a reference (structure type Reference; see below). ***Note:*** *Tagged PDF does not prescribe the placement of footnotes in the page content order; they can be either inline or at the end of the page, at the discretion of the producer application.*
Reference	(Reference) A citation to content elsewhere in the document.
BibEntry	(Bibliography entry) A reference identifying the external source of some cited content. It may contain a label (structure type Lbl; see "List Elements" on page 772) as a child. ***Note:*** *Although a bibliography entry is likely to include component parts identifying the cited content's author, work, publisher, and so forth, at the time of publication no standard structure types are defined at this level of detail.*
Code	(Code) A fragment of computer program text.
Link	(Link) An association between a portion of the ILSE's content and a corresponding link annotation or annotations (see "Link Annotations" on page 576). Its children are one or more content items or child ILSEs and one or more object references (see "PDF Objects as Content Items" on page 737) identifying the associated link annotations. See "Link Elements," below, for further discussion.

STRUCTURE TYPE	**DESCRIPTION**
Annot	(Annotation; *PDF 1.5*) An association between a portion of the ILSE's content and a corresponding PDF annotation (see Section 8.4, "Annotations"). Annot is used for all PDF annotations except link annotations (see the Link element, above) and widget annotations (see the Form element in Table 10.25 on page 782). See "Annotation Elements" on page 780 for further discussion.
Ruby	(Ruby; *PDF 1.5*) A side-note (annotation), written in a smaller text size and placed adjacent to the base text to which it refers. It is used in Japanese and Chinese to describe the pronunciation of unusual words or to describe such items as abbreviations and logos. A Ruby element may also contain the RB, RT and RP elements. See "Ruby and Warichu Elements" on page 780 for more details.
Warichu	(Warichu; *PDF 1.5*) A comment or annotation in a smaller text size and formatted onto two smaller lines within the height of the containing text line and placed following (inline) the base text to which it refers. It is used in Japanese for descriptive comments and for ruby annotation text that is too long to be aesthetically formatted as a ruby. A Warichu element may also contain the WT and WP elements. See "Ruby and Warichu Elements" on page 780 for more details.

Link Elements

Link annotations (like all PDF annotations) are associated with a geometric region of the page rather than with a particular object in its content stream. Any connection between the link and the content is based solely on visual appearance rather than on an explicitly specified association. For this reason, link annotations in themselves are not useful to the visually impaired or to applications needing to determine which content can be activated to invoke a hypertext link.

Tagged PDF link elements (structure type Link) use PDF's logical structure facilities to establish the association between content items and link annotations, providing functionality comparable to HTML hypertext links. The children of such an element consist of:

- One or more content items or other ILSEs (except other links)
- Object references (see "PDF Objects as Content Items" on page 737) to one or more link annotations associated with the content

A link element may contain several link annotations if the geometry of the content requires it; for instance, if a span of text wraps from the end of one line to the beginning of another, separate link annotations may be needed in order to cover the two portions of text. All of the child link annotations must have the same target and action. In order to maintain a geometric association between the content and the annotation that is consistent with the logical association, all of the link element's content must be covered by the union of its child link annotations.

As an example, consider the following fragment of HTML code, which produces a line of text containing a hypertext link:

```
<html>
    <body>
        <p>
        Here is some text <a href=http://www.adobe.com>with a link</a> inside.
    </body>
</html>
```

Example 10.14 shows an equivalent fragment of PDF using a link element, whose text it displays in blue and underlined. Example 10.15 shows an excerpt from the associated logical structure hierarchy.

Example 10.14

```
/P << /MCID 0 >>                          % Marked-content sequence 0 (paragraph)
    BDC                                   % Begin marked-content sequence
        BT                                % Begin text object
            /T1_0 1 Tf                    % Set text font and size
            14 0 0 14 10.000 753.976 Tm   % Set text matrix
            0.0 0.0 0.0 rg                % Set nonstroking color to black
            (Here is some text ) Tj       % Show text preceding link
        ET                                % End text object
    EMC                                   % End marked-content sequence

/Link << /MCID 1 >>                       % Marked-content sequence 1 (link)
    BDC                                   % Begin marked-content sequence
        0.7 w                             % Set line width
        [ ] 0 d                           % Solid dash pattern
        111.094 751.8587 m                % Move to beginning of underline
        174.486 751.8587 l                % Draw underline
        0.0 0.0 1.0 RG                    % Set stroking color to blue
        S                                 % Stroke underline
        BT                                      % Begin text object
            14  0  0  14  111.094  753.976  Tm  % Set text matrix
            0.0  0.0  1.0  rg                   % Set nonstroking color to blue
            (with a link)  Tj                   % Show text of link
        ET                                      % End text object
    EMC                                         % End marked-content sequence

/P  << /MCID  2 >>                              % Marked-content sequence 2 (paragraph)
    BDC                                         % Begin marked-content sequence
        BT                                      % Begin text object
            14  0  0  14  174.486  753.976  Tm  % Set text matrix
            0.0  0.0  0.0  rg                   % Set nonstroking color to black
            ( inside.)  Tj                      % Show text following link
        ET                                      % End text object
    EMC                                         % End marked-content sequence
```

Example 10.15

```
501  0  obj                            % Structure element for paragraph
    <<  /Type  /StructElem
        /S  /P
        . . .
        /K  [  0                       % Three children: marked-content sequence 0
               502 0 R                 %     Link
               2                       %     Marked-content sequence 2
            ]
    >>
endobj

502  0  obj                            % Structure element for link
    <<  /Type  /StructElem
        /S  /Link
        . . .
        /K  [  1                       % Two children: marked-content sequence 1
               503 0 R                 %     Object reference to link annotation
            ]
    >>
endobj

503  0  obj                            % Object reference to link annotation
    <<  /Type  /OBJR
        /Obj  600 0 R                  % Link annotation (not shown)
    >>
endobj
```

Annotation Elements

Tagged PDF annotation elements (structure type Annot; *PDF 1.5*)use PDF's logical structure facilities to establish the association between content items and PDF annotations. Annotation elements are used for all types of annotations other than links (see "Link Elements" on page 777) and forms (see Table 10.25 on page 782).

The children of annotation elements consist of:

- Object references (see "PDF Objects as Content Items" on page 737) to one or more annotation dictionaries.
- Optionally, one or more content items, such as marked-content sequences, or other ILSEs (except other annotations) associated with the annotations.

If an Annot element has no children other than object references, its rendering is defined by the appearance of the referenced annotations, and its text content is treated as if it were a Span element. It may have an optional **BBox** attribute; if supplied, this attribute overrides the rectangle specified by the annotation dictionary's **Rect** entry.

If the Annot element has children that are content items, those children represent the displayed form of the annotation, and the appearance of the associated annotation may also be applied (for example, with a **Highlight** annotation).

There can be multiple children that are object references to different annotations, subject to the constraint that the annotations must be the same except for their **Rect** entry. This is much the same as is done for the Link element; it allows an annotation to be associated with discontiguous pieces of content, such as line-wrapped text.

Ruby and Warichu Elements

Ruby text is a side-note, written in a smaller text size and placed adjacent to the base text to which it refers. It is used in Japanese and Chinese to describe the pronunciation of unusual words or to describe such items as abbreviations and logos.

Warichu text is a comment or annotation, written in a smaller text size and formatted onto two smaller lines within the height of the containing text line and placed following (inline) the base text to which it refers. It is used in Japanese for descriptive comments and for ruby annotation text that is too long to be aesthetically formatted as a ruby.

TABLE 10.24 Standard structure types for Ruby and Warichu elements (*PDF 1.5*)

STRUCTURE TYPE	DESCRIPTION
Ruby	(Ruby) The wrapper around the entire ruby assembly. It contains one RB element, followed by either an RT element or a three-element group consisting of RP, RT, and RP. Ruby elements, and their content elements, may not break across multiple lines.
RB	(Ruby base text) The full-size text to which the ruby annotation is applied. RB can contain text, other inline elements, or a mixture of both. It may have the **RubyAlign** attribute.
RT	(Ruby annotation text) The smaller-size text that is placed adjacent to the ruby base text. It can contain text, other inline elements, or a mixture of both. It may have the **RubyAlign** and **RubyPosition** attributes.
RP	(Ruby punctuation) Punctuation surrounding the ruby annotation text. It us used only when a ruby annotation cannot be properly formatted in a ruby style and instead is formatted as a normal comment, or when it is formatted as a warichu. It contains text (usually a single opening or closing parenthesis or similar bracketing character).
Warichu	(Warichu) The wrapper around the entire warichu assembly. It may contain a three-element group consisting of WP, WT, and WP. Warichu elements (and their content elements) may wrap across multiple lines, according to the warichu breaking rules described in the Japanese Industrial Standard (JIS) X 4051-1995.
WT	(Warichu text) The smaller-size text of a warichu comment that is formatted into two lines and placed between surrounding WP elements.
WP	(Warichu punctuation) The punctuation that surrounds the WT text. It contains text (usually a single opening or closing parenthesis or similar bracketing character). According to JIS X 4051-1995, the parentheses surrounding a warichu may be converted to a space (nominally 1/4 EM in width) at the discretion of the formatter.

Illustration Elements

Tagged PDF defines an *illustration element* as any structure element whose structure type (after role mapping, if any) is one of those listed in Table 10.25. The illustration's content must consist of one or more complete graphics objects; it may not appear between the **BT** and **ET** operators delimiting a text object (see Section 5.3, "Text Objects"). It may include clipping only in the form of a contained marked clipping sequence, as defined in Section 10.5.2, "Marked Content and Clipping." In Tagged PDF, all such marked clipping sequences must carry the marked-content tag Clip.

TABLE 10.25 Standard structure types for illustration elements

STRUCTURE TYPE	DESCRIPTION
Figure	(Figure) An item of graphical content. Its placement may be specified with the **Placement** layout attribute (see "General Layout Attributes" on page 787).
Formula	(Formula) A mathematical formula. ***Note:*** *This structure type is useful only for identifying an entire content element as a formula; no standard structure types are defined for identifying individual components within the formula. From a formatting standpoint, the formula is treated similarly to a figure (structure type* Figure*; see above).*
Form	(Form) A widget annotation representing an interactive form field (see Section 8.6, "Interactive Forms"). Its only child is an object reference (see "PDF Objects as Content Items" on page 737) identifying the widget annotation; the annotation's appearance stream (see Section 8.4.4, "Appearance Streams") defines the rendering of the form element.

An illustration may have logical substructure, including other illustrations. For purposes of reflow, however, it is moved (and perhaps resized) as a unit, without examining its internal contents. To be useful for reflow, it must have a **BBox** attribute; it may also have **Placement**, **Width**, **Height**, and **BaselineShift** attributes (see "Layout Attributes" on page 786).

Often an illustration will be logically part of, or at least attached to, a paragraph or other element of a document. Any such containment or attachment is represented through the use of the Figure structure type; the Figure element indicates the point of attachment, while its **Placement** attribute describes the nature of the attachment. An illustration element without a **Placement** attribute is treated as an ILSE and laid out inline.

Note: *For accessibility to disabled users and other text extraction purposes, an illustration element should always have an* ***Alt*** *entry or an* ***ActualText*** *entry (or both) in its structure element dictionary (see Sections 10.8.2, "Alternate Descriptions," and 10.8.3, "Replacement Text").* ***Alt*** *is a description of the illustration, whereas* ***ActualText*** *gives the exact text equivalent of a graphical illustration that has the appearance of text.*

10.7.5 Standard Structure Attributes

In addition to the standard structure types themselves, Tagged PDF also defines standard layout and styling attributes for structure elements of those types. These attributes enable predictable formatting to be applied during operations such as reflow and export of PDF content to other document formats.

As discussed in Section 10.6.4, "Structure Attributes," attributes are defined in *attribute objects*, which are dictionaries or streams attached to a structure element in either of two ways:

- The **A** entry in the structure element dictionary identifies an attribute object or an array of such objects.
- The **C** entry in the structure element dictionary gives the name of an *attribute class* or an array of such names. The class name is in turn looked up in the *class map*, a dictionary identified by the **ClassMap** entry in the structure tree root, yielding an attribute object, or array of objects, corresponding to the class.

In addition to the standard structure attributes described below, there are several other optional entries—**Lang**, **Alt**, **ActualText** and **E**—that are described in Section 10.8, "Accessibility Support," but are useful to other PDF consumers as well. They appear in the following places in a PDF file (rather than in attribute dictionaries):

- As entries in the structure element dictionary (see Table 10.9 on page 728).
- As entries in property lists attached to marked-content sequences with a Span tag (see Section 10.5, "Marked Content").

Example 10.13 illustrates the use of standard structure attributes.

Standard Attribute Owners

Each attribute object has an *owner*, specified by the object's **O** entry, which determines the interpretation of the attributes defined in the object's dictionary. Multiple owners may define like-named attributes with different value types or interpretations. Tagged PDF defines a set of standard attribute owners, shown in Table 10.26.

TABLE 10.26 Standard attribute owners

OWNER	DESCRIPTION
Layout	Attributes governing the layout of content
List	Attributes governing the numbering of lists
Table	Attributes governing the organization of cells in tables
XML-1.00	Additional attributes governing translation to XML, version 1.00
HTML-3.20	Additional attributes governing translation to HTML, version 3.20
HTML-4.01	Additional attributes governing translation to HTML, version 4.01
OEB-1.00	Additional attributes governing translation to OEB, version 1.0
RTF-1.05	Additional attributes governing translation to Microsoft Rich Text Format, version 1.05
CSS-1.00	Additional attributes governing translation to a format using CSS, version 1.00
CSS-2.00	Additional attributes governing translation to a format using CSS, version 2.00

An attribute object owned by a specific export format, such as **XML-1.00**, is applied only when exporting PDF content to that format. Such format-specific attributes override any corresponding attributes owned by **Layout**, **List**, or **Table**. There may also be additional format-specific attributes; the set of possible attributes is open-ended and is not explicitly specified or limited by Tagged PDF.

Attribute Values and Inheritance

Some attributes are defined as *inheritable*. Inheritable attributes propagate down the structure tree; that is, an attribute that is specified for an element applies to all the descendants of the element in the structure tree, unless a descendent element specifies an explicit value for the attribute.

***Note:** The description of each of the standard attributes in this section specifies whether their values are inheritable or not.*

It is permissible to specify an inheritable attribute on an element for the purpose of propagating its value to child elements, even if the attribute is not meaningful for the parent element itself. Non-inheritable attributes may be specified only for elements on which they would be meaningful.

The following list shows the priority for setting attribute values. A processing application will set an attribute's value to the first item in the list that applies:

1. The value of the attribute specified in the element's **A** entry, owned by one of the export formats (such as **XML**, **HTML-3.20**, **HTML-4.01**, **OEB-1.0**, **RTF**, **CSS-1.00**, **CSS-2.0**), if present, and if outputting to that format.
2. The value of the attribute specified in the element's **A** entry, owned by **Layout** or **Table** or **List**, if present.
3. The value of the attribute specified in a class map associated with the element's **C** entry, if there is one.
4. The resolved value of the parent structure element, if the attribute is inheritable.
5. The default value for the attribute, if there is one.

***Note:** The attributes **Lang**, **Alt**, **ActualText** and **E** do not appear in attribute dictionaries. The rules governing their application are discussed in Section 10.8, "Accessibility Support."*

There is no semantic distinction between attributes that are specified explicitly and ones that are inherited. Logically, the structure tree has attributes fully bound to each element, even though some may be inherited from an ancestor element. This is consistent with the behavior of properties (such as font characteristics) that are not specified by structure attributes but are derived from the content.

Layout Attributes

Layout attributes specify parameters of the layout process used to produce the appearance described by a document's PDF content. Attributes in this category are defined in attribute objects whose **O** (owner) entry has the value **Layout** (or is one of the format-specific owner names listed in Table 10.26 on page 784). The intent is that these parameters can be used to reflow the content or export it to some other document format with at least basic styling preserved.

Table 10.30 summarizes the standard layout attributes and the structure elements to which they apply. The following sections describe the meaning and usage of these attributes.

Note: *An "*" after the attribute name indicates that the attribute is inheritable. As described in "Attribute Values and Inheritance" on page 785, an inheritable attribute may be specified for any element, regardless of whether it is meaningful for that element, in order to propagate it to the descendants.*

TABLE 10.27 Standard layout attribute

STRUCTURE ELEMENTS	ATTRIBUTES
Any structure element	**Placement** **WritingMode*** **BackgroundColor** **BorderColor*** **BorderStyle** **BorderThickness*** **Color*** **Padding**
Any BLSE ILSEs with **Placement** other than Inline	**SpaceBefore** **SpaceAfter** **StartIndent*** **EndIndent***
BLSEs containing text	**TextIndent*** **TextAlign***
Illustration elements (Figure, Formula, Form) Table	**BBox** **Width** **Height**

STRUCTURE ELEMENTS	ATTRIBUTES
TH (Table header) TD (Table data)	**Width** **Height** **BlockAlign*** **InlineAlign*** **TBorderStyle*** **TPadding***
Any ILSE BLSEs containing ILSEs or containing direct or nested content items	**LineHeight*** **BaselineShift** **TextDecorationType** **TextDecorationColor*** **TextDecorationThickness***
Vertical text	**GlyphOrientationVertical***
Ruby text	**RubyAlign*** **RubyPosition***

General Layout Attributes

The layout attributes described in Table 10.28 can apply to structure elements of any of the standard types, whether at the block level (BLSEs) or the inline level (ILSEs).

TABLE 10.28 Standard layout attributes common to all standard structure types

KEY	TYPE	VALUE
Placement	name	*(Optional; not inheritable)* The positioning of the element with respect to the enclosing reference area and other content: Block — Stacked in the block-progression direction within an enclosing reference area or parent BLSE. Inline — Packed in the inline-progression direction within an enclosing BLSE.

KEY	TYPE	VALUE
		Before — Placed so that the before edge of the element's allocation rectangle (see "Content and Allocation Rectangles" on page 801) coincides with that of the nearest enclosing reference area. The element may float, if necessary, to achieve the specified placement (see note below). The element is treated as a block occupying the full extent of the enclosing reference area in the inline direction; other content is stacked so as to begin at the after edge of the element's allocation rectangle.
		Start — Placed so that the start edge of the element's allocation rectangle (see "Content and Allocation Rectangles" on page 801) coincides with that of the nearest enclosing reference area. The element may float, if necessary, to achieve the specified placement (see note below). Other content that would intrude into the element's allocation rectangle is laid out as a runaround.
		End — Placed so that the end edge of the element's allocation rectangle (see "Content and Allocation Rectangles" on page 801) coincides with that of the nearest enclosing reference area. The element may float, if necessary, to achieve the specified placement (see note below). Other content that would intrude into the element's allocation rectangle is laid out as a runaround.
		When applied to an ILSE, any value except Inline causes the element to be treated as a BLSE instead. Default value: Inline.
		Note: *Elements with* **Placement** *values of Before, Start, or End are removed from the normal stacking or packing process and allowed to "float" to the specified edge of the enclosing reference area or parent BLSE. Multiple such floating elements may be positioned adjacent to one another against the specified edge of the reference area, or placed serially against the edge, in the order encountered. Complex cases such as floating elements that interfere with each other or do not fit on the same page may be handled differently by different layout applications; Tagged PDF merely identifies the elements as floating and indicates their desired placement.*

KEY	TYPE	VALUE
WritingMode	name	*(Optional; inheritable)* The directions of layout progression for packing of ILSEs (inline progression) and stacking of BLSEs (block progression): LrTb — Inline progression from left to right; block progression from top to bottom. This is the typical writing mode for Western writing systems. RlTb — Inline progression from right to left; block progression from top to bottom. This is the typical writing mode for Arabic and Hebrew writing systems. TbRl — Inline progression from top to bottom; block progression from right to left. This is the typical writing mode for Chinese and Japanese writing systems. The specified layout directions apply to the given structure element and all of its descendants to any level of nesting. Default value: LrTb. For elements that produce multiple columns, the writing mode defines the direction of column progression within the reference area: the inline direction determines the stacking direction for columns and the default flow order of text from column to column. For tables, the writing mode controls the layout of rows and columns: table rows (structure type TR) are stacked in the block direction, cells within a row (structure type TD) in the inline direction. ***Note:*** *The inline-progression direction specified by the writing mode is subject to local override within the text being laid out, as described in Unicode Standard Annex #9,* The Bidirectional Algorithm*, available from the Unicode Consortium (see the Bibliography).*
BackgroundColor	array	*(Optional; not inheritable; PDF 1.5)* The color to be used to fill the background of a table cell or any element's content rectangle (possibly adjusted by the **Padding** attribute). The value is an array of three numbers in the range 0.0 to 1.0, representing the red, green and blue values, respectively, of an RGB color space. If this attribute is not specified, the element is treated as if it were transparent.

KEY	TYPE	VALUE
BorderColor	array	*(Optional; inheritable; PDF 1.5)* The color of the border drawn on the edges of a table cell or any element's content rectangle (possibly adjusted by the **Padding** attribute). The value of each edge is an array of three numbers in the range 0.0 to 1.0, representing the red, green and blue values, respectively, of an RGB color space. There are two forms: • A single array of three numbers, representing the RGB values to apply to all four edges. • An array of four arrays, each specifying the RGB values for one edge of the border, in the order of the before, after, start and end edges. A value of **null** for any of the edges means that it is not to be drawn. If this attribute is not specified, the border color for this element is the current text fill color in effect at the start of its associated content.
BorderStyle	array or name	*(Optional; not inheritable; PDF 1.5)* The style of an element's border. Specifies the stroke pattern of each edge of a table cell or any element's content rectangle (possibly adjusted by the **Padding** attribute). There are two forms: • A name from the list below, representing the border style to apply to all four edges. • An array of four entries, each entry specifying the style for one edge of the border, in the order of the before, after, start and end edges. A value of **null** for any of the edges means that it is not to be drawn. None — No border. Forces the computed value of **BorderThickness** to be 0. Hidden — Same as None, except in terms of border conflict resolution for table elements. Dotted — The border is a series of dots. Dashed — The border is a series of short line segments. Solid — The border is a single line segment. Double — The border is two solid lines. The sum of the two lines and the space between them equals the value of **BorderThickness**. Groove — The border looks as though it were carved into the canvas. Ridge — The border looks as though it were coming out of the canvas (the opposite of Groove.

KEY	TYPE	VALUE
		Inset — The border makes the entire box look as though it were embedded in the canvas. Outset — The border makes the entire box look as though it were coming out of the canvas (the opposite of Inset). Default value: None ***Note:** All borders are drawn on top of the box's background. The color of borders drawn for values of Groove, Ridge, Inset and Outset depends on the structure element's **BorderColor** attribute and the color of the background over which the border is being drawn.* ***Note:** Conforming HTML applications may interpret Dotted, Dashed, Double, Groove, Ridge, Inset, and Outset to be Solid.*
BorderThickness	number or array	*(Optional; inheritable; PDF 1.5)* The thickness of the border drawn on the edges of a table cell or any element's content rectangle (possibly adjusted by the **Padding** attribute). The value of each edge is a positive number in default user space units representing the border's thickness (a value of 0 indicates the border is not drawn). There are two forms: • A number representing the border thickness for all four edges. • An array of four entries, each entry specifying the thickness for one edge of the border, in the order of the before, after, start and end edges. A value of **null** for any of the edges means that it is not to be drawn.
Padding	number or array	*(Optional; not inheritable; PDF 1.5)* Specifies an offset to account for the separation between the element's content rectangle and the surrounding border (see "Content and Allocation Rectangles" on page 801). A positive value enlarges the background area; a negative value trims it, possibly allowing the border to overlap the element's text or graphic. The value may be a single number representing the width of the padding, in default user space units, that applies to all four sides; or a 4-entry array representing the padding width for the before edge, after edge, start edge and end edge, respectively, of the content rectangle. Default value: 0.
Color	array	*(Optional; inheritable; PDF 1.5)* The color to be used for drawing text and the default value for the color of table borders and text decorations. The value is an array of three numbers in the range 0.0 to 1.0, representing the red, green and blue values, respectively, of an RGB color space. If this attribute is not specified, the border color for this element is the current text fill color in effect at the start of its associated content.

Layout Attributes for BLSEs

Table 10.29 describes layout attributes that apply only to block-level structure elements (BLSEs).

Note: *Inline-level structure elements (ILSEs) with a* ***Placement*** *attribute other than the default value of* Inline *are treated as BLSEs, and hence are also subject to the attributes described here.*

TABLE 10.29 Additional standard layout attributes specific to block-level structure elements

KEY	TYPE	VALUE
SpaceBefore	number	*(Optional; not inheritable)* The amount of extra space preceding the before edge of the BLSE, measured in default user space units in the block-progression direction. This value is added to any adjustments induced by the **LineHeight** attributes of ILSEs within the first line of the BLSE (see “Layout Attributes for ILSEs” on page 797). If the preceding BLSE has a **SpaceAfter** attribute, the greater of the two attribute values is used. Default value: 0. ***Note:*** *This attribute is disregarded for the first BLSE placed in a given reference area.*
SpaceAfter	number	*(Optional; not inheritable)* The amount of extra space following the after edge of the BLSE, measured in default user space units in the block-progression direction. This value is added to any adjustments induced by the **LineHeight** attributes of ILSEs within the last line of the BLSE (see “Layout Attributes for ILSEs” on page 797). If the following BLSE has a **SpaceBefore** attribute, the greater of the two attribute values is used. Default value: 0. ***Note:*** *This attribute is disregarded for the last BLSE placed in a given reference area.*

KEY	TYPE	VALUE
StartIndent	number	*(Optional; inheritable)* The distance from the start edge of the reference area to that of the BLSE, measured in default user space units in the inline-progression direction. This attribute applies only to structure elements with a **Placement** attribute of Block or Start (see “General Layout Attributes” on page 787); it is disregarded for those with other **Placement** values. Default value: 0. ***Note:** A negative value for this attribute places the start edge of the BLSE outside that of the reference area. The results are implementation-dependent and may not be supported by all Tagged PDF consumer applications or export formats.* ***Note:** If a structure element with a **StartIndent** attribute is placed adjacent to a floating element with a **Placement** attribute of Start, the actual value used for the element’s starting indent will be its own **StartIndent** attribute or the inline extent of the adjacent floating element, whichever is greater. This value may then be further adjusted by the element’s **TextIndent** attribute, if any.*
EndIndent	number	*(Optional; inheritable)* The distance from the end edge of the BLSE to that of the reference area, measured in default user space units in the inline-progression direction. This attribute applies only to structure elements with a **Placement** attribute of Block or End (see “General Layout Attributes” on page 787); it is disregarded for those with other **Placement** values. Default value: 0. ***Note:** A negative value for this attribute places the end edge of the BLSE outside that of the reference area. The results are implementation-dependent and may not be supported by all Tagged PDF consumer applications or export formats.* ***Note:** If a structure element with an **EndIndent** attribute is placed adjacent to a floating element with a **Placement** attribute of End, the actual value used for the element’s ending indent will be its own **EndIndent** attribute or the inline extent of the adjacent floating element, whichever is greater.*
TextIndent	number	*(Optional; inheritable; applies only to some BLSEs, as described below)* The additional distance, measured in default user space units in the inline-progression direction, from the start edge of the BLSE, as specified by **StartIndent** (above), to that of the first line of text. A negative value indicates a hanging indent. Default value: 0. This attribute applies only to paragraphlike BLSEs and those of structure types Lbl (Label), LBody (List body), TH (Table header), and TD (Table data), provided that they contain content other than nested BLSEs.

KEY	TYPE	VALUE
TextAlign	name	*(Optional; inheritable; applies only to BLSEs containing text)* The alignment, in the inline-progression direction, of text and other content within lines of the BLSE: Start — Aligned with the start edge. Center — Centered between the start and end edges. End — Aligned with the end edge. Justify — Aligned with both the start and end edges, with internal spacing within each line expanded, if necessary, to achieve such alignment. The last (or only) line is aligned with the start edge only, as for Start (above). Default value: Start.
BBox	rectangle	*(Optional for Annot; required for any figure or table appearing in its entirety on a single page; not inheritable)*. An array of four numbers in default user space units giving the coordinates of the left, bottom, right, and top edges, respectively, of the element's bounding box (the rectangle that completely encloses its visible content). This attribute applies to any element that lies on a single page and occupies a single rectangle.
Width	number or name	*(Optional; not inheritable; illustrations, tables, table headers, and table cells only; strongly recommended for table cells)* The desired width of the element's content rectangle (see "Content and Allocation Rectangles" on page 801), measured in default user space units in the inline-progression direction. This attribute applies only to elements of structure type Figure, Formula, Form, Table, TH (Table header), or TD (Table data). The name Auto in place of a numeric value indicates that no specific width constraint is to be imposed; the element's width is determined by the intrinsic width of its content. Default value: Auto.
Height	number or name	*(Optional; not inheritable; illustrations, tables, table headers, and table cells only)* The desired height of the element's content rectangle (see "Content and Allocation Rectangles" on page 801), measured in default user space units in the block-progression direction. This attribute applies only to elements of structure type Figure, Formula, Form, Table, TH (Table header), or TD (Table data). The name Auto in place of a numeric value indicates that no specific height constraint is to be imposed; the element's height is determined by the intrinsic height of its content. Default value: Auto.

KEY	TYPE	VALUE
BlockAlign	name	*(Optional; inheritable; table cells only)* The alignment, in the block-progression direction, of content within the table cell: Before — Before edge of the first child's allocation rectangle aligned with that of the table cell's content rectangle. Middle — Children centered within the table cell, so that the distance between the before edge of the first child's allocation rectangle and that of the table cell's content rectangle is the same as the distance between the after edge of the last child's allocation rectangle and that of the table cell's content rectangle. After — After edge of the last child's allocation rectangle aligned with that of the table cell's content rectangle. Justify — Children aligned with both the before and after edges of the table cell's content rectangle. The first child is placed as described above for Before and the last child as described for After, with equal spacing between the children. If there is only one child, it is aligned with the before edge only, as for Before. This attribute applies only to elements of structure type TH (Table header) or TD (Table data), and controls the placement of all BLSEs that are children of the given element. The table cell's content rectangle (see "Content and Allocation Rectangles" on page 801) becomes the reference area for all of its descendants. Default value: Before.

KEY	TYPE	VALUE
InlineAlign	name	*(Optional; inheritable; table cells only)* The alignment, in the inline-progression direction, of content within the table cell: Start — Start edge of each child's allocation rectangle aligned with that of the table cell's content rectangle Center — Each child centered within the table cell, so that the distance between the start edges of the child's allocation rectangle and the table cell's content rectangle is the same as the distance between their end edges End — End edge of each child's allocation rectangle aligned with that of the table cell's content rectangle This attribute applies only to elements of structure type TH (Table header) or TD (Table data), and controls the placement of all BLSEs that are children of the given element. The table cell's content rectangle (see "Content and Allocation Rectangles" on page 801) becomes the reference area for all of its descendants. Default value: Start.
TBorderStyle	name or array	*(Optional; inheritable; PDF 1.5)* The style of the border drawn on each edge of a table cell. The possible values are the same as those specified for **BorderStyle** (see Table 10.28). If both **TBorderStyle** and **BorderStyle** apply to a given table cell, **BorderStyle** supersedes **TBorderStyle**. Default value: None.
TPadding	integer	*(Optional; inheritable; PDF 1.5)* Specifies an offset to account for the separation between the table cell's content rectangle and the surrounding border (see "Content and Allocation Rectangles" on page 801). If both **TPadding** and **Padding** apply to a given table cell, **Padding** supersedes **TPadding**. A positive value enlarges the background area; a negative value trims it, possibly allowing the border to overlap the element's text or graphic. The value may be a single number representing the width of the padding, in default user space units, that applies to all four edges of the table cell; or a 4-entry array representing the padding width for the before edge, after edge, start edge and end edge, respectively, of the content rectangle. Default value: 0.

Layout Attributes for ILSEs

The attributes described in Table 10.30 apply to inline-level structure elements (ILSEs). They may also be specified for a block-level element (BLSE) and will apply to any content items that are its immediate children.

TABLE 10.30 Standard layout attributes specific to inline-level structure elements

KEY	TYPE	VALUE
LineHeight	number or name	*(Optional; inheritable)* The element's preferred height, measured in default user space units in the block-progression direction. The height of a line is determined by the largest **LineHeight** value for any complete or partial ILSE that it contains. The name Normal or Auto in place of a numeric value indicates that no specific height constraint is to be imposed; the element's height is set to a reasonable value based on the content's font size: Normal — Adjust the line height to include any nonzero value specified for **BaselineShift** (see below). Auto — Do not adjust for the value of **BaselineShift**. Default value: Normal. This attribute applies to all ILSEs (including implicit ones) that are children of this element or of its nested ILSEs, if any; it does not apply to nested BLSEs. ***Note:** When translating to a specific export format, the values Normal and Auto, if specified, are used directly if they are available in the target format. The meaning of the term "reasonable value," used above, is left to the consumer application to determine; it can be assumed to be approximately 1.2 times the font size, but this value may vary depending on the export format. In the absence of a numeric value for **LineHeight** or an explicit value for the font size, a reasonable method of calculating the line height from the information in a Tagged PDF file is to find the difference between the associated font's **Ascent** and **Descent** values (see Section 5.7, "Font Descriptors"), map it from glyph space to default user space (see Section 5.3.3, "Text Space Details"), and use the maximum resulting value for any character in the line.*

KEY	TYPE	VALUE
BaselineShift	number	*(Optional; not inheritable)* The distance, in default user space units, by which the element's baseline is shifted relative to that of its parent element. The shift direction is the opposite of the block-progression direction specified by the prevailing **WritingMode** attribute (see "General Layout Attributes" on page 787); thus positive values shift the baseline toward the before edge and negative values toward the after edge of the reference area (upward and downward, respectively, in Western writing systems). Default value: 0. The shifted element might be a superscript, a subscript, or an inline graphic. The shift applies to the element itself, its content, and all of its descendants. Any further baseline shift applied to a child of this element is measured relative to the shifted baseline of this (parent) element.
TextDecorationType	name	*(Optional; not inheritable)* The *text decoration*, if any, to be applied to the element's text. None — No text decoration Underline — A line below the text Overline — A line above the text LineThrough — A line through the middle of the text Default value: None. This attribute applies to all text content items that are children of this element or of its nested ILSEs, if any; it does not apply to nested BLSEs or to content items other than text. ***Note:*** *The color, position, and thickness of the decoration should be uniform across all children, in spite of changes in color, font size, or other variations in the content's text characteristics.*
TextDecorationColor	array	*(Optional; inheritable; PDF 1.5)* The color to be used for drawing text decorations. The value is an array of three numbers in the range 0.0 to 1.0, representing the red, green and blue values, respectively, of an RGB color space. If this attribute is not specified, the border color for this element is the current fill color in effect at the start of its associated content.

KEY	TYPE	VALUE
TextDecorationThickness	number	*(Optional; inheritable; PDF 1.5)* The thickness of each line drawn as part of the text decoration. The value is a non-negative number in default user space units representing the thickness (0 is interpreted as the thinnest possible line). If this attribute is not specified, it is derived from the current stroke thickness in effect at the start of the element's associated content, transformed into default user space units.
RubyAlign	name	*(Optional; inheritable; PDF 1.5)* The justification of the lines within a ruby assembly. Start — The content is to be aligned on the start edge in the inline-progression direction. Center — The content is to be centered in the inline-progression direction. End — The content is to be aligned on the end edge in the inline-progression direction. Justify — The content is to be expanded to fill the available width in the inline-progression direction. Distribute — The content is to be expanded to fill the available width in the inline-progression direction. However, some space is also inserted at the start edge and end edge of the text. Normally, the spacing is distributed using a 1:2:1 (start:infix:end) ratio; it is changed to a 0:1:1 ratio if the ruby appears at the start of a text line or to a 1:1:0 ratio if the ruby appears at the end of the text line. Default value: Distribute. This attribute may be specified on the RB and RT elements. When a ruby is formatted, the attribute is applied to the shorter line of these two elements. (If the RT element has a shorter width than the RB element, then the RT element is aligned as specified in its **RubyAlign** attribute.)

KEY	TYPE	VALUE
RubyPosition	name	*(Optional; inheritable; PDF 1.5)* The placement of the RT structure element relative to the RB element in a ruby assembly. Before — The RT content is to be aligned along the before edge of the element. After — The RT content is to be aligned along the after edge of the element. Warichu — The RT and associated RP elements are to be formatted as a warichu, following the RB element. Inline — The RT and associated RP elements are to be formatted as a parenthesis comment, following the RB element. Default value: Before.
GlyphOrientationVertical	name	*(Optional; inheritable; PDF 1.5)* Specifies the orientation of glyphs when the inline-progression direction is top to bottom or bottom to top. This attribute may take one of the following values: *angle* — A number representing the clockwise rotation in degrees of the top of the glyphs relative to the top of the reference area. Must be a multiple of 90 degrees between -180 and +360. Auto — Specifies a default orientation for text, depending on whether it is *fullwidth* (as wide as it is high). Fullwidth Latin and fullwidth ideographic text (excluding ideographic punctuation) is set with an angle of 0. Ideographic punctuation and other ideographic characters having alternate horizontal and vertical forms use the vertical form of the glyph. Non-fullwidth text is set with an angle of 90. Default value: Auto.

KEY	TYPE	VALUE
		This attribute is used most commonly to differentiate between the preferred orientation of alphabetic (non-ideographic) text in vertically written Japanese documents (Auto or 90) and the orientation of the ideographic characters and/or alphabetic (non-ideographic) text in western signage and advertising (90). It affects both the alignment and width of the glyphs. If a glyph is perpendicular to the vertical baseline, its horizontal alignment point is aligned with the alignment baseline for the script to which the glyph belongs. The width of the glyph area is determined from the horizontal width font characteristic for the glyph.

Content and Allocation Rectangles

As defined in Section 10.7.3, "Basic Layout Model," an element's *content rectangle* is an enclosing rectangle derived from the shape of the element's content, which defines the bounds used for the layout of any included child elements; the *allocation rectangle* includes any additional borders or spacing surrounding the element, affecting how it will be positioned with respect to adjacent elements and the enclosing content rectangle or reference area.

The exact definition of the content rectangle depends on the element's structure type:

- For a table cell (structure type TH or TD), the content rectangle is determined from the bounding box of all graphics objects in the cell's content, taking into account any explicit bounding boxes (such as the **BBox** entry in a form XObject). This implied size can be explicitly overridden by the cell's **Width** and **Height** attributes. The cell's height is then further adjusted to equal the maximum height of any cell in its row, and its width to the maximum width of any cell in its column.
- For any other BLSE, the height of the content rectangle is the sum of the heights of all BLSEs it contains, plus any additional spacing adjustments between these elements.
- For an ILSE that contains text, the height of the content rectangle is set by the **LineHeight** attribute. The width is determined by summing the widths of the contained characters, adjusted for any indents, letter spacing, word spacing, or line-end conditions.

- For an ILSE that contains an illustration or table, the content rectangle is determined from the bounding box of all graphics objects in the content, taking into account any explicit bounding boxes (such as the **BBox** entry in a form XObject). This implied size can be explicitly overridden by the element's **Width** and **Height** attributes.
- For an ILSE that contains a mixture of elements, the height of the content rectangle is determined by aligning the child objects relative to one another based on their text baseline (for text ILSEs) or end edge (for non-text ILSEs), along with any applicable **BaselineShift** attribute (for all ILSEs), and then finding the extreme top and bottom for all elements.

  Note: *Some applications may apply this process to all elements within the block; others may apply it on a line-by-line basis.*

The allocation rectangle is derived from the content rectangle in a way that also depends on the structure type:

- For a BLSE, the allocation rectangle is equal to the content rectangle with its before and after edges adjusted by the element's **SpaceBefore** and **SpaceAfter** attributes, if any, but with no changes to the start and end edges.
- For an ILSE, the allocation rectangle is the same as the content rectangle.

Note: *Future versions of Tagged PDF are likely to include additional attributes that can adjust all four edges of the allocation rectangle for both BLSEs and ILSEs.*

Illustration Attributes

Certain additional restrictions arise in connection with particular uses of illustration elements (structure types Figure, Formula, or Form):

- When an illustration element has a **Placement** attribute of Block, it must have a **Height** attribute with an explicitly specified numerical value (not Auto). This value is the sole source of information about the illustration's extent in the block-progression direction.
- When an illustration element has a **Placement** attribute of Inline, it must have a **Width** attribute with an explicitly specified numerical value (not Auto). This value is the sole source of information about the illustration's extent in the inline-progression direction.

- When an illustration element has a **Placement** attribute of Inline, Start, or End, the value of its **BaselineShift** attribute is used to determine the position of its after edge relative to the text baseline; **BaselineShift** is ignored for all other values of **Placement**. (An illustration element with a **Placement** value of Start can be used to create a dropped capital; one with a **Placement** value of Inline can be used to create a raised capital.)

List Attribute

The **ListNumbering** attribute, described in Table 10.31, is carried by an L (List) element, but controls the interpretation of the Lbl (Label) elements within the list's LI (List item) elements (see "List Elements" on page 772). This attribute is defined in attribute objects whose **O** (owner) entry has the value **List** (or is one of the format-specific owner names listed in Table 10.26 on page 784).

TABLE 10.31 Standard list attribute

KEY	TYPE	VALUE
ListNumbering	name	*(Optional; inheritable)* The numbering system used to generate the content of the Lbl (Label) elements in an autonumbered list, or the symbol used to identify each item in an unnumbered list: None — No autonumbering; Lbl elements (if present) contain arbitrary text not subject to any numbering scheme Disc — Solid circular bullet Circle — Open circular bullet Square — Solid square bullet Decimal — Decimal arabic numerals (1–9, 10–99, …) UpperRoman — Uppercase roman numerals (I, II, III, IV, …) LowerRoman — Lowercase roman numerals (i, ii, iii, iv, …) UpperAlpha — Uppercase letters (A, B, C, …) LowerAlpha — Lowercase letters (a, b, c, …) Default value: None. ***Note:*** *The alphabet used for UpperAlpha and LowerAlpha is determined by the prevailing **Lang** entry (see Section 10.8.1, "Natural Language Specification").* ***Note:*** *The set of possible values may be expanded as Unicode identifies additional numbering systems.*

Note: *This attribute is used to allow a content extraction tool to autonumber a list. However, the Lbl elements within the table should nevertheless contain the resulting numbers explicitly, so that the document can be reflowed or printed without the need for autonumbering.*

Table Attributes

The attributes described in Table 10.32 apply only to table cells (structure types TH and TD; see "Table Elements" on page 772). Attributes in this category are defined in attribute objects whose **O** (owner) entry has the value **Table** (or is one of the format-specific owner names listed in Table 10.26 on page 784).

TABLE 10.32 Standard table attributes

KEY	TYPE	VALUE
RowSpan	integer	*(Optional; not inheritable)* The number of rows in the enclosing table that are spanned by the cell. The cell expands by adding rows in the block-progression direction specified by the table's **WritingMode** attribute. Default value: 1.
ColSpan	integer	*(Optional; not inheritable)* The number of columns in the enclosing table that are spanned by the cell. The cell expands by adding columns in the inline-progression direction specified by the table's **WritingMode** attribute. Default value: 1.
Headers	array	*(Optional; not inheritable; PDF 1.5)* An array of strings, where each string is the element identifier (see the **ID** entry in Table 10.9) for a **TH** structure element that is a header associated with this cell. This attribute may apply to header cells (**TH**) as well as data cells (**TD**). Therefore, the headers associated with any cell are those in its **Headers** array plus those in the **Headers** array of any **TH** cells in that array, and so on recursively.
Scope	name	*(Optional; not inheritable; PDF 1.5)* A name with one of the values **Row**, **Column** or **Both**. This attribute applies only to **TH** elements, and indicates whether the header cell applies to the rest of the cells in the row that contains it, the column that contains it, or both the row and the column that contain it.

10.8 Accessibility Support

PDF includes several facilities in support of accessibility of documents to disabled users. In particular, many visually impaired computer users use screen readers to read documents aloud. To enable proper vocalization, either through a screen reader or by some more direct invocation of a text-to-speech engine, PDF supports the following features:

- Specifying the natural language used for text in a PDF document—for example, as English or Spanish (see Section 10.8.1, "Natural Language Specification")
- Providing textual descriptions for images or other items that do not translate naturally into text (Section 10.8.2, "Alternate Descriptions"), or replacement text for content that does translate into text but is represented in a nonstandard way (such as with a ligature or illuminated character; see Section 10.8.3, "Replacement Text")
- Specifying the expansion of abbreviations or acronyms (Section 10.8.4, "Expansion of Abbreviations and Acronyms")

The core of this support lies in the ability to determine the logical order of content in a PDF document, independently of the content's appearance or layout, through logical structure and Tagged PDF, as described under "Page Content Order" on page 758. An accessibility application can extract the content of a document for presentation to a disabled user by traversing the structure hierarchy and presenting the contents of each node. For this reason, producers of PDF files must ensure that all information in a document is reachable via the structure hierarchy, and they are strongly encouraged to use the facilities described in this section.

Note: *Text can be extracted from Tagged PDF documents and examined or reused for purposes other than accessibility; see Section 10.7, "Tagged PDF."*

Additional guidelines for accessibility support of content published on the World Wide Web can be found in the World Wide Web Consortium document *Web Content Accessibility Guidelines* and the documents it points to (see the Bibliography).

10.8.1 Natural Language Specification

The natural language used for text in a document is determined in a hierarchical fashion, based on whether an optional **Lang** entry *(PDF 1.4)* is present in any of several possible locations. At the highest level, the document's default language (which applies to both text strings and text within content streams) can be specified by a **Lang** entry in the document catalog (see Section 3.6.1, "Document Catalog"). Below this, the language can be specified for the following:

- Structure elements of any type (see Section 10.6.1, "Structure Hierarchy"), through a **Lang** entry in the structure element dictionary.

- Marked-content sequences that are not in the structure hierarchy (see Section 10.5, "Marked Content"), through a **Lang** entry in a property list attached to the marked-content sequence with a Span tag. (Note that although Span is also a standard structure type, as described under "Inline-Level Structure Elements" on page 775, its use here is entirely independent of logical structure.)

The following sections provide details on the value of the **Lang** entry and the hierarchical manner in which the language for text in a document is determined.

***Note:** Text strings encoded in Unicode may include an escape sequence or language tag indicating the language of the text, overriding the prevailing* **Lang** *entry (see Section 3.8.1, "Text Strings").*

Language Identifiers

The value of the **Lang** entry in the document catalog, structure element dictionary, or property list is a text string that specifies the language with a *language identifier* having the syntax defined in Internet RFC 3066, *Tags for the Identification of Languages*. This syntax, which is summarized below, is also used to identify languages in XML, according to the World Wide Web Consortium document *Extensible Markup Language (XML) 1.1*; see the Bibliography for more information about these documents. An empty string indicates that the language is unknown.

Language identifiers can be based on codes defined by the International Organization for Standardization in ISO 639 and ISO 3166 (see the Bibliography) or registered with the Internet Assigned Numbers Authority (IANA, whose Web site is located at <http://iana.org/>), or they can include codes created for private use. A language identifier consists of a primary code optionally followed by one or more subcodes (each preceded by a hyphen). The primary code can be any of the following:

- A 2-character ISO 639 language code—for example, en for English or es for Spanish
- The letter i, designating an IANA-registered identifier
- The letter x, for private use

The first subcode can be a 2-character ISO 3166 country code, as in en-US, or a 3- to 8-character subcode registered with IANA, as in en-cockney or i-cherokee (except in private identifiers, for which subcodes are not registered). Subcodes beyond the first can be any that have been registered with IANA.

Language Specification Hierarchy

The **Lang** entry in the document catalog specifies the natural language for all text in the document except where overridden by language specifications for structure elements or for marked-content sequences that are not in the structure hierarchy (for example, within an entirely unstructured document). Examples in this section illustrate the hierarchical manner in which the language for text in a document is determined.

Example 10.16 shows how a language specified for the document as a whole could be overridden by one specified for a marked-content sequence within a page's content stream, independent of any logical structure. In this case, the **Lang** entry in the document catalog (not shown) has the value en-US, meaning U.S. English, and it is overridden by the **Lang** property attached (with the Span tag) to the marked-content sequence Hasta la vista. The **Lang** property identifies the language for this marked content sequence with the value es-MX, meaning Mexican Spanish.

Example 10.16

```
2 0 obj                                         % Page object
    << /Type /Page
        /Contents 3 0 R                         % Content stream
        …
    >>
endobj

3 0 obj                                         % Page's content stream
    << /Length … >>
stream
    BT
        (See you later, or as Arnold would say, ) Tj
        /Span << /Lang (es-MX) >>               % Start of marked-content sequence
            BDC
                (Hasta la vista.) Tj
            EMC                                 % End of marked-content sequence
    ET
endstream
endobj
```

Where logical structure is described (by a structure hierarchy) within a document, the **Lang** entry in the document catalog sets the default for the document, as usual; below that, any language specifications within the structure hierarchy apply in this order:

- A structure element's language specification.

  Note: *If a structure element does not have a* **Lang** *entry, the element inherits its language from any parent element that has one.*

- Within a structure element, a language specification for a nested structure element or marked-content sequence

In Example 10.17, the **Lang** entry in the structure element dictionary (specifying English) applies to the marked-content sequence having an **MCID** (marked-content identifier) value of 0 within the indicated page's content stream. However, nested within that marked-content sequence is another one in which the **Lang** property attached with the Span tag (specifying Spanish) overrides the structure element's language specification.

Note: *This example and the next one below omit required* ***StructParents*** *entries in the objects used as content items (see "Finding Structure Elements from Content Items" on page 739).*

Example 10.17

```
1 0 obj                                      % Structure element
   << /Type /StructElem
       /S /P                                 % Structure type
       /P …                                  % Parent in structure hierarchy
       /K << /Type /MCR
              /Pg 2 0 R                      % Page containing marked-content sequence
              /MCID 0                        % Marked-content identifier
          >>
       /Lang (en-US)                         % Language specification for this element
   >>
endobj

2 0 obj                                      % Page object
   << /Type /Page
       /Contents 3 0 R                       % Content stream
       …
   >>
endobj
```

```
3 0 obj                                   % Page's content stream
   << /Length … >>
stream
   BT
      /P << /MCID 0 >>                    % Start of marked-content sequence
         BDC
            (See you later, or as Arnold would say, ) Tj
            /Span << /Lang (es-MX) >>     % Start of nested marked-content sequence
               BDC
                  (Hasta la vista.) Tj
               EMC                        % End of nested marked-content sequence
         EMC                              % End of marked-content sequence
   ET
endstream
endobj
```

If only part of the page content is contained in the structure hierarchy, and the structured content is nested within nonstructured content to which a different language specification applies, the structure element's language specification takes precedence. In Example 10.18, the page's content stream consists of a marked-content sequence that specifies (via the Span tag with a **Lang** property) Spanish as its language, but nested within it is content that is part of a structure element (indicated by the **MCID** entry in that property list), and the language specification that applies to the latter content is that of the structure element, English.

Example 10.18

```
1 0 obj                              % Structure element
   << /Type /StructElem
      /S /P                          % Structure type
      /P …                           % Parent in structure hierarchy
      /K << /Type /MCR
            /Pg 2 0 R                % Page containing marked-content sequence
            /MCID 0                  % Marked-content identifier
         >>
      /Lang (en-US)                  % Language specification for this element
   >>
endobj
```

```
2  0  obj                               % Page object
   <<  /Type  /Page
       /Contents  3 0 R                 % Content stream
       …
   >>
endobj

3  0  obj                               % Page's content stream
   <<  /Length  …  >>
stream
   /Span  << /Lang  (es-MX) >>          % Start of marked-content sequence
      BDC
         (Hasta la vista, )  Tj
         /P  << /MCID  0 >>             % Start of structured marked-content sequence,
            BDC                         %    to which structure element's language applies
               (as Arnold would say.)  Tj
            EMC                         % End of structured marked-content sequence
      EMC                               % End of marked-content sequence
endstream
endobj
```

In other words, a language identifier attached to a marked-content sequence with the Span tag specifies the language for all text in the sequence except for nested marked content that is contained in the structure hierarchy (in which case the structure element's language applies) and except where overridden by language specifications for other nested marked content.

Multi-language Text Arrays

A *multi-language text array (PDF 1.5)* allows multiple text strings to be specified, each in association with a language identifier. (See the **Alt** entry in Tables 9.9 and 9.12 for examples of its use.) The array contains pairs of strings:

- The first string in each pair is a language identifier. A given language identifier may not appear more than once in the array; any unrecognized language identifier should be ignored. An empty string specifies default text to be used when no matching language identifier is found in the array.
- The second string is text associated with the language.

Example 10.19

```
[ (en-US) (My vacation) (fr) (mes vacances) ( ) (default text) ]
```

When a viewer application searches a multi-language text array to find text for a given language, it should look for an exact (though case-insensitive) match between the given language's identifier and the language identifiers in the array. If no exact match is found, prefix matching is attempted in increasing array order: a match is declared if the given identifier is a leading, case insensitive, substring of an identifier in the array, and the first post-substring character in the array identifier is a hyphen. For example, given identifier en matches array identifier en-US, but given identifier en-US matches neither en nor en-GB. If no exact or prefix match can be found, the default text (if any) should be used.

10.8.2 Alternate Descriptions

PDF documents can be enhanced by providing alternate descriptions for images, formulas, or other items that do not translate naturally into text. Alternate descriptions are human-readable text that could, for example, be vocalized by a text-to-speech engine for the benefit of visually impaired users.

An alternate description can be specified for the following:

- A structure element (see Section 10.6.1, "Structure Hierarchy"), through an **Alt** entry in the structure element dictionary
- *(PDF 1.5)* A marked-content sequence (see Section 10.5, "Marked Content"), through an **Alt** entry in a property list attached to the marked-content sequence with a Span tag.
- Any type of annotation (see Section 8.4, "Annotations") that does not already have a text representation, through a **Contents** entry in the annotation dictionary

For annotation types that normally display text, that text (specified in the **Contents** entry of the annotation dictionary) is the natural source for vocalization purposes. For annotation types that do not display text, a **Contents** entry *(PDF 1.4)* may optionally be included to specify an alternate description. Sound annotations, which are vocalized by default and so need no alternate description for that purpose, may include a **Contents** entry specifying a description that will be displayed in a pop-up window for the benefit of hearing-impaired users.

In addition, an alternate name can be specified for an interactive form field (see Section 8.6, "Interactive Forms"), to be used in place of the actual field name wherever the field must be identified in the user interface (such as in error or status messages referring to the field). This alternate name, specified in the optional **TU** entry of the field dictionary, can be useful for vocalization purposes.

Alternate descriptions are text strings, which may be encoded in either **PDFDocEncoding** or Unicode character encoding. As described in Section 3.8.1, "Text Strings," Unicode defines an escape sequence for indicating the language of the text; this mechanism enables the alternate description to change from the language specified by the prevailing **Lang** entry (as described in the preceding section).

When applied to structure elements, the text is considered to be a word or phrase substitution for the current element. For example, if each of two (or more) elements in a sequence has an **Alt** entry in its dictionary, they should be treated as if a word break is present between them. The same would apply to consecutive marked-content sequences.

Note: *The* **Alt** *entry in property lists can be combined with other entries, as shown in Example 10.20.*

Example 10.20

```
/Span << /Lang (en-us) /Alt (six-point star) >> BDC (✡) Tj  EMC
```

10.8.3 Replacement Text

Just as alternate descriptions can be provided for images and other items that do not translate naturally into text (as described in the preceding section), replacement text can be specified for content that does translate into text but that is represented in a nonstandard way. These nonstandard representations might include, for example, glyphs for ligatures or custom characters, or inline graphics corresponding to letters in an illuminated manuscript or to dropped capitals.

Replacement text can be specified for the following:

- A structure element (see Section 10.6.1, "Structure Hierarchy"), in the optional **ActualText** entry *(PDF 1.4)* of the structure element dictionary.
- *(PDF 1.5)* A marked-content sequence (see Section 10.5, "Marked Content"), through an **ActualText** entry in a property list attached to the marked-content sequence with a Span tag.

The **ActualText** value is not a description but a replacement for the content, providing text that is equivalent to what a sighted reader would see when viewing the content. In contrast to the value of **Alt**, which is considered to be a word or phrase substitution, the value of **ActualText** is considered to be a character substitution for the structure element or marked-content sequence. Thus, if each of two (or more) consecutive structure or marked-content sequences has an **ActualText** entry, they should be treated as if no word break is present between them.

The following example shows the use of replacement text to indicate the correct character content in a case where hyphenation changes the spelling of a word (in German, the word "Drucker" when hyphenated is rendered as "Druk-" and "ker").

Example 10.21

```
(Dru) Tj
/Span
    <</Actual Text (c) >>
    BDC
        (k-) Tj
    EMC
(ker) '
```

Like alternate descriptions (and other text strings), replacement text, if encoded in Unicode, may include an escape sequence for indicating the language of the text, overriding the prevailing **Lang** entry (see Section 3.8.1, "Text Strings").

10.8.4 Expansion of Abbreviations and Acronyms

Abbreviations and acronyms can pose a problem for text-to-speech engines. Sometimes the full pronunciation for an abbreviation can be divined without aid; for instance, a search through a dictionary will probably reveal that "Blvd." is pronounced "boulevard" and that "Ave." is pronounced "avenue." However, some abbreviations are difficult to resolve, as in the sentence "Dr. Healwell works at 123 Industrial Dr." For this reason, the expansion of an abbreviation or acronym can be specified for the following:

- Marked-content sequences, through an **E** property *(PDF 1.4)* in a property list attached to the sequence with a Span tag. For example:

Example 10.22

```
BT
    /Span  << /E  (Doctor) >>
        BDC
            (Dr. )  Tj
        EMC
    (Healwell works at 123 Industrial )  Tj
    /Span  << /E  (Drive) >>
        BDC
            (Dr.)  Tj
        EMC
ET
```

- Structure elements, through an **E** entry *(PDF 1.5)* in the structure element dictionary.

The **E** value (a text string) is considered to be a word or phrase substitution for the tagged text and so should be treated as if a word break separates it from any surrounding text. Like other text strings, the expansion text, if encoded in Unicode, may include an escape sequence for indicating the language of the text (see Section 3.8.1, "Text Strings").

Some abbreviations or acronyms are conventionally not expanded into words. For the text "CBS," for example, either no expansion should be supplied (leaving its pronunciation up to the text-to-speech engine) or, to be safe, the expansion "C B S" should be specified.

10.9 Web Capture

Web Capture is a PDF 1.3 feature that allows information from Internet-based or locally resident HTML, PDF, GIF, JPEG, and ASCII text files to be imported into a PDF file. This feature is implemented in Acrobat 4.0 and later viewers by a Web Capture plug-in extension (sometimes called AcroSpider). The information in the Web Capture data structures enables viewer applications to perform the following operations:

- Save locally and preserve the visual appearance of material from the World Wide Web
- Retrieve additional material from the Web and add it to an existing PDF file
- Update or modify existing material previously captured from the Web
- Find source information for material captured from the Web, such as the URL (if any) from which it was captured
- Find all material in a PDF file that was generated from a given URL
- Find all material in a PDF file that matches a given digital identifier (MD5 hash)

The information needed to perform these operations is recorded in two data structures in the PDF file:

- The *Web Capture information dictionary* holds document-level information related to Web Capture.
- The Web Capture *content database* keeps track of the material retrieved by Web Capture and where it came from, enabling Web Capture to avoid needlessly downloading material that is already present in the file.

The following sections provide a detailed overview of these structures. See Appendix C for information about implementation limits in Web Capture.

***Note:** The following discussion centers on HTML and GIF files, although Web Capture handles other file types as well.*

10.9.1 Web Capture Information Dictionary

The optional **SpiderInfo** entry in the document catalog (see Section 3.6.1, "Document Catalog") holds an optional *Web Capture information dictionary* containing document-level information related to Web Capture. Table 10.33 shows the contents of this dictionary.

TABLE 10.33 Entries in the Web Capture information dictionary

KEY	TYPE	VALUE
V	number	*(Required)* The Web Capture version number. For PDF 1.3, the version number is 1.0. ***Note:*** *This value is a single real number, not a major and minor version number. Thus, for example, a version number of 1.2 would be considered greater than 1.15.*
C	array	*(Optional)* An array of indirect references to Web Capture command dictionaries (see "Command Dictionaries" on page 829) describing commands that were used in building the PDF file. The commands appear in the array in the order in which they were executed in building the file.

10.9.2 Content Database

Web Capture retrieves HTML files from URLs and converts them into PDF. The resulting PDF file may contain the contents of multiple HTML pages. Conversely, since HTML pages do not have a fixed size, a single HTML page may give rise to multiple PDF pages. To keep track of the correspondences, Web Capture maintains a *content database* mapping URLs and digital identifiers to PDF objects such as pages and XObjects. By looking up digital identifiers in the database, Web Capture can determine whether newly downloaded content is identical to content already retrieved from a different URL. This allows it to perform optimizations such as storing only one copy of an image that is referenced by multiple HTML pages.

Web Capture's content database is organized into *content sets*. Each content set is a dictionary holding information about a group of related PDF objects generated from the same source data. Content sets are of two subtypes: *page sets* and *image sets*. When Web Capture converts an HTML file into PDF pages, for example, it creates a page set to hold information about the pages. Similarly, when it converts a GIF image into one or more image XObjects, it creates an image set describing those XObjects.

The content set corresponding to a given data source can be accessed in either of two ways:

- By the URLs from which it was retrieved
- By a digital identifier generated from the source data itself (see "Digital Identifiers" on page 821)

The **URLS** and **IDS** entries in a PDF document's name dictionary (see Section 3.6.3, "Name Dictionary") contain name trees mapping URLs and digital identifiers, respectively, to Web Capture content sets. Figure 10.1 shows a simple example. An HTML file retrieved from the URL <http://www.adobe.com/> has been converted into three pages in the PDF file. The entry for that URL in the **URLS** name tree points to a page set containing the three pages. Similarly, the **IDS** name tree contains an entry pointing to the same page set, associated with the digital identifier calculated from the HTML source (the string shown in the figure as 904B…1EA2).

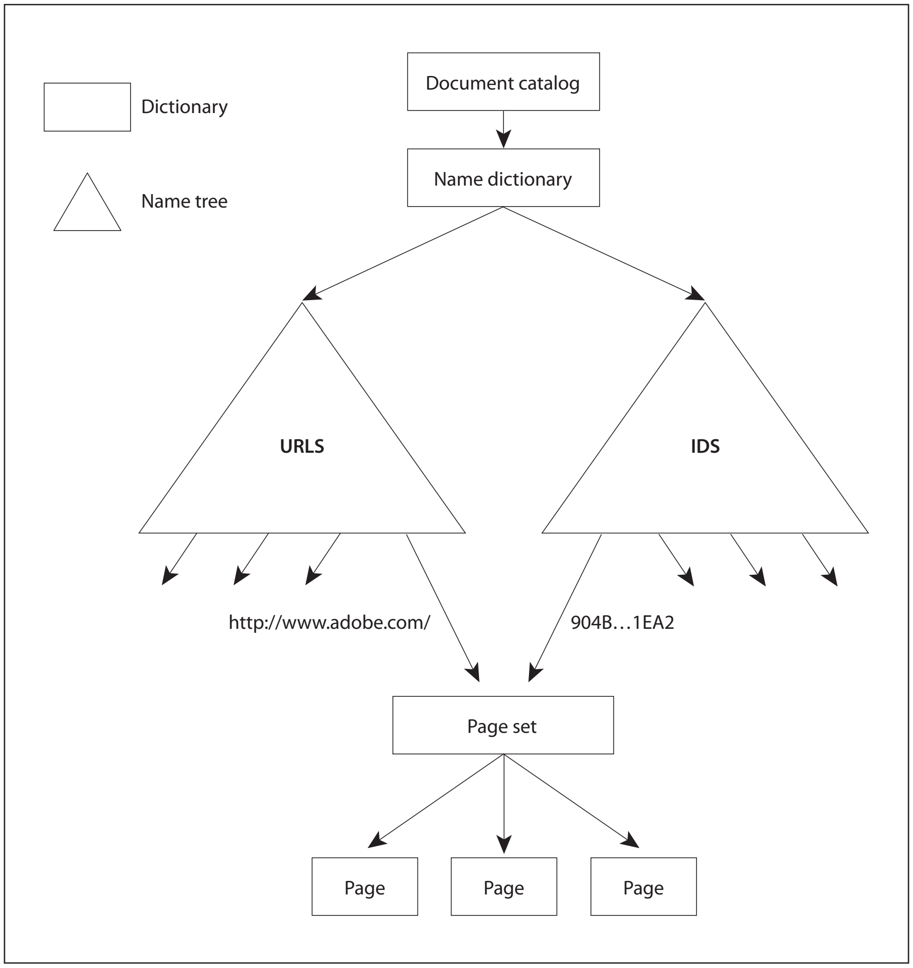


FIGURE 10.1 *Simple Web Capture file structure*

Entries in the **URLS** and **IDS** name trees may refer to an array of content sets instead of just a single content set. The content sets need not have the same subtype, but may include both page sets and image sets. In Figure 10.2, for example, a GIF file has been retrieved from a URL (<http://www.adobe.com/getacro.gif>) and converted into a single PDF page. As in Figure 10.1, a page set has been created to hold information about the new page. However, since the retrieval also resulted in a new image XObject, an image set has also been created. Instead of pointing directly to a single content set, the **URLS** and **IDS** entries point to an array containing both the page set and the image set.

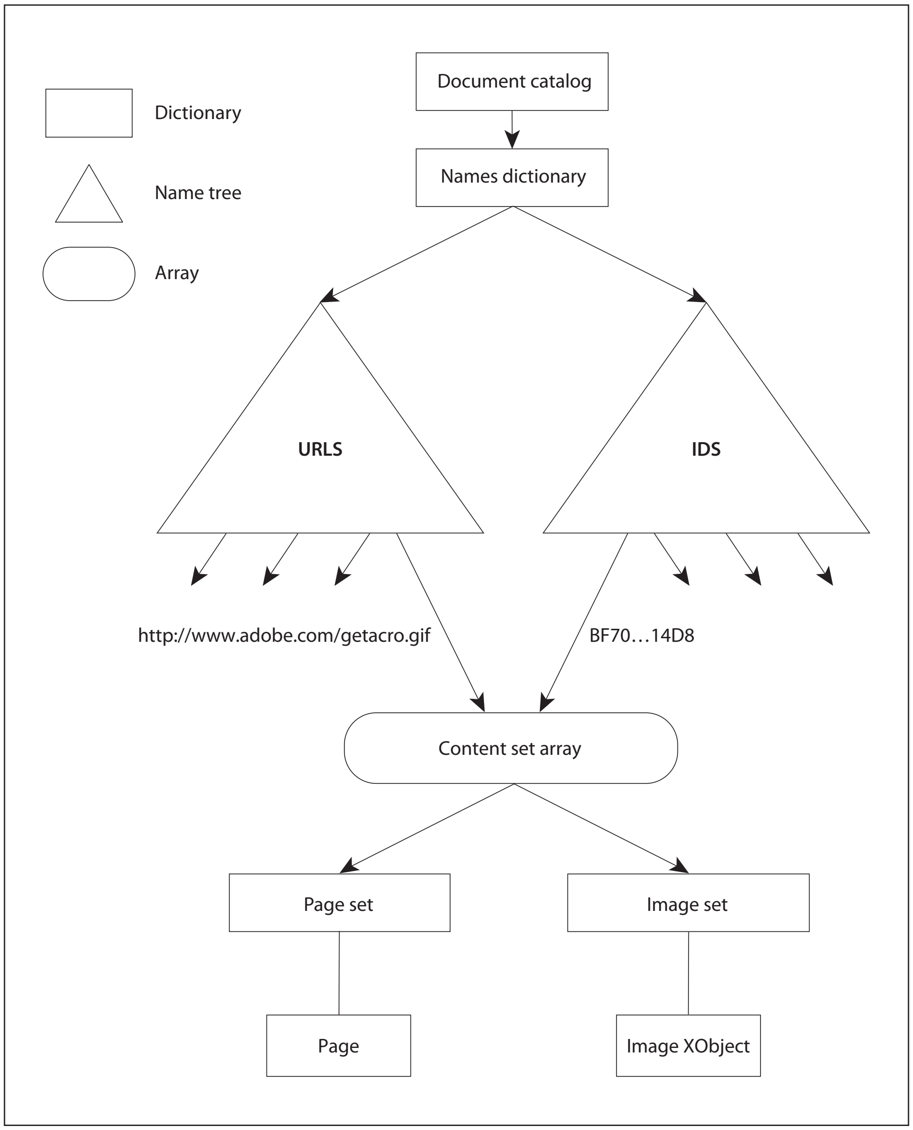


FIGURE 10.2 *Complex Web Capture file structure*

URL Strings

URLs associated with Web Capture content sets must be reduced to a predictable, canonical form before being used as keys in the **URLS** name tree. The following steps describe how to perform this reduction, using terminology from Internet RFCs 1738, *Uniform Resource Locators*, and 1808, *Relative Uniform Resource Locators* (see the Bibliography). This algorithm is relevant for HTTP, FTP, and file URLs:

1. If the URL is relative, make it absolute.
2. If the URL contains one or more number sign characters (#), strip the leftmost number sign and any characters after it.
3. Convert the scheme section to lowercase ASCII.
4. If there is a host section, convert it to lowercase ASCII.
5. If the scheme is file and the host is localhost, strip the host section.
6. If there is a port section and the port is the default port for the given protocol (80 for HTTP or 21 for FTP), strip the port section.
7. If the path section contains dot (.) or double-dot (..) subsequences, transform the path as described in section 4 of RFC 1808.

***Note:** Because the percent character (%) is "unsafe" according to RFC 1738 and is also the escape character for encoded characters, it is not possible in general to distinguish a URL with unencoded characters from one with encoded characters. For example, it is impossible to decide whether the sequence %00 represents a single encoded null character or a sequence of three unencoded characters. Hence no number of encoding or decoding passes on a URL will ever cause it to reach a stable state. Empirically, URLs embedded in HTML files have unsafe characters encoded with one encoding pass, and Web servers perform one decoding pass on received paths (though CGI scripts are free to make their own decisions). Canonical URLs are thus assumed to have undergone one and only one encoding pass. A URL whose initial encoding state is known can be safely transformed into a URL that has undergone only one encoding pass.*

Digital Identifiers

Digital identifiers associated with Web Capture content sets by the **IDS** name tree are generated using the MD5 message-digest algorithm (described in Internet

RFC 1321, *The MD5 Message-Digest Algorithm*; see the Bibliography). The exact data passed to the algorithm depends on the type of content set and the nature of the identifier being calculated.

For a page set, the source data itself is passed to the MD5 algorithm first, followed by strings representing the digital identifiers of any auxiliary data files (such as images) referenced in the source data, in the order in which they are first referenced. (If an auxiliary file is referenced more than once, its identifier is passed only the first time.) This produces a composite identifier representing the visual appearance of the pages in the page set. Two HTML source files that are identical, but for which the referenced images contain different data—for example, if they have been generated by a script or are pointed to by relative URLs—will not produce the same identifier.

***Note:** When the source data is taken from a PDF file, the identifier will be generated solely from the contents of that file; there is no auxiliary data. (See also implementation note 141 in Appendix H.)*

A page set can also have a *text identifier*, calculated by applying the MD5 algorithm to just the rendered text present in the source data. For an HTML file, for example, the text identifier is based solely on the text between markup tags; no images are used in the calculation.

For an image set, the digital identifier is calculated by passing the source data for the original image to the MD5 algorithm. For example, the identifier for an image set created from a GIF image is calculated from the contents of the GIF itself.

Unique Name Generation

In generating PDF pages from a data source, Web Capture converts items such as hypertext links and HTML form fields into corresponding named destinations and interactive form fields. These items must have names that do not conflict with those of existing items in the file. Also, when updating the file, Web Capture may need to locate all destinations and fields constructed for a given page set. Accordingly, each destination or field is given a unique name, derived from its original name but constructed so as to avoid conflicts with similarly named items in other page sets.

***Note:** As used here, the term* name *refers to a string, not a name object.*

The unique name is formed by appending an encoded form of the page set's digital identifier string to the original name of the destination or field. The identifier string must be encoded to remove characters that have special meaning in destinations and fields. For example, since the period character (`.`) is used as the field separator in interactive form field names, it must not appear in the identifier portion of the unique name; it is therefore encoded internally as two bytes, 92 and 112, corresponding to the ASCII characters `\p`. Note that since the backslash character (`\`) has special meaning for the syntax of string objects, it must be preceded by another backslash when written in the PDF file. For example, if the original digital identifier string were

```
alpha.beta
```

this would be encoded internally as

```
alpha\pbeta
```

and written in the PDF file as

```
(alpha\\pbeta)
```

Similarly, the null character (character code 0) is encoded internally as the two bytes 92 and 48, corresponding to the ASCII characters \0. If the original digital identifier string were

```
alphaØbeta
```

(where Ø denotes the null character), it would be encoded internally as

```
alpha\0beta
```

and written in the PDF file as

```
(alpha\\0beta)
```

Finally, the backslash character itself is encoded internally as the two bytes 92 and 92, corresponding to the characters \\. In written form, each of these in turn requires a preceding backslash. Thus the digital identifier string

```
alpha\beta
```

would be encoded internally as

```
alpha\\beta
```

and written in the PDF file as

```
(alpha\\\\beta)
```

If the name is used for an interactive form field, there is an additional encoding to ensure uniqueness and compatibility with interactive forms. Each byte in the source string, encoded as described above, is replaced by two bytes in the destination string. The first byte in each pair is 65 (corresponding to the ASCII character A) plus the high-order 4 bits of the source byte; the second byte is 65 plus the low-order 4 bits of the source byte.

10.9.3 Content Sets

A Web Capture *content set* is a dictionary describing a set of PDF objects generated from the same source data. It may include information common to all the objects in the set as well as about the set itself. Table 10.34 shows the contents of this type of dictionary.

Page Sets

A *page set* is a content set containing a group of PDF page objects generated from a common source, such as an HTML file. The pages are listed in the **O** array (see Table 10.34) in the same order in which they were initially added to the file. A single page object may not belong to more than one page set. Table 10.35 shows the content set dictionary entries specific to this type of content set.

The optional **TID** (text identifier) entry may be used to store an identifier generated from the text of the pages belonging to the page set (see "Digital Identifiers" on page 821). This identifier may be used, for example, to determine whether the text of a document has changed. A text identifier may not be appropriate for some page sets (such as those with no text), and should be omitted in these cases.

TABLE 10.34 Entries common to all Web Capture content sets

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **SpiderContentSet** for a Web Capture content set.
S	name	*(Required)* The subtype of content set that this dictionary describes: **SPS** (“Spider page set”) A page set **SIS** (“Spider image set”) An image set
ID	string	*(Required)* The digital identifier of the content set (see “Digital Identifiers” on page 821). If the content set has been located via the **URLS** name tree, this allows its related entry in the **IDS** name tree to be found.
O	array	*(Required)* An array of indirect references to the objects belonging to the content set. The order of objects in the array is undefined in general, but may be restricted by specific content set subtypes.
SI	dictionary or array	*(Required)* A source information dictionary (see Section 10.9.4, “Source Information”), or an array of such dictionaries, describing the sources from which the objects belonging to the content set were created.
CT	string	*(Optional)* The *content type*, a string characterizing the source from which the objects belonging to the content set were created. The string should conform to the content type specification described in Internet RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* (see the Bibliography). For example, for a page set consisting of a group of PDF pages created from an HTML file, the content type would be text/html.
TS	date	*(Optional)* A time stamp giving the date and time at which the content set was created.

TABLE 10.35 Additional entries specific to a Web Capture page set

KEY	TYPE	VALUE
S	name	*(Required)* The subtype of content set that this dictionary describes; must be **SPS** (“Spider page set”) for a page set.
T	text string	*(Optional)* The title of the page set, a text string representing it in human-readable form.
TID	string	*(Optional)* A text identifier generated from the text of the page set, as described in “Digital Identifiers” on page 821.

Image Sets

An *image set* is a content set containing a group of image XObjects generated from a common source, such as multiple frames of an animated GIF image. (Web Capture 4.0 will always generate a single image XObject for a given image.) A single XObject may not belong to more than one image set. Table 10.36 shows the content set dictionary entries specific to this type of content set.

TABLE 10.36 Additional entries specific to a Web Capture image set

KEY	TYPE	VALUE
S	name	*(Required)* The subtype of content set that this dictionary describes; must be **SIS** (“Spider image set”) for an image set.
R	integer or array	*(Required)* The reference counts (see below) for the image XObjects belonging to the image set. For an image set containing a single XObject, the value is simply the integer reference count for that XObject. If the image set contains multiple XObjects, the value is an array of reference counts parallel to the **O** array (see Table 10.34 on page 825); that is, each element in the **R** array holds the reference count for the image XObject at the corresponding position in the **O** array.

Each image XObject in an image set has a reference count indicating the number of PDF pages referring to that XObject. The reference count is incremented whenever Web Capture creates a new page referring to the XObject (including copies of already existing pages) and decremented whenever such a page is destroyed. (The reference count is incremented or decremented only once per page, regardless of the number of times the XObject may be referenced by that same page.) When the reference count reaches 0, it is assumed that there are no remaining pages referring to the XObject, and that it can be removed from the image set’s **O** array. (See implementation note 142 in Appendix H.)

10.9.4 Source Information

The **SI** entry in a content set dictionary (see Table 10.34 on page 825) identifies one or more *source information dictionaries* containing information about the locations from which the source data for the content set was retrieved. Table 10.37 shows the contents of this type of dictionary.

TABLE 10.37 Entries in a source information dictionary

KEY	TYPE	VALUE
AU	string or dictionary	*(Required)* A string or URL alias dictionary (see "URL Alias Dictionaries," below) identifying the URLs from which the source data was retrieved.
TS	date	*(Optional)* A time stamp giving the most recent date and time at which the content set's contents were known to be up to date with the source data.
E	date	*(Optional)* An expiration stamp giving the date and time at which the content set's contents should be considered out of date with the source data.
S	integer	*(Optional)* A code indicating the type of form submission, if any, by which the source data was accessed (see "Submit-Form Actions" on page 639): 0 Not accessed via a form submission 1 Accessed via an HTTP GET request 2 Accessed via an HTTP POST request This entry should be present only in source information dictionaries associated with page sets. Default value: 0.
C	dictionary	*(Optional; must be an indirect reference)* A command dictionary (see "Command Dictionaries" on page 829) describing the command that caused the source data to be retrieved. This entry should be present only in source information dictionaries associated with page sets.

In the simplest case, the content set's **SI** entry just contains a single source information dictionary. However, it is not uncommon for the same source data to be accessible via two or more unrelated URLs. When Web Capture detects such a condition (by comparing digital identifiers), it generates a single content set from the source data, containing just one copy of the relevant PDF pages or image XObjects, but creates multiple source information dictionaries describing the separate ways in which the original source data can be accessed. It then stores an array containing these multiple source information dictionaries as the value of the **SI** entry in the content set dictionary.

A source information dictionary's **AU** (aliased URLs) entry identifies the URLs from which the source data was retrieved. If there is only one such URL, a simple string suffices as the value of this entry. If multiple URLs map to the same location through redirection, the **AU** value is a URL alias dictionary representing them (see "URL Alias Dictionaries," below).

***Note:** For file size efficiency, it is recommended that the entire URL alias dictionary (excluding the URL strings) be represented as a direct object, as its internal structure should never be shared or externally referenced.*

The **TS** (time stamp) entry allows each source location associated with a content set to have its own time stamp. This is necessary because the time stamp in the content set dictionary itself (see Table 10.34 on page 825) merely refers to the creation date of the content set. A hypothetical "Update Content Set" command might reset the time stamp in the source information dictionary to the current time if it found the that the source data had not changed since the time stamp was last set.

The **E** (expiration) entry specifies an expiration date for each source location associated with a content set. If the current date and time are later than those specified, the contents of the content set should be considered out of date with the original source.

URL Alias Dictionaries

When a URL is accessed via HTTP, a response header may be returned indicating that the requested data is to be found at a different URL. This *redirection* process may be repeated in turn at the new URL, and can potentially continue indefinitely. It is not uncommon to find multiple URLs that all lead eventually to the same destination through one or more redirections. A *URL alias dictionary* represents such a set of URL chains leading to a common destination. Table 10.38 shows the contents of this type of dictionary.

TABLE 10.38 Entries in a URL alias dictionary

KEY	TYPE	VALUE
U	string	*(Required)* The destination URL to which all of the chains specified by the **C** entry lead.
C	array	*(Optional)* An array of one or more arrays of strings, each representing a chain of URLs leading to the common destination specified by **U**.

The **C** (chains) entry should be omitted if the URL alias dictionary contains only one URL. If **C** is present, its value is an array of arrays, each representing a chain of URLs leading to the common destination. Within each chain, the URLs are stored as strings in the order in which they occur in the redirection sequence. The common destination (the last URL in a chain) may be omitted, since it is already identified by the **U** entry. (See implementation note 143 in Appendix H.)

Command Dictionaries

A Web Capture *command dictionary* represents a command executed by Web Capture to retrieve one or more pieces of source data that were used to create new pages or modify existing pages. The entries in this dictionary represent parameters that were originally specified interactively by the user who requested that the Web content be captured. This information is recorded so that the command can subsequently be repeated to update the captured content. Table 10.39 shows the contents of this type of dictionary.

TABLE 10.39 Entries in a Web Capture command dictionary

KEY	TYPE	VALUE
URL	string	*(Required)* The initial URL from which source data was requested.
L	integer	*(Optional)* The number of levels of pages retrieved from the initial URL. Default value: 1.
F	integer	*(Optional)* A set of flags specifying various characteristics of the command (see Table 10.40). Default value: 0.
P	string or stream	*(Optional)* Data that was posted to the URL.
CT	string	*(Optional)* A content type describing the data posted to the URL. Default value: application/x–www–form–urlencoded.
H	string	*(Optional)* Additional HTTP request headers sent to the URL.
S	dictionary	*(Optional)* A command settings dictionary containing settings used in the conversion process (see "Command Settings" on page 831).

The **URL** entry specifies the initial URL for the retrieval command. The **L** (levels) entry specifies the number of levels of pages requested to be retrieved from this URL. If the **L** entry is omitted, its value is assumed to be 1, denoting retrieval of the initial URL only.

The value of the command dictionary's **F** entry is an unsigned 32-bit integer containing flags specifying various characteristics of the command. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order). Table 10.40 shows the meanings of the flags; all undefined flag bits are reserved and must be set to 0.

TABLE 10.40 Web Capture command flags

BIT POSITION	NAME	MEANING
1	SameSite	If set, pages were retrieved only from the host specified in the initial URL.
2	SamePath	If set, pages were retrieved only from the path specified in the initial URL (see below).
3	Submit	If set, the command represents a form submission (see below).

The SamePath flag, if set, indicates that pages were retrieved only if they were in the same path specified in the initial URL. A page is considered to be in the same path if its scheme and network location components (as defined in Internet RFC 1808, *Relative Uniform Resource Locators*) match those of the initial URL and its path component matches up to and including the last forward slash (/) character in the initial URL. For example, the URL

```
http://www.adobe.com/fiddle/faddle/foo.html
```

is considered to be in the same path as the initial URL

```
http://www.adobe.com/fiddle/initial.html
```

The comparison is case-insensitive for the scheme and network location components and case-sensitive for the path component.

If the Submit flag is set, the command represents a form submission. If no **P** (posted data) entry is present, the submitted data is encoded in the URL (an HTTP GET request). If **P** is present, the command represents an HTTP POST request. In this case, the value of the Submit flag is ignored. If the posted data is small enough, it may be represented by a string; for large amounts of data, a stream is recommended, as it can offer compression.

The **CT** (content type) entry is relevant only for POST requests. It describes the content type of the posted data, as described in Internet RFC 2045, *Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies* (see the Bibliography).

The **H** (headers) entry specifies additional HTTP request headers that were sent in the request for the URL. Each header line in the string is terminated with a carriage return and a line feed. For example:

```
(Referer: http://frumble.com\015\012From:veeble@frotz.com\015\012)
```

The HTTP request header format is specified in Internet RFC 2616, *Hypertext Transfer Protocol—HTTP/1.1* (see the Bibliography).

The **S** (settings) entry specifies a command settings dictionary (see the next section). holding settings specific to the conversion engines. If this entry is omitted, default values are assumed. It is recommended that command settings dictionaries be shared by any command dictionaries that use the same settings.

Command Settings

The **S** (settings) entry in a command dictionary contains a *command settings dictionary*, which holds settings for conversion engines used in converting the results of the command to PDF. Table 10.41 shows the contents of this type of dictionary.

TABLE 10.41 Entries in a Web Capture command settings dictionary

KEY	TYPE	VALUE
G	dictionary	*(Optional)* A dictionary containing global conversion engine settings relevant to all conversion engines. If this key is absent, default settings will be used.
C	dictionary	*(Optional)* Settings for specific conversion engines. Each key in this dictionary is the internal name of a conversion engine (see below). The associated value is a dictionary containing the settings associated with that conversion engine. If the settings for a particular conversion engine are not found in the dictionary, default settings will be used.

Each key in the **C** dictionary is the internal name of a conversion engine, which should be a name object of the following form:

/*company*:*product*:*version*:*contentType* where

company is the name (or abbreviation) of the company that created the conversion engine.

product is the name of the conversion engine. This field may be left blank, but the trailing colon character (:) is still required.

version is the version of the conversion engine.

contentType is an identifier for the content type that the settings are associated with. This is required because some converters may handle multiple content types.

For example:

```
/ADBE:H2PDF:1.0:HTML
```

Note that all fields in the internal name are case-sensitive. The *company* field must conform to the naming guidelines described in Appendix E; the values of the other fields are unrestricted, except that they must not contain a colon.

Note: *It must be possible to make a deep copy of a command settings dictionary without explicit knowledge of the settings it may contain. To facilitate this operation, the directed graph of PDF objects rooted by the command settings dictionary must be entirely self-contained; that is, it must not contain any object referred to from elsewhere in the PDF file.*

10.9.5 Object Attributes Related to Web Capture

A given page object or image XObject can belong to at most one Web Capture content set, called its *parent content set*. However, the object has no direct pointer to its parent content set; such a pointer might present problems for an application that traces all pointers from an object to determine, for example, what resources the object depends on. Instead, the object's **ID** entry (see Tables 3.27 on page 118 and 4.36 on page 303) contains the digital identifier of the parent content set, which can be used to locate the parent content set via the **IDS** name tree in the document's name dictionary. (If the **IDS** entry for the identifier contains an array of content sets, the parent can be found by searching the array for the content set whose **O** entry includes the child object.)

In the course of creating PDF pages from HTML files, Web Capture frequently scales the contents down to fit on fixed-sized pages. The **PZ** (preferred zoom) entry in a page object (see "Page Objects" on page 118) specifies a magnification factor by which the page can be scaled to undo the downscaling and view the page at its original size. That is, when the page is viewed at the preferred magnification factor, one unit in default user space will correspond to one original source pixel.

10.10 Prepress Support

This section describes features of PDF that support prepress production workflows:

- The specification of *page boundaries* governing various aspects of the prepress process, such as cropping, bleed, and trimming (Section 10.10.1, "Page Boundaries")
- Facilities for including *printer's marks* such as registration targets, gray ramps, color bars, and cut marks to assist in the production process (Section 10.10.2, "Printer's Marks")
- Information for generating *color separations* for pages in a document (Section 10.10.3, "Separation Dictionaries")
- *Output intents* for matching the color characteristics of a document with those of a target output device or production environment in which it will be printed (Section 10.10.4, "Output Intents")
- Support for the generation of *traps* to minimize the visual effects of misregistration between multiple colorants (Section 10.10.5, "Trapping Support")
- The *Open Prepress Interface (OPI)* for creating low-resolution proxies for high-resolution images (Section 10.10.6, "Open Prepress Interface (OPI)")

10.10.1 Page Boundaries

A PDF page may be prepared either for a finished medium, such as a sheet of paper, or as part of a prepress process in which the content of the page is placed on an intermediate medium, such as film or an imposed reproduction plate. In the latter case, it is important to distinguish between the intermediate page and the finished page. The intermediate page may often include additional production-related content, such as bleeds or printer marks, that falls outside the boundaries of the finished page. To handle such cases, a PDF page can define as many as five separate boundaries to control various aspects of the imaging process:

- The *media box* defines the boundaries of the physical medium on which the page is to be printed. It may include any extended area surrounding the finished page for bleed, printing marks, or other such purposes. It may also include areas close to the edges of the medium that cannot be marked because of physical limitations of the output device. Content falling outside this boundary can safely be discarded without affecting the meaning of the PDF file.
- The *crop box* defines the region to which the contents of the page are to be clipped (cropped) when displayed or printed. Unlike the other boxes, the crop box has no defined meaning in terms of physical page geometry or intended use; it merely imposes clipping on the page contents. However, in the absence of additional information (such as imposition instructions specified in a JDF or PJTF job ticket), the crop box will determine how the page's contents are to be positioned on the output medium. The default value is the page's media box.
- The *bleed box (PDF 1.3)* defines the region to which the contents of the page should be clipped when output in a production environment. This may include any extra "bleed area" needed to accommodate the physical limitations of cutting, folding, and trimming equipment. The actual printed page may include printing marks that fall outside the bleed box. The default value is the page's crop box.
- The *trim box (PDF 1.3)* defines the intended dimensions of the finished page after trimming. It may be smaller than the media box, to allow for production-related content such as printing instructions, cut marks, or color bars. The default value is the page's crop box.
- The *art box (PDF 1.3)* defines the extent of the page's meaningful content (including potential white space) as intended by the page's creator. The default value is the page's crop box.

These boundaries are specified by the **MediaBox**, **CropBox**, **BleedBox**, **TrimBox**, and **ArtBox** entries, respectively, in the page object dictionary (see Table 3.27 on page 118). All of them are rectangles expressed in default user space units. The crop, bleed, trim, and art boxes should not ordinarily extend beyond the boundaries of the media box; if they do, they will be effectively reduced to their intersection with the media box. Figure 10.3 illustrates the relationships among these boundaries. (The crop box is not shown in the figure because it has no defined relationship with any of the other boundaries.)

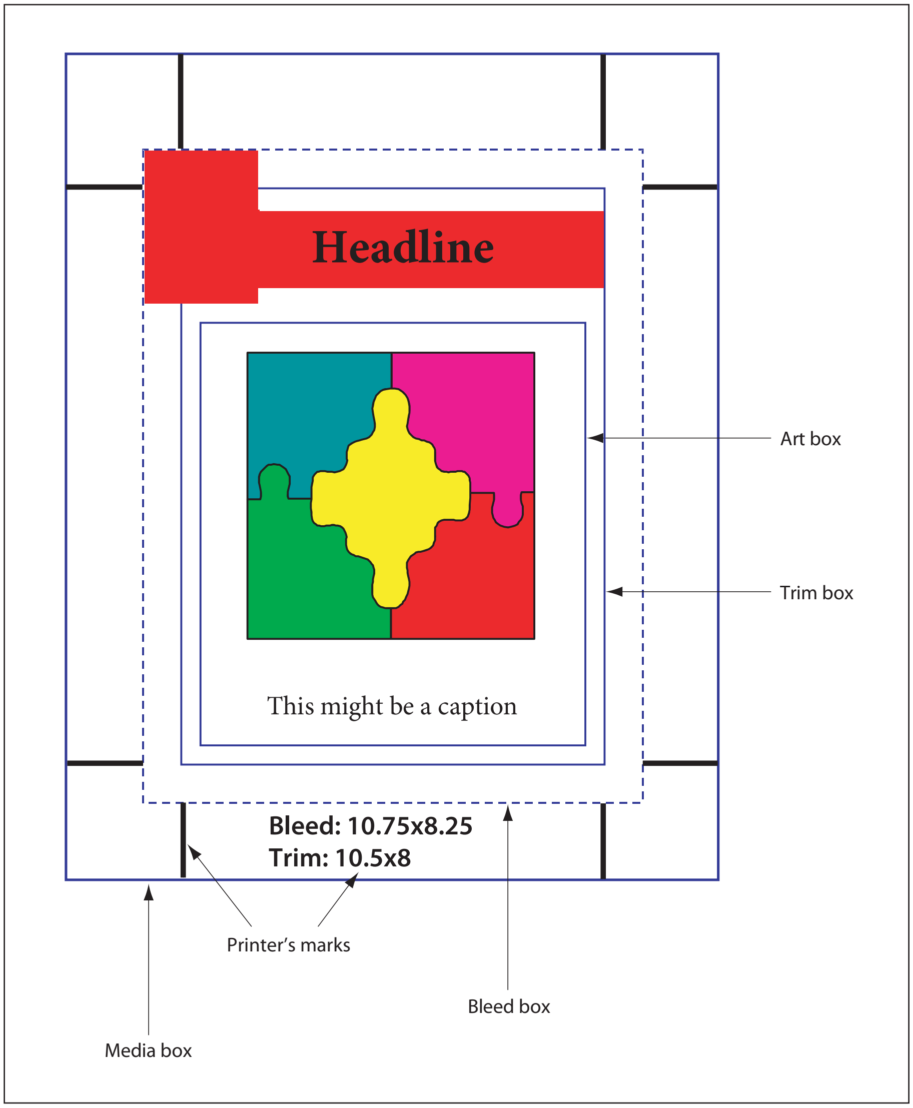


FIGURE 10.3 *Page boundaries*

How the various boundaries are used depends on the purpose to which the page is being put. Typical purposes might include the following:

- *Placing the content of a page in another application.* The art box determines the boundary of the content that is to be placed in the application. Depending on the applicable usage conventions, the placed content may be clipped to either the art box or the bleed box. (For example, a quarter-page advertisement to be placed on a magazine page might be clipped to the art box on the two sides of the ad that face into the middle of the page and to the bleed box on the two sides that bleed over the edge of the page.) The media box and trim box are ignored.
- *Printing a finished page.* This case is typical of desktop or shared page printers, in which the page content is positioned directly on the final output medium. The art box and bleed box are ignored. The media box may be used as advice for selecting media of the appropriate size. The crop box and trim box, if present, should be the same as the media box. (See implementation note 144 in Appendix H.)
- *Printing an intermediate page for use in a prepress process.* The art box is ignored. The bleed box defines the boundary of the content to be imaged. The trim box specifies the positioning of the content on the medium; it may also be used to generate cut or fold marks outside the bleed box. Content falling within the media box but outside the bleed box may or may not be imaged, depending on the specific production process being used.
- *Building an imposition of multiple pages on a press sheet.* The art box is ignored. The bleed box defines the clipping boundary of the content to be imaged; content outside the bleed box will be ignored. The trim box specifies the positioning of the page's content within the imposition. Cut and fold marks are typically generated for the imposition as a whole.

In the scenarios above, an application that interprets the bleed, trim, and art boxes for some purpose will typically alter the crop box so as to impose the clipping that those boxes prescribe.

Display of Page Boundaries

For the user's convenience, viewer applications may wish to offer the ability to display guidelines on the screen for the various page boundaries. The optional **BoxColorInfo** entry in a page object (see "Page Objects" on page 118) holds a *box* *color information dictionary (PDF 1.4)* specifying the colors and other visual characteristics to be used for such display. Viewer applications will typically provide a user interface to allow the user to set these characteristics interactively. Note that this information is page-specific and can vary from one page to another.

As shown in Table 10.42, the box color information dictionary contains an optional entry for each of the possible page boundaries other than the media box; the value of each entry is in turn a *box style dictionary*, whose contents are shown in Table 10.43. If a given entry is absent, the viewer application should use its own current default settings instead.

10.10.2 Printer's Marks

Printer's marks are graphic symbols or text added to a page to assist production personnel in identifying components of a multiple-plate job and maintaining consistent output during production. Examples commonly used in the printing industry include:

- Registration targets for aligning plates
- Gray ramps and color bars for measuring colors and ink densities
- Cut marks showing where the output medium is to be trimmed

Although PDF producer applications traditionally include such marks in the content stream of a document, they are logically separate from the content of the page itself and typically appear outside the boundaries (the crop box, trim box, and art box) defining the extent of that content (see Section 10.10.1, "Page Boundaries").

Printer's mark annotations (PDF 1.4) provide a mechanism for incorporating printer's marks into the PDF representation of a page, while keeping them separate from the actual page content. Each page in a PDF document may contain any number of such annotations, each of which represents a single printer's mark.

***Note:** Because printer's marks typically fall outside the page's content boundaries, each mark must be represented as a separate annotation. Otherwise—if, for example, the cut marks at the four corners of the page were defined in a single annotation—the annotation rectangle would encompass the entire contents of the page and could interfere with the user's ability to select content or interact with other annotations on the page. Defining printer's marks in separate annotations also facilitates the implementation of a drag-and-drop user interface for specifying them.*

TABLE 10.42 Entries in a box color information dictionary

KEY	TYPE	VALUE
CropBox	dictionary	*(Optional)* A box style dictionary (see Table 10.43) specifying the visual characteristics for displaying guidelines for the page's crop box. This entry is ignored if no crop box is defined in the page object.
BleedBox	dictionary	*(Optional)* A box style dictionary (see Table 10.43) specifying the visual characteristics for displaying guidelines for the page's bleed box. This entry is ignored if no bleed box is defined in the page object.
TrimBox	dictionary	*(Optional)* A box style dictionary (see Table 10.43) specifying the visual characteristics for displaying guidelines for the page's trim box. This entry is ignored if no trim box is defined in the page object.
ArtBox	dictionary	*(Optional)* A box style dictionary (see Table 10.43) specifying the visual characteristics for displaying guidelines for the page's art box. This entry is ignored if no art box is defined in the page object.

TABLE 10.43 Entries in a box style dictionary

KEY	TYPE	VALUE
C	array	*(Optional)* An array of three numbers in the range 0.0 to 1.0, representing the components in the **DeviceRGB** color space of the color to be used for displaying the guidelines. Default value: [0.0 0.0 0.0].
W	number	*(Optional)* The guideline width in default user space units. Default value: 1.
S	name	*(Optional)* The guideline style: S (Solid) A solid rectangle. D (Dashed) A dashed rectangle. The dash pattern is specified by the **D** entry (see below). Other guideline styles may be defined in the future. Default value: S.
D	array	*(Optional)* A *dash array* defining a pattern of dashes and gaps to be used in drawing dashed guidelines (guideline style D above). The dash array is specified in default user space units, in the same format as in the line dash pattern parameter of the graphics state (see "Line Dash Pattern" on page 187). The dash phase is not specified and is assumed to be 0. For example, a **D** entry of [3 2] specifies guidelines drawn with 3-point dashes alternating with 2-point gaps. Default value: [3].

The visual presentation of a printer's mark is defined by a form XObject specified as an appearance stream in the **N** (normal) entry of the printer's mark annotation's appearance dictionary (see Section 8.4.4, "Appearance Streams"). More than one appearance may be defined for the same printer's mark, to meet the requirements of different regions or production facilities; in this case, the appearance dictionary's **N** entry holds a subdictionary containing the alternate appearances, each identified by an arbitrary key. The **AS** (appearance state) entry in the annotation dictionary designates one of them to be displayed or printed.

Note: *The printer's mark annotation's appearance dictionary may include* ***R*** *(rollover) or* ***D*** *(down) entries, but appearances defined in either of these entries will never be displayed or printed.*

Like all annotations, a printer's mark annotation is defined by an annotation dictionary (see Section 8.4.1, "Annotation Dictionaries"); its annotation type is **PrinterMark**. The **AP** (appearances) and **F** (flags) entries (which ordinarily are optional) must be present, as must the **AS** (appearance state) entry if the appearance dictionary **AP** contains more than one appearance stream. The Print and ReadOnly flags in the **F** entry must be set and all others clear (see Section 8.4.2, "Annotation Flags"). Table 10.44 shows an additional annotation dictionary entry specific to this type of annotation.

TABLE 10.44 Additional entries specific to a printer's mark annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **PrinterMark** for a printer's mark annotation.
MN	name	*(Optional)* An arbitrary name identifying the type of printer's mark, such as ColorBar or RegistrationTarget.

The form dictionary defining a printer's mark can contain the optional entries shown in Table 10.45 in addition to the standard ones common to all form dictionaries (see Section 4.9.1, "Form Dictionaries").

TABLE 10.45 Additional entries specific to a printer's mark form dictionary

KEY	TYPE	VALUE
MarkStyle	text string	*(Optional; PDF 1.4)* A text string representing the printer's mark in human-readable form, suitable for presentation to the user on the screen.

KEY	TYPE	VALUE
Colorants	dictionary	*(Optional; PDF 1.4)* A dictionary identifying the individual colorants associated with a printer's mark such as a color bar. For each entry in this dictionary, the key is a colorant name and the value is an array defining a **Separation** color space for that colorant (see "Separation Color Spaces" on page 234). The key must match the colorant name given in that color space.

10.10.3 Separation Dictionaries

In high-end printing workflows, pages are ultimately produced as sets of *separations*, one per colorant (see "Separation Color Spaces" on page 234). Ordinarily, each page in a PDF file is treated as a composite page that paints graphics objects using all the process colorants and perhaps some spot colorants as well. In other words, all separations for a page are generated from a single PDF description of that page.

In some workflows, however, pages are *pre-separated* prior to the generation of the PDF file. In a pre-separated PDF file, the separations for a page are described as separate page objects, each painting only a single colorant (usually specified in the **DeviceGray** color space). When this is done, additional information is needed to identify the actual colorant associated with each separation and to group together the page objects representing all the separations for a given page. This information is contained in a *separation dictionary (PDF 1.3)* in the **SeparationInfo** entry of each page object (see "Page Objects" on page 118). Table 10.46 shows the contents of this type of dictionary.

TABLE 10.46 Entries in a separation dictionary

KEY	TYPE	VALUE
Pages	array	*(Required)* An array of indirect references to page objects representing separations of the same document page. One of the page objects in the array must be the one with which this separation dictionary is associated, and all of them must have separation dictionaries (**SeparationInfo** entries) containing **Pages** arrays identical to this one.
DeviceColorant	name or string	*(Required)* The name of the device colorant to be used in rendering this separation, such as Cyan or PANTONE 35 CV.

KEY	TYPE	VALUE
ColorSpace	array	*(Optional)* An array defining a **Separation** or **DeviceN** color space (see “Separation Color Spaces” on page 234 and “DeviceN Color Spaces” on page 238). This provides additional information about the color specified by **DeviceColorant**—in particular, the alternate color space and tint transformation function that would be used to represent the colorant as a process color. This information enables a viewer application to preview the separation in a color that approximates the device colorant. The value of **DeviceColorant** must match the space’s colorant name (if it is a **Separation** space) or be one of the space’s colorant names (if it is a **DeviceN** space).

10.10.4 Output Intents

Output intents (PDF 1.4) provide a means for matching the color characteristics of a PDF document with those of a target output device or production environment in which the document will be printed. The optional **OutputIntents** entry in the document catalog (see Section 3.6.1, “Document Catalog”) holds an array of *output intent dictionaries*, each describing the color reproduction characteristics of a possible output device or production condition. The contents of these dictionaries can vary for different devices and conditions; the dictionary’s **S** entry specifies an *output intent subtype* that determines the format and meaning of the remaining entries.

This use of multiple output intents allows the production process to be customized to the expected workflow and the specific tools available. For example, one production facility might process files conforming to a recognized format standard such as PDF/X-1, while another uses custom Acrobat plug-in extensions to produce *RGB* output for document distribution on the World Wide Web; each of these workflows would require different sets of output intent information. Multiple output intents also allow the same PDF file to be distributed unmodified to multiple production facilities. The choice of which output intent to use in a given production environment is a matter for agreement between the purchaser and provider of production services; PDF intentionally does not include a selector for choosing a particular output intent from within the PDF file itself.

At the time of publication, only one output intent subtype, **GTS_PDFX**, has been defined, corresponding to the PDF/X format standard (available on the World Wide Web at <http://www.npes.org/standards/>); Table 10.47 shows the contents of this type of output intent dictionary. Other subtypes may be added in the future; the names of any such additional subtypes must conform to the naming guidelines described in Appendix E.

TABLE 10.47 Entries in a PDF/X output intent dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* The type of PDF object that this dictionary describes; if present, must be **OutputIntent** for an output intent dictionary.
S	name	*(Required)* The output intent subtype; must be **GTS_PDFX** for a PDF/X output intent.
OutputCondition	text string	*(Optional)* A text string concisely identifying the intended output device or production condition in human-readable form. This is the preferred method of defining such a string for presentation to the user.
OutputConditionIdentifier	string	*(Required)* A string identifying the intended output device or production condition in human- or machine-readable form. If human-readable, this string may be used in lieu of an **OutputCondition** string for presentation to the user. A typical value for this entry would be the name of a production condition maintained in an industry-standard registry such as the *ICC Characterization Data Registry* (see the Bibliography). If the designated condition matches that in effect at production time, it is the responsibility of the production software to provide the corresponding ICC profile as defined in the registry. If the intended production condition is not a recognized standard, the value Custom is recommended for this entry; the **DestOutputProfile** entry defines the ICC profile and the **Info** entry is used for further human-readable identification.
RegistryName	string	*(Optional)* A string (conventionally a uniform resource identifier, or URI) identifying the registry in which the condition designated by **OutputConditionIdentifier** is defined.
Info	text string	*(Required if **OutputConditionIdentifier** does not specify a standard production condition; optional otherwise)* A human-readable text string containing additional information or comments about the intended target device or production condition.

KEY	TYPE	VALUE
DestOutputProfile	stream	*(Required if **OutputConditionIdentifier** does not specify a standard production condition; optional otherwise)* An ICC profile stream defining the transformation from the PDF document's source colors to output device colorants. The format of the profile stream is the same as that used in specifying an **ICCBased** color space (see "ICCBased Color Spaces" on page 222). The output transformation uses the profile's "from CIE" information (*BToA* in ICC terminology); the "to CIE" (*AToB*) information can optionally be used to remap source color values to some other destination color space, such as for screen preview or hardcopy proofing. (See implementation note 145 in Appendix H.)

***Note:** PDF/X is actually a family of standards representing varying levels of conformance. The standard for a given conformance level may prescribe further restrictions on the usage and meaning of entries in the output intent dictionary. Any such restrictions take precedence over the descriptions given in Table 10.47.*

The ICC profile information in an output intent dictionary supplements rather than replaces that in an **ICCBased** or default color space (see "ICCBased Color Spaces" on page 222 and "Default Color Spaces" on page 227). Those mechanisms are specifically intended for describing the characteristics of source color component values; an output intent can be used in conjunction with them to convert source colors to those required for a specific production condition or to enable the display or proofing of the intended output.

The data in an output intent dictionary is provided for informational purposes only, and PDF consumer applications are free to disregard it. In particular, there is no expectation that PDF production tools will automatically convert colors expressed in the same source color space to the specified target space before generating output. (In some workflows, such conversion may, in fact, be undesirable. For example, when working with *CMYK* source colors tagged with a source ICC profile solely for purposes of characterization, converting such colors from four components to three and back is unnecessary and will result in a loss of fidelity in the values of the black component; see "Implicit Conversion of CIE-Based Color Spaces" on page 228 for further discussion.) On the other hand, when source colors are expressed in different base color spaces—for example, when combining separately generated images on the same PDF page—it is possible (though not required) to use the destination profile specified in the output intent dictionary to convert source colors to the same target color space. (See implementation note 146 in Appendix H.)

Example 10.23 shows a PDF/X output intent dictionary based on an industry-standard production condition (CGATS TR 001) from the *ICC Characterization Data Registry*; Example 10.24 shows one for a custom production condition.

Example 10.23

```
<<  /Type /OutputIntent                              % Output intent dictionary
    /S /GTS_PDFX
    /OutputCondition (CGATS TR 001 (SWOP))
    /OutputConditionIdentifier (CGATS TR 001)
    /RegistryName (http://www.color.org)
    /DestOutputProfile 100 0 R
>>

100 0 obj                                            % ICC profile stream
  <<  /N 4
      /Length 1605
      /Filter /ASCIIHexDecode
  >>
stream
00 00 02 0C 61 70 … >
endstream
endobj
```

Example 10.24

```
<<  /Type /OutputIntent                              % Output intent dictionary
    /S /GTS_PDFX
    /OutputCondition (Coated)
    /OutputConditionIdentifier (Custom)
    /Info (Coated 150lpi)
    /DestOutputProfile 100 0 R
>>

100 0 obj                                            % ICC profile stream
  <<  /N 4
      /Length 1605
      /Filter /ASCIIHexDecode
  >>
```

```
stream
00 00 02 0C 61 70 … >
endstream
endobj
```

10.10.5 Trapping Support

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 10.4 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.)

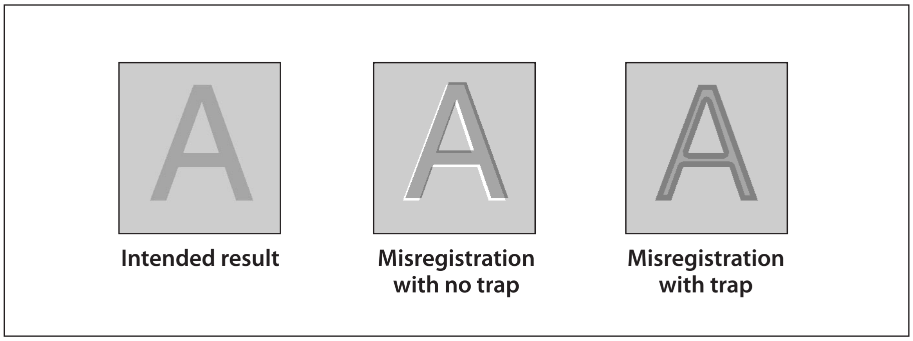


FIGURE 10.4 *Trapping example*

Trapping can be implemented by the application generating a PDF file, by some intermediate application that adds traps to a PDF document, or by the raster image processor (RIP) that produces final output. In the last two cases, the trapping process is controlled by a set of *trapping instructions*, which define two kinds of information:

- *Trapping zones* within which traps should be created
- *Trapping parameters* specifying the nature of the traps within each zone

Trapping zones and trapping parameters are discussed fully in Sections 6.3.2 and 6.3.3, respectively, of the *PostScript Language Reference*, Third Edition. Trapping instructions are not directly specified in a PDF file (as they are in a PostScript file); instead, they are specified in a *job ticket* that accompanies the PDF file or can be embedded within it. Various standards exist for the format of job tickets; two of them, JDF (Job Definition Format) and PJTF (Portable Job Ticket Format), are described in the CIP4 document *JDF Specification* and in Adobe Technical Note #5620, *Portable Job Ticket Format* (see the Bibliography).

When trapping is performed before the production of final output, the resulting traps are placed in the PDF file for subsequent use. The traps themselves are described as a content stream in a trap network annotation (see below). The stream dictionary can include additional entries describing the method that was used to produce the traps and other information about their appearance.

Trap Network Annotations

A complete set of traps generated for a given page under a specified set of trapping instructions is called a *trap network (PDF 1.3)*. It is a form XObject containing graphics objects for painting the required traps on the page. A page may have more than one trap network based on different trapping instructions, presumably intended for different output devices. All of the trap networks for a given page are contained in a single *trap network annotation* (see Section 8.4, "Annotations"). There can be at most one trap network annotation per page, which must be the last element in the page's **Annots** array (see "Page Objects" on page 118). This ensures that the trap network is printed after all of the page's other contents. (See implementation note 147 in Appendix H.)

The form XObject defining a trap network is specified as an appearance stream in the **N** (normal) entry of the trap network annotation's appearance dictionary (see Section 8.4.4, "Appearance Streams"). If more than one trap network is defined for the same page, the **N** entry holds a subdictionary containing the alternate trap networks, each identified by an arbitrary key. The **AS** (appearance state) entry in the annotation dictionary designates one of them as the *current trap network* to be displayed or printed.

***Note:** The trap network annotation's appearance dictionary may include* **R** *(rollover) or* **D** *(down) entries, but appearances defined in either of these entries will never be printed.*

Like all annotations, a trap network annotation is defined by an annotation dictionary (see Section 8.4.1, "Annotation Dictionaries"); its annotation type is **TrapNet**. The **AP** (appearances), **AS** (appearance state), and **F** (flags) entries (which ordinarily are optional) must be present, with the Print and ReadOnly flags set and all others clear (see Section 8.4.2, "Annotation Flags"). Table 10.48 shows the additional annotation dictionary entries specific to this type of annotation.

The **Version** and **AnnotStates** entries, if present, are used to detect changes in the content of a page that might require regenerating its trap networks. The **Version** array identifies elements of the page's content that might be changed by an editing application and thus invalidate its trap networks. Because there is at most one **Version** array per trap network annotation (and thus per page), any application generating a new trap network must also verify the validity of existing trap networks by enumerating the objects identified in the array and verifying that the results exactly match the array's current contents. Any trap networks found to be invalid must be regenerated. (See implementation notes 148 and 149 in Appendix H.)

Beginning with PDF 1.4, the **LastModified** entry can be used in place of the **Version** array to track changes to a page's trap network. (The trap network annotation must include either a **LastModified** entry or the combination of **Version** and **AnnotStates**, but not all three.) If the modification date in the **LastModified** entry of the page object (see "Page Objects" on page 118) is more recent than the one in the trap network annotation dictionary, then the page's trap networks are invalid and must be regenerated. Note, however, that not all editing applications and plug-in extensions correctly maintain these modification dates; this method of tracking trap network modifications can be used reliably only in a controlled workflow environment where the integrity of the modification dates is assured.

TABLE 10.48 Additional entries specific to a trap network annotation

KEY	TYPE	VALUE
Subtype	name	*(Required)* The type of annotation that this dictionary describes; must be **TrapNet** for a trap network annotation.
LastModified	date	*(Required if **Version** and **AnnotStates** are absent; must be absent if **Version** and **AnnotStates** are present; PDF 1.4)* The date and time (see Section 3.8.3, "Dates") when the trap network was most recently modified.
Version	array	*(Required if **AnnotStates** is present; must be absent if **LastModified** is present)* An unordered array of all objects present in the page description at the time the trap networks were generated and that, if changed, could affect the appearance of the page. If present, the array must include the following objects: • All content streams identified in the page object's **Contents** entry (see "Page Objects" on page 118) • All resource objects (other than procedure sets) in the page's resource dictionary (see Section 3.7.2, "Resource Dictionaries") • All resource objects (other than procedure sets) in the resource dictionaries of any form XObjects on the page (see Section 4.9, "Form XObjects") • All OPI dictionaries associated with XObjects on the page (see Section 10.10.6, "Open Prepress Interface (OPI)")
AnnotStates	array	*(Required if **Version** is present; must be absent if **LastModified** is present)* An array of name objects representing the appearance states (value of the **AS** entry) for annotations associated with the page. The appearance states must be listed in the same order as the annotations in the page's **Annots** array (see "Page Objects" on page 118). For an annotation with no **AS** entry, the corresponding array element should be **null**. No appearance state should be included for the trap network annotation itself.
FontFauxing	array	*(Optional)* An array of font dictionaries representing fonts that were "fauxed" (replaced by substitute fonts) during the generation of trap networks for the page.

Trap Network Appearances

Each entry in the **N** (normal) subdictionary of a trap network annotation's appearance dictionary holds an appearance stream defining a trap network associated with the given page. Like all appearances, a trap network is a stream object defining a form XObject (see Section 4.9, "Form XObjects"). The body of the stream contains the graphics objects needed to paint the traps making up the trap network. Its dictionary entries include, besides the standard entries for a form dictionary, the additional entries shown in Table 10.49.

TABLE 10.49 Additional entries specific to a trap network appearance stream

KEY	TYPE	VALUE
PCM	name	*(Required)* The name of the process color model that was assumed when this trap network was created; equivalent to the PostScript page device parameter **ProcessColorModel** (see Section 6.2.5 of the *PostScript Language Reference*, Third Edition). Valid values are **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, **DeviceCMY**, **DeviceRGBK**, and **DeviceN**.
SeparationColorNames	array	*(Optional)* An array of names identifying the colorants that were assumed when this network was created; equivalent to the PostScript page device parameter of the same name (see Section 6.2.5 of the *PostScript Language Reference*, Third Edition). Colorants implied by the process color model **PCM** are available automatically and need not be explicitly declared. If this entry is absent, the colorants implied by **PCM** are assumed.
TrapRegions	array	*(Optional)* An array of indirect references to **TrapRegion** objects defining the page's trapping zones and the associated trapping parameters, as described in Adobe Technical Note #5620, *Portable Job Ticket Format*. These references are to objects comprising portions of a PJTF job ticket that is embedded in the PDF file. When the trapping zones and parameters are defined by an external job ticket (or by some other means, such as with JDF), this entry is absent.
TrapStyles	text string	*(Optional)* A human-readable text string that applications can use to describe this trap network to the user (for example, to allow switching between trap networks).

***Note:** Pre-separated PDF files (see Section 10.10.3, "Separation Dictionaries") cannot be trapped, because traps are defined along the borders between different colors and a pre-separated file uses only one color. Pre-separation must therefore occur after trapping, not before. An application pre-separating a trapped PDF file is responsible for calculating new **Version** arrays for the separated trap networks.*

10.10.6 Open Prepress Interface (OPI)

The workflow in a prepress environment often involves multiple applications in areas such as graphic design, page layout, word processing, photo manipulation, and document construction. As pieces of the final document are moved from one application to another, it is useful to separate the data of high-resolution images, which can be quite large—in some cases, many times the size of the rest of the document combined—from that of the document itself. The *Open Prepress Interface (OPI)* is a mechanism, originally developed by Aldus® Corporation, for creating low-resolution placeholders, or *proxies*, for such high-resolution images. The proxy typically consists of a downsampled version of the full-resolution image, to be used for screen display and proofing. Before the document is printed, it passes through a filter known as an *OPI server*, which replaces the proxies with the original full-resolution images.

In PostScript programs, OPI proxies are defined by PostScript code surrounded by special *OPI comments*, which specify such information as the placement and cropping of the image and adjustments to its size, rotation, color, and other attributes. In PDF, proxies are embedded in a document as image or form XObjects with an associated *OPI dictionary (PDF 1.2)* containing the same information conveyed in PostScript by the OPI comments. Two versions of OPI are supported, versions 1.3 and 2.0. In OPI 1.3, a proxy consisting of a single image, with no changes in the graphics state, may be represented as an image XObject; otherwise it must be a form XObject. In OPI 2.0, the proxy always entails changes in the graphics state and hence must be represented as a form XObject. (See implementation notes 150 and 151 in Appendix H.)

An XObject representing an OPI proxy must contain an **OPI** entry in its image or form dictionary (see Tables 4.36 on page 303 and 4.42 on page 321). The value of this entry is an *OPI version dictionary* (Table 10.50) identifying the version of OPI to which the proxy corresponds. This dictionary consists of a single entry, whose key is the name **1.3** or **2.0** and whose value is the OPI dictionary defining the proxy's OPI attributes.

TABLE 10.50 Entry in an OPI version dictionary

KEY	TYPE	VALUE
version number	dictionary	*(Required; PDF 1.2)* An OPI dictionary specifying the attributes of this proxy (see Tables 10.51 and 10.52). The key for this entry must be the name **1.3** or **2.0**, identifying the version of OPI to which the proxy corresponds.

***Note:** As in any other PDF dictionary, the key in an OPI version dictionary must be a name object. The OPI version dictionary would thus be written in the PDF file in either the form*

```
<< /1.3  d 0 R >>                    % OPI 1.3 dictionary
```

or

```
<< /2.0  d 0 R >>                    % OPI 2.0 dictionary
```

where d *is the object number of the corresponding OPI dictionary.*

Tables 10.51 and 10.52 describe the contents of the OPI dictionaries for OPI 1.3 and OPI 2.0, respectively, along with the corresponding PostScript OPI comments. The dictionary entries are listed in the order in which the corresponding OPI comments should appear in a PostScript program. Complete details on the meanings of these entries and their effects on OPI servers can be found in *OPI: Open Prepress Interface Specification 1.3* (available from Adobe) and Adobe Technical Note #5660, *Open Prepress Interface (OPI) Specification, Version 2.0*.

TABLE 10.51 Entries in a version 1.3 OPI dictionary

KEY	TYPE	OPI COMMENT	VALUE
Type	name		*(Optional)* The type of PDF object that this dictionary describes; if present, must be **OPI** for an OPI dictionary.
Version	number		*(Required)* The version of OPI to which this dictionary refers; must be the number 1.3 (not the name 1.3, as in an OPI version dictionary).
F	file specification	%ALDImageFilename	*(Required)* The external file containing the image corresponding to this proxy. (See implementation note 152 in Appendix H.)
ID	string	%ALDImageID	*(Optional)* An identifying string denoting the image.
Comments	text string	%ALDObjectComments	*(Optional)* A human-readable comment, typically containing instructions or suggestions to the operator of the OPI server on how to handle the image.

KEY	TYPE	OPI COMMENT	VALUE
Size	array	%ALDImageDimensions	*(Required)* An array of two integers of the form [*pixelsWide pixelsHigh*] specifying the dimensions of the image in pixels.
CropRect	rectangle	%ALDImageCropRect	*(Required)* An array of four integers of the form [*left top right bottom*] specifying the portion of the image to be used.
CropFixed	array	%ALDImageCropFixed	*(Optional)* An array with the same form and meaning as **CropRect**, but expressed in real numbers instead of integers. Default value: the value of **CropRect**.
Position	array	%ALDImagePosition	*(Required)* An array of eight numbers of the form [ll_x ll_y ul_x ul_y ur_x ur_y lr_x lr_y] specifying the location on the page of the cropped image, where (ll_x, ll_y) are the user space coordinates of the lower-left corner, (ul_x, ul_y) those of the upper-left corner, (ur_x, ur_y) those of the upper-right corner, and (lr_x, lr_y) those of the lower-right corner. The specified coordinates must define a parallelogram; that is, they must satisfy the conditions $ul_x - ll_x = ur_x - lr_x$ and $ul_y - ll_y = ur_y - lr_y$ The combination of **Position** and **CropRect** determines the image's scaling, rotation, reflection, and skew.
Resolution	array	%ALDImageResolution	*(Optional)* An array of two numbers of the form [*horizRes vertRes*] specifying the resolution of the image in samples per inch.

KEY	TYPE	OPI COMMENT	VALUE
ColorType	name	%ALDImageColorType	*(Optional)* The type of color specified by the **Color** entry. Valid values are Process, Spot, and Separation. Default value: Spot.
Color	array	%ALDImageColor	*(Optional)* An array of four numbers and a string of the form [*C M Y K colorName*] specifying the value and name of the color in which the image is to be rendered. The values of *C*, *M*, *Y*, and *K* must all be in the range 0.0 to 1.0. Default value: [0.0 0.0 0.0 1.0 (Black)].
Tint	number	%ALDImageTint	*(Optional)* A number in the range 0.0 to 1.0 specifying the concentration of the color specified by **Color** in which the image is to be rendered. Default value: 1.0.
Overprint	boolean	%ALDImageOverprint	*(Optional)* A flag specifying whether the image is to overprint (**true**) or knock out (**false**) underlying marks on other separations. Default value: **false**.
ImageType	array	%ALDImageType	*(Optional)* An array of two integers of the form [*samples bits*] specifying the number of samples per pixel and bits per sample in the image.
GrayMap	array	%ALDImageGrayMap	*(Optional)* An array of 2^n integers in the range 0 to 65,535 (where n is the number of bits per sample) recording changes made to the brightness or contrast of the image.
Transparency	boolean	%ALDImageTransparency	*(Optional)* A flag specifying whether white pixels in the image are to be treated as transparent. Default value: **true**.

KEY	TYPE	OPI COMMENT	VALUE
Tags	array	%ALDImageAsciiTag<*NNN*>	*(Optional)* An array of pairs of the form [*tagNum*$_1$ *tagText*$_1$ … *tagNum*$_n$ *tagText*$_n$] where each *tagNum* is an integer representing a TIFF tag number and each *tagText* is a string representing the corresponding ASCII tag value.

TABLE 10.52 Entries in a version 2.0 OPI dictionary

KEY	TYPE	OPI COMMENT	VALUE
Type	name		*(Optional)* The type of PDF object that this dictionary describes; if present, must be **OPI** for an OPI dictionary.
Version	number		*(Required)* The version of OPI to which this dictionary refers; must be the number 2 or 2.0 (not the name 2.0, as in an OPI version dictionary).
F	file specification	%%ImageFilename	*(Required)* The external file containing the low-resolution proxy image. (See implementation note 152 in Appendix H.)
MainImage	string	%%MainImage	*(Optional)* The pathname of the file containing the full-resolution image corresponding to this proxy, or any other identifying string that uniquely identifies the full-resolution image.
Tags	array	%%TIFFASCIITag	*(Optional)* An array of pairs of the form [*tagNum*$_1$ *tagText*$_1$ … *tagNum*$_n$ *tagText*$_n$] where each *tagNum* is an integer representing a TIFF tag number and each *tagText* is a string or an array of strings representing the corresponding ASCII tag value.
Size	array	%%ImageDimensions	*(Optional; see note below)* An array of two numbers of the form [*width* *height*] specifying the dimensions of the image in pixels.

KEY	TYPE	OPI COMMENT	VALUE
CropRect	rectangle	%%ImageCropRect	*(Optional; see note below)* An array of four numbers of the form [*left top right bottom*] specifying the portion of the image to be used. ***Note:*** *The* ***Size*** *and* ***CropRect*** *entries should either both be present or both absent. If present, they must satisfy the conditions* $0 \le left < right \le width$ *and* $0 \le top < bottom \le height$ *(Note that in this coordinate space, the positive y axis extends vertically downward; hence the requirement that top < bottom.)*
Overprint	boolean	%%ImageOverprint	*(Optional)* A flag specifying whether the image is to overprint (**true**) or knock out (**false**) underlying marks on other separations. Default value: **false**.
Inks	name or array	%%ImageInks	*(Optional)* A name object or array specifying the colorants to be applied to the image. The value may be the name full_color or registration or an array of the form [/monochrome $name_1$ $tint_1$ … $name_n$ $tint_n$] where each *name* is a string representing the name of a colorant and each *tint* is a real number in the range 0.0 to 1.0 specifying the concentration of that colorant to be applied.
IncludedImageDimensions	array	%%IncludedImageDimensions	*(Optional)* An array of two integers of the form [*pixelsWide pixelsHigh*] specifying the dimensions of the included image in pixels.
IncludedImageQuality	number	%%IncludedImageQuality	*(Optional)* A number indicating the quality of the included image. Valid values are 1, 2, and 3.

APPENDIX A

Operator Summary

THIS APPENDIX LISTS all the operators used in PDF content streams, in alphabetical order. Corresponding PostScript language operators are given in Table A.1 only when they are exact or near-exact equivalents of the PDF operators. Table and page references are to the place in the text where each operator is introduced.

TABLE A.1 PDF content stream operators

OPERATOR	POSTSCRIPT EQUIVALENT	DESCRIPTION	TABLE	PAGE
b	**closepath**, **fill**, **stroke**	Close, fill, and stroke path using nonzero winding number rule	4.10	200
B	**fill**, **stroke**	Fill and stroke path using nonzero winding number rule	4.10	200
b*	**closepath**, **eofill**, **stroke**	Close, fill, and stroke path using even-odd rule	4.10	200
B*	**eofill**, **stroke**	Fill and stroke path using even-odd rule	4.10	200
BDC		*(PDF 1.2)* Begin marked-content sequence with property list	10.7	721
BI		Begin inline image object	4.39	316
BMC		*(PDF 1.2)* Begin marked-content sequence	10.7	721
BT		Begin text object	5.4	367
BX		*(PDF 1.1)* Begin compatibility section	3.29	126
c	**curveto**	Append curved segment to path (three control points)	4.9	196
cm	**concat**	Concatenate matrix to current transformation matrix	4.7	189
CS	**setcolorspace**	*(PDF 1.1)* Set color space for stroking operations	4.21	250

OPERATOR	POSTSCRIPT EQUIVALENT	DESCRIPTION	TABLE	PAGE
cs	**setcolorspace**	*(PDF 1.1)* Set color space for nonstroking operations	4.21	250
d	**setdash**	Set line dash pattern	4.7	189
d0	**setcharwidth**	Set glyph width in Type 3 font	5.10	385
d1	**setcachedevice**	Set glyph width and bounding box in Type 3 font	5.10	385
Do		Invoke named XObject	4.34	295
DP		*(PDF 1.2)* Define marked-content point with property list	10.7	721
EI		End inline image object	4.39	316
EMC		*(PDF 1.2)* End marked-content sequence	10.7	721
ET		End text object	5.4	367
EX		*(PDF 1.1)* End compatibility section	3.29	126
f	**fill**	Fill path using nonzero winding number rule	4.10	200
F	**fill**	Fill path using nonzero winding number rule (obsolete)	4.10	200
f*	**eofill**	Fill path using even-odd rule	4.10	200
G	**setgray**	Set gray level for stroking operations	4.21	251
g	**setgray**	Set gray level for nonstroking operations	4.21	251
gs		*(PDF 1.2)* Set parameters from graphics state parameter dictionary	4.7	189
h	**closepath**	Close subpath	4.9	197
i	**setflat**	Set flatness tolerance	4.7	189
ID		Begin inline image data	4.39	316
j	**setlinejoin**	Set line join style	4.7	189
J	**setlinecap**	Set line cap style	4.7	189
K	**setcmykcolor**	Set *CMYK* color for stroking operations	4.21	251
k	**setcmykcolor**	Set *CMYK* color for nonstroking operations	4.21	251

OPERATOR	**POSTSCRIPT EQUIVALENT**	**DESCRIPTION**	**TABLE**	**PAGE**
l	**lineto**	Append straight line segment to path	4.9	196
m	**moveto**	Begin new subpath	4.9	196
M	**setmiterlimit**	Set miter limit	4.7	189
MP		*(PDF 1.2)* Define marked-content point	10.7	721
n		End path without filling or stroking	4.10	200
q	**gsave**	Save graphics state	4.7	189
Q	**grestore**	Restore graphics state	4.7	189
re		Append rectangle to path	4.9	197
RG	**setrgbcolor**	Set *RGB* color for stroking operations	4.21	251
rg	**setrgbcolor**	Set *RGB* color for nonstroking operations	4.21	251
ri		Set color rendering intent	4.7	189
s	**closepath**, **stroke**	Close and stroke path	4.10	200
S	**stroke**	Stroke path	4.10	200
SC	**setcolor**	*(PDF 1.1)* Set color for stroking operations	4.21	250
sc	**setcolor**	*(PDF 1.1)* Set color for nonstroking operations	4.21	251
SCN	**setcolor**	*(PDF 1.2)* Set color for stroking operations (**ICCBased** and special color spaces)	4.21	251
scn	**setcolor**	*(PDF 1.2)* Set color for nonstroking operations (**ICCBased** and special color spaces)	4.21	251
sh	**shfill**	*(PDF 1.3)* Paint area defined by shading pattern	4.24	266
T*		Move to start of next text line	5.5	369
Tc		Set character spacing	5.2	360
Td		Move text position	5.5	368
TD		Move text position and set leading	5.5	368

OPERATOR	**POSTSCRIPT EQUIVALENT**	**DESCRIPTION**	**TABLE**	**PAGE**
Tf	**selectfont**	Set text font and size	5.2	360
Tj	**show**	Show text	5.6	369
TJ		Show text, allowing individual glyph positioning	5.6	370
TL		Set text leading	5.2	360
Tm		Set text matrix and text line matrix	5.5	369
Tr		Set text rendering mode	5.2	360
Ts		Set text rise	5.2	360
Tw		Set word spacing	5.2	360
Tz		Set horizontal text scaling	5.2	360
v	**curveto**	Append curved segment to path (initial point replicated)	4.9	196
w	**setlinewidth**	Set line width	4.7	189
W	**clip**	Set clipping path using nonzero winding number rule	4.11	205
W*	**eoclip**	Set clipping path using even-odd rule	4.11	205
y	**curveto**	Append curved segment to path (final point replicated)	4.9	196
'		Move to next line and show text	5.6	369
"		Set word and character spacing, move to next line, and show text	5.6	370

APPENDIX B

Operators in Type 4 Functions

THIS APPENDIX SUMMARIZES the PostScript operators that can appear in a type 4 function, as discussed in Section 3.9.4, "Type 4 (PostScript Calculator) Functions." For details on these operators, see the *PostScript Language Reference*, Third Edition.

B.1 Arithmetic Operators

num_1 num_2	**add**	*sum*	Return *num_1* plus *num_2*
num_1 num_2	**sub**	*difference*	Return *num_1* minus *num_2*
num_1 num_2	**mul**	*product*	Return *num_1* times *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* as an integer
int_1 int_2	**mod**	*remainder*	Return remainder after dividing *int_1* by *int_2*
num_1	**neg**	*num_2*	Return negative of *num_1*
num_1	**abs**	*num_2*	Return absolute value 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*
angle	**sin**	*real*	Return sine of *angle* degrees
angle	**cos**	*real*	Return cosine of *angle* degrees
num den	**atan**	*angle*	Return arc tangent of *num*/*den* in 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)
num	**cvi**	*int*	Convert to integer
num	**cvr**	*real*	Convert to real

B.2 Relational, Boolean, and Bitwise Operators

any_1 any_2	**eq**	$bool$	Test equal
any_1 any_2	**ne**	$bool$	Test not equal
num_1 num_2	**gt**	$bool$	Test greater than
num_1 num_2	**ge**	$bool$	Test greater than or equal
num_1 num_2	**lt**	$bool$	Test less than
num_1 num_2	**le**	$bool$	Test less than or equal
$bool_1 \vert int_1$ $bool_2 \vert int_2$	**and**	$bool_3 \vert int_3$	Perform logical|bitwise and
$bool_1 \vert int_1$ $bool_2 \vert int_2$	**or**	$bool_3 \vert int_3$	Perform logical|bitwise inclusive or
$bool_1 \vert int_1$ $bool_2 \vert int_2$	**xor**	$bool_3 \vert int_3$	Perform logical|bitwise exclusive or
$bool_1 \vert int_1$	**not**	$bool_2 \vert int_2$	Perform logical|bitwise not
int_1 $shift$	**bitshift**	int_2	Perform bitwise shift of int_1 (positive is left)
–	**true**	*true*	Return boolean value *true*
–	**false**	*false*	Return boolean value *false*

B.3 Conditional Operators

$bool$ $\{expr\}$	**if**	–	Execute *expr* if *bool* is *true*
$bool$ $\{expr_1\}$ $\{expr_2\}$	**ifelse**	–	Execute $expr_1$ if *bool* is *true*, $expr_2$ if *false*

B.4 Stack 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
$any_1 \ldots any_n$ n	**copy**	$any_1 \ldots any_n$ $any_1 \ldots any_n$	Duplicate top n elements
$any_n \ldots any_0$ n	**index**	$any_n \ldots any_0$ any_n	Duplicate arbitrary element
$any_{n-1} \ldots any_0$ n j	**roll**	$any_{(j-1) \bmod n} \ldots any_0$ $any_{n-1} \ldots any_{j \bmod n}$	Roll n elements up j times

APPENDIX C

Implementation Limits

IN GENERAL, PDF does not restrict the size or quantity of things described in the file format, such as numbers, arrays, images, and so on. However, a PDF viewer application running on a particular processor and in a particular operating environment does have such limits. If a viewer application attempts to perform an action that exceeds one of the limits, it will display an error.

PostScript interpreters also have implementation limits, listed in Appendix B of the *PostScript Language Reference*, Third Edition. It is possible to construct a PDF file that does not violate viewer application limits but will not print on a PostScript printer. Keep in mind that these limits vary according to the PostScript language level, interpreter version, and the amount of memory available to the interpreter.

This appendix describes typical limits for Acrobat. These limits fall into two main classes:

- *Architectural limits*. The hardware on which a viewer application executes imposes certain constraints. For example, an integer is usually represented in 32 bits, limiting the range of allowed integers. In addition, the design of the software imposes other constraints, such as a limit to the number of elements in an array or string.
- *Memory limits*. The amount of memory available to a viewer application limits the number of memory-consuming objects that can be held simultaneously.

PDF itself has one architectural limit: Because ten digits are allocated to byte offsets, the size of a file is limited to 10^{10} bytes (approximately 10 gigabytes).

C.1 General Implementation Limits

Table C.1 describes the architectural limits for Acrobat viewer applications running on 32-bit machines. Because Acrobat implementations are subject to these limits, applications producing PDF files are strongly advised to remain within them. Note, however, that memory limits will often be exceeded before architectural limits (such as the limit on the number of indirect objects) are reached.

TABLE C.1 Architectural limits

QUANTITY	LIMIT	DESCRIPTION
integer	2,147,483,647	Largest integer value; equal to $2^{31} - 1$.
	-2,147,483,648	Smallest integer value; equal to -2^{31}.
real	$\pm 3.403 \times 10^{38}$	Largest and smallest real values (approximate).
	$\pm 1.175 \times 10^{-38}$	Nonzero real values closest to 0 (approximate). Values closer than these are automatically converted to 0.
	5	Number of significant decimal digits of precision in fractional part (approximate).
	Note: *To represent real numbers, Acrobat 6 uses IEEE single-precision floating-point numbers, as described in the IEEE Standard for Binary Floating-Point Arithmetic (see the Bibliography). Previous versions used 32-bit fixed-point numbers (16 bits on either side of the radix point), which have greater precision but a much smaller range than IEEE floating-point numbers. (Acrobat 6 still converts floating-point to fixed point for some components, such as screen display and fonts.)*	
string	65,535	Maximum length of a string, in bytes.
name	127	Maximum length of a name, in bytes.
array	8191	Maximum capacity of an array, in elements.
dictionary	4095	Maximum capacity of a dictionary, in entries.
indirect object	8,388,607	Maximum number of indirect objects in a PDF file.
q/**Q** nesting	28	Maximum depth of graphics state nesting by **q** and **Q** operators. (This is not a limit of Acrobat as such, but arises from the fact that **q** and **Q** are implemented by the PostScript **gsave** and **grestore** operators when generating PostScript output; see implementation note 153 in Appendix H.)

QUANTITY	LIMIT	DESCRIPTION
DeviceN components	32	Maximum number of colorants or tint components in a **DeviceN** color space.
CID	65,535	Maximum value of a CID (character identifier).

Acrobat has some additional architectural limits:

- Thumbnail images may be no larger than 106 by 106 samples, and should be created at one-eighth scale for 8.5-by-11-inch and A4-size pages.
- The minimum allowed page size in Acrobat 4.0 is 3 by 3 units in default user space (approximately 0.04 by 0.04 inch); the maximum is 14,400 by 14,400 units (200 by 200 inches). (See implementation note 154 in Appendix H.)
- The magnification factor of a view is constrained to be between approximately 8 percent and 3200 percent. These limits are not fixed; they vary with the size of the page being displayed, as well as with the size of the pages previously viewed within the file.
- When Acrobat reads a PDF file with a damaged or missing cross-reference table, it attempts to rebuild the table by scanning all the objects in the file. However, the generation numbers of deleted entries are lost if the cross-reference table is missing or severely damaged. Reconstruction fails if any object identifiers do not appear at the start of a line or if the **endobj** keyword does not appear at the start of a line. Also, reconstruction fails if a stream contains a line beginning with the word **endstream**, aside from the required **endstream** that delimits the end of the stream.

Memory limits cannot be characterized as precisely as architectural limits can, because the amount of available memory and the ways in which it is allocated vary from one product to another. 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 to another purpose if that memory is currently unused or its use is nonessential (a cache, for example). Also, data is often saved to a temporary file when memory is limited. Because of this behavior, it is not possible to state limits for such items as the number of pages in a document, number of text annotations or hypertext links on a page, number of graphics objects on a page, or number of fonts on a page or in a document.

C.2 Implementation Limits Affecting Web Capture

The data structures constructed by the Web Capture plug-in extension (*PDF 1.3*; see Section 10.9) depend on the maximum length of an array, *k*, which is 8191 elements in the Acrobat 4 implementation.

- A content set array can associate at most *k* content sets with a given name.
- A content set can reference at most *k* objects.
- There can be at most *k* source information dictionaries associated with a single content set.
- A URL alias dictionary can contain at most *k* chains, and each chain can contain at most *k* URLs.
- A maximum of *k* command dictionaries can be stored in the **C** array of the Web Capture information dictionary.
- There can be at most $k \div 2$ entries in the **C** dictionary of a Web Capture command settings dictionary.

APPENDIX D

Character Sets and Encodings

THIS APPENDIX LISTS the character sets and encodings that are assumed to be predefined in any PDF viewer application. Only simple fonts, encompassing Latin text and some symbols, are described here. See "Predefined CMaps" on page 404 for a list of predefined CMaps for CID-keyed fonts.

Section D.1, "Latin Character Set and Encodings," describes the entire character set for Adobe's standard Latin-text fonts. This is the character set supported by the Times, Helvetica, and Courier font families, which are among the standard 14 predefined fonts (see "Standard Type 1 Fonts" on page 378). For each named character, an octal character code is given in four different encodings: **StandardEncoding**, **MacRomanEncoding**, **WinAnsiEncoding**, and **PDFDocEncoding** (see Table D.1). Unencoded characters are indicated by a dash (—).

Section D.2, "Expert Set and MacExpertEncoding," describes the so-called "expert" character set, which contains additional characters useful for sophisticated typography, such as small capitals, ligatures, and fractions. For each named character, an octal character code is given in **MacExpertEncoding**. Note that the built-in encoding in an expert font program is usually different from **MacExpertEncoding**.

Sections D.3, "Symbol Set and Encoding," and D.4, "ZapfDingbats Set and Encoding," describe the character sets and built-in encodings for the Symbol and ZapfDingbats (ITC Zapf Dingbats) font programs, which are among the standard 14 predefined fonts. These fonts have built-in encodings that are unique to each font. (The characters for ZapfDingbats are ordered by code instead of by name, since the names in that font are meaningless.)

TABLE D.1 Latin-text encodings

ENCODING	DESCRIPTION
StandardEncoding	Adobe standard Latin-text encoding. This is the built-in encoding defined in Type 1 Latin-text font programs (but generally not in TrueType font programs). PDF does not have a predefined encoding named **StandardEncoding**. However, it is useful to describe this encoding, since a font's built-in encoding can be used as the base encoding from which differences are specified in an encoding dictionary.
MacRomanEncoding	Mac OS standard encoding for Latin text in Western writing systems. PDF has a predefined encoding named **MacRomanEncoding** that can be used with both Type 1 and TrueType fonts.
WinAnsiEncoding	Windows Code Page 1252, often called the "Windows ANSI" encoding. This is the standard Windows encoding for Latin text in Western writing systems. PDF has a predefined encoding named **WinAnsiEncoding** that can be used with both Type 1 and TrueType fonts.
PDFDocEncoding	Encoding for text strings in a PDF document *outside* the document's content streams. This is one of two encodings (the other being Unicode) that can be used to represent text strings; see Section 3.8.1, "Text Strings." PDF does not have a predefined encoding named **PDFDocEncoding**; it is not customary to use this encoding to show text from fonts.
MacExpertEncoding	An encoding for use with expert fonts—ones containing the expert character set. PDF has a predefined encoding named **MacExpertEncoding**. Despite its name, it is not a platform-specific encoding; however, only certain fonts have the appropriate character set for use with this encoding. No such fonts are among the standard 14 predefined fonts.

D.1 Latin Character Set and Encodings

		CHAR CODE (OCTAL)						CHAR CODE (OCTAL)			
CHAR	**NAME**	**STD**	**MAC**	**WIN**	**PDF**	**CHAR**	**NAME**	**STD**	**MAC**	**WIN**	**PDF**
A	A	101	101	101	101	Œ	OE	352	316	214	226
Æ	AE	341	256	306	306	Ó	Oacute	—	356	323	323
Á	Aacute	—	347	301	301	Ô	Ocircumflex	—	357	324	324
Â	Acircumflex	—	345	302	302	Ö	Odieresis	—	205	326	326
Ä	Adieresis	—	200	304	304	Ò	Ograve	—	361	322	322
À	Agrave	—	313	300	300	Ø	Oslash	351	257	330	330
Å	Aring	—	201	305	305	Õ	Otilde	—	315	325	325
Ã	Atilde	—	314	303	303	P	P	120	120	120	120
B	B	102	102	102	102	Q	Q	121	121	121	121
C	C	103	103	103	103	R	R	122	122	122	122
Ç	Ccedilla	—	202	307	307	S	S	123	123	123	123
D	D	104	104	104	104	Š	Scaron	—	—	212	227
E	E	105	105	105	105	T	T	124	124	124	124
É	Eacute	—	203	311	311	Þ	Thorn	—	—	336	336
Ê	Ecircumflex	—	346	312	312	U	U	125	125	125	125
Ë	Edieresis	—	350	313	313	Ú	Uacute	—	362	332	332
È	Egrave	—	351	310	310	Û	Ucircumflex	—	363	333	333
Đ	Eth	—	—	320	320	Ü	Udieresis	—	206	334	334
€	Euro[1]	—	—	200	240	Ù	Ugrave	—	364	331	331
F	F	106	106	106	106	V	V	126	126	126	126
G	G	107	107	107	107	W	W	127	127	127	127
H	H	110	110	110	110	X	X	130	130	130	130
I	I	111	111	111	111	Y	Y	131	131	131	131
Í	Iacute	—	352	315	315	Ý	Yacute	—	—	335	335
Î	Icircumflex	—	353	316	316	Ÿ	Ydieresis	—	331	237	230
Ï	Idieresis	—	354	317	317	Z	Z	132	132	132	132
Ì	Igrave	—	355	314	314	Ž	Zcaron[2]	—	—	216	231
J	J	112	112	112	112	a	a	141	141	141	141
K	K	113	113	113	113	á	aacute	—	207	341	341
L	L	114	114	114	114	â	acircumflex	—	211	342	342
Ł	Lslash	350	—	—	225	´	acute	302	253	264	264
M	M	115	115	115	115	ä	adieresis	—	212	344	344
N	N	116	116	116	116	æ	ae	361	276	346	346
Ñ	Ntilde	—	204	321	321	à	agrave	—	210	340	340
O	O	117	117	117	117	&	ampersand	046	046	046	046

å	aring	—	214	345	345	ê	ecircumflex	—	220	352	352
^	asciicircum	136	136	136	136	ë	edieresis	—	221	353	353
~	asciitilde	176	176	176	176	è	egrave	—	217	350	350
*	asterisk	052	052	052	052	8	eight	070	070	070	070
@	at	100	100	100	100	…	ellipsis	274	311	205	203
ã	atilde	—	213	343	343	—	emdash	320	321	227	204
b	b	142	142	142	142	–	endash	261	320	226	205
\	backslash	134	134	134	134	=	equal	075	075	075	075
|	bar	174	174	174	174	ð	eth	—	—	360	360
{	braceleft	173	173	173	173	!	exclam	041	041	041	041
}	braceright	175	175	175	175	¡	exclamdown	241	301	241	241
[	bracketleft	133	133	133	133	f	f	146	146	146	146
]	bracketright	135	135	135	135	ﬁ	fi	256	336	—	223
˘	breve	306	371	—	030	5	five	065	065	065	065
¦	brokenbar	—	—	246	246	ﬂ	fl	257	337	—	224
•	bullet[3]	267	245	225	200	ƒ	florin	246	304	203	206
c	c	143	143	143	143	4	four	064	064	064	064
ˇ	caron	317	377	—	031	⁄	fraction	244	332	—	207
ç	ccedilla	—	215	347	347	g	g	147	147	147	147
¸	cedilla	313	374	270	270	ß	germandbls	373	247	337	337
¢	cent	242	242	242	242	`	grave	301	140	140	140
ˆ	circumflex	303	366	210	032	>	greater	076	076	076	076
:	colon	072	072	072	072	«	guillemotleft[4]	253	307	253	253
,	comma	054	054	054	054	»	guillemotright[4]	273	310	273	273
©	copyright	—	251	251	251	‹	guilsinglleft	254	334	213	210
¤	currency[1]	250	333	244	244	›	guilsinglright	255	335	233	211
d	d	144	144	144	144	h	h	150	150	150	150
†	dagger	262	240	206	201	˝	hungarumlaut	315	375	—	034
‡	daggerdbl	263	340	207	202	-	hyphen[5]	055	055	055	055
°	degree	—	241	260	260	i	i	151	151	151	151
¨	dieresis	310	254	250	250	í	iacute	—	222	355	355
÷	divide	—	326	367	367	î	icircumflex	—	224	356	356
$	dollar	044	044	044	044	ï	idieresis	—	225	357	357
˙	dotaccent	307	372	—	033	ì	igrave	—	223	354	354
ı	dotlessi	365	365	—	232	j	j	152	152	152	152
e	e	145	145	145	145	k	k	153	153	153	153
é	eacute	—	216	351	351	l	l	154	154	154	154

<	less	074	074	074	074	q	q	161	161	161	161
¬	logicalnot	—	302	254	254	?	question	077	077	077	077
ł	lslash	370	—	—	233	¿	questiondown	277	300	277	277
m	m	155	155	155	155	"	quotedbl	042	042	042	042
¯	macron	305	370	257	257	„	quotedblbase	271	343	204	214
−	minus	—	—	—	212	“	quotedblleft	252	322	223	215
µ	mu	—	265	265	265	”	quotedblright	272	323	224	216
×	multiply	—	—	327	327	‘	quoteleft	140	324	221	217
n	n	156	156	156	156	’	quoteright	047	325	222	220
9	nine	071	071	071	071	‚	quotesinglbase	270	342	202	221
ñ	ntilde	—	226	361	361	'	quotesingle	251	047	047	047
#	numbersign	043	043	043	043	r	r	162	162	162	162
o	o	157	157	157	157	®	registered	—	250	256	256
ó	oacute	—	227	363	363	˚	ring	312	373	—	036
ô	ocircumflex	—	231	364	364	s	s	163	163	163	163
ö	odieresis	—	232	366	366	š	scaron	—	—	232	235
œ	oe	372	317	234	234	§	section	247	244	247	247
˛	ogonek	316	376	—	035	;	semicolon	073	073	073	073
ò	ograve	—	230	362	362	7	seven	067	067	067	067
1	one	061	061	061	061	6	six	066	066	066	066
½	onehalf	—	—	275	275	/	slash	057	057	057	057
¼	onequarter	—	—	274	274		space[6]	040	040	040	040
¹	onesuperior	—	—	271	271	£	sterling	243	243	243	243
ª	ordfeminine	343	273	252	252	t	t	164	164	164	164
º	ordmasculine	353	274	272	272	þ	thorn	—	—	376	376
ø	oslash	371	277	370	370	3	three	063	063	063	063
õ	otilde	—	233	365	365	¾	threequarters	—	—	276	276
p	p	160	160	160	160	³	threesuperior	—	—	263	263
¶	paragraph	266	246	266	266	˜	tilde	304	367	230	037
(	parenleft	050	050	050	050	™	trademark	—	252	231	222
)	parenright	051	051	051	051	2	two	062	062	062	062
%	percent	045	045	045	045	²	twosuperior	—	—	262	262
.	period	056	056	056	056	u	u	165	165	165	165
·	periodcentered	264	341	267	267	ú	uacute	—	234	372	372
‰	perthousand	275	344	211	213	û	ucircumflex	—	236	373	373
+	plus	053	053	053	053	ü	udieresis	—	237	374	374
±	plusminus	—	261	261	261	ù	ugrave	—	235	371	371

		CHAR CODE (OCTAL)						CHAR CODE (OCTAL)			
CHAR	NAME	STD	MAC	WIN	PDF	CHAR	NAME	STD	MAC	WIN	PDF
_	underscore	137	137	137	137	ÿ	ydieresis	—	330	377	377
v	v	166	166	166	166	¥	yen	245	264	245	245
w	w	167	167	167	167	z	z	172	172	172	172
x	x	170	170	170	170	ž	zcaron[2]	—	—	236	236
y	y	171	171	171	171	0	zero	060	060	060	060
ý	yacute	—	—	375	375						

1. In PDF 1.3, the euro character was added to the Adobe standard Latin character set. It is encoded as 200 in **WinAnsiEncoding** and 240 in **PDFDocEncoding**, assigning codes that were previously unused. Apple changed the Mac OS Latin-text encoding for code 333 from the currency character to the euro character. However, this incompatible change has *not* been reflected in PDF's **MacRomanEncoding**, which continues to map code 333 to currency. If the euro character is desired, an encoding dictionary can be used to specify this single difference from **MacRomanEncoding**.
2. In PDF 1.3, the existing Zcaron and zcaron characters were added to **WinAnsiEncoding** as the previously unused codes 216 and 236.
3. In **WinAnsiEncoding**, all unused codes greater than 40 map to the bullet character. However, only code 225 is specifically assigned to the bullet character; other codes are subject to future reassignment.
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. The hyphen character is also encoded as 255 in **WinAnsiEncoding**. The meaning of this duplicate code is "soft hyphen," but it is typographically the same as hyphen.
6. The space character is also encoded as 312 in **MacRomanEncoding** and as 240 in **WinAnsiEncoding**. The meaning of this duplicate code is "nonbreaking space," but it is typographically the same as space.

D.2 Expert Set and MacExpertEncoding

CHAR	NAME	CODE	CHAR	NAME	CODE
Æ	AEsmall	276	J	Jsmall	152
Á	Aacutesmall	207	K	Ksmall	153
Â	Acircumflexsmall	211	Ł	Lslashsmall	302
´	Acutesmall	047	L	Lsmall	154
Ä	Adieresissmall	212	¯	Macronsmall	364
À	Agravesmall	210	M	Msmall	155
Å	Aringsmall	214	N	Nsmall	156
A	Asmall	141	Ñ	Ntildesmall	226
Ã	Atildesmall	213	Œ	OEsmall	317
˘	Brevesmall	363	Ó	Oacutesmall	227
B	Bsmall	142	Ô	Ocircumflexsmall	231
ˇ	Caronsmall	256	Ö	Odieresissmall	232
Ç	Ccedillasmall	215	˛	Ogoneksmall	362
¸	Cedillasmall	311	Ò	Ogravesmall	230
ˆ	Circumflexsmall	136	Ø	Oslashsmall	277
C	Csmall	143	O	Osmall	157
¨	Dieresissmall	254	Õ	Otildesmall	233
˙	Dotaccentsmall	372	P	Psmall	160
D	Dsmall	144	Q	Qsmall	161
É	Eacutesmall	216	˚	Ringsmall	373
Ê	Ecircumflexsmall	220	R	Rsmall	162
Ë	Edieresissmall	221	Š	Scaronsmall	247
È	Egravesmall	217	S	Ssmall	163
E	Esmall	145	Þ	Thornsmall	271
Ð	Ethsmall	104	˜	Tildesmall	176
F	Fsmall	146	T	Tsmall	164
`	Gravesmall	140	Ú	Uacutesmall	234
G	Gsmall	147	Û	Ucircumflexsmall	236
H	Hsmall	150	Ü	Udieresissmall	237
˝	Hungarumlautsmall	042	Ù	Ugravesmall	235
Í	Iacutesmall	222	U	Usmall	165
Î	Icircumflexsmall	224	V	Vsmall	166
Ï	Idieresissmall	225	W	Wsmall	167
Ì	Igravesmall	223	X	Xsmall	170
I	Ismall	151	Ý	Yacutesmall	264

Ÿ	Ydieresissmall	330	4	fouroldstyle	064
Y	Ysmall	171	4	foursuperior	335
Ž	Zcaronsmall	275	⁄	fraction	057
Z	Zsmall	172	-	hyphen	055
&	ampersandsmall	046	-	hypheninferior	137
a	asuperior	201	-	hyphensuperior	321
b	bsuperior	365	i	isuperior	351
¢	centinferior	251	l	lsuperior	361
¢	centoldstyle	043	m	msuperior	367
¢	centsuperior	202	9	nineinferior	273
:	colon	072	9	nineoldstyle	071
₡	colonmonetary	173	9	ninesuperior	341
,	comma	054	n	nsuperior	366
,	commainferior	262	.	onedotenleader	053
,	commasuperior	370	⅛	oneeighth	112
$	dollarinferior	266	1	onefitted	174
$	dollaroldstyle	044	½	onehalf	110
$	dollarsuperior	045	1	oneinferior	301
d	dsuperior	353	1	oneoldstyle	061
8	eightinferior	245	¼	onequarter	107
8	eightoldstyle	070	1	onesuperior	332
8	eightsuperior	241	⅓	onethird	116
e	esuperior	344	o	osuperior	257
¡	exclamdownsmall	326	(	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	263
‒	figuredash	320	·	periodsuperior	371
⅝	fiveeighths	114	¿	questiondownsmall	300
5	fiveinferior	260	?	questionsmall	077
5	fiveoldstyle	065	r	rsuperior	345
5	fivesuperior	336	Rp	rupiah	175
fl	fl	130	;	semicolon	073
4	fourinferior	242	⅞	seveneighths	115

CHAR	NAME	CODE	CHAR	NAME	CODE
$_7$	seveninferior	246	—	threequartersemdash	075
7	sevenoldstyle	067	^{3}	threesuperior	334
^{7}	sevensuperior	340	^{t}	tsuperior	346
$_6$	sixinferior	244	‥	twodotenleader	052
6	sixoldstyle	066	$_2$	twoinferior	252
^{6}	sixsuperior	337	2	twooldstyle	062
	space	040	^{2}	twosuperior	333
^{s}	ssuperior	352	⅔	twothirds	117
⅜	threeeighths	113	$_0$	zeroinferior	274
$_3$	threeinferior	243	0	zerooldstyle	060
3	threeoldstyle	063	^{0}	zerosuperior	342
¾	threequarters	111			

D.3 Symbol Set and Encoding

CHAR	NAME	CODE	CHAR	NAME	CODE
Α	Alpha	101	↔	arrowboth	253
Β	Beta	102	⇔	arrowdblboth	333
Χ	Chi	103	⇓	arrowdbldown	337
Δ	Delta	104	⇐	arrowdblleft	334
Ε	Epsilon	105	⇒	arrowdblright	336
Η	Eta	110	⇑	arrowdblup	335
€	Euro	240	↓	arrowdown	257
Γ	Gamma	107	⎯	arrowhorizex	276
ℑ	Ifraktur	301	←	arrowleft	254
Ι	Iota	111	→	arrowright	256
Κ	Kappa	113	↑	arrowup	255
Λ	Lambda	114	⏐	arrowvertex	275
Μ	Mu	115	∗	asteriskmath	052
Ν	Nu	116	|	bar	174
Ω	Omega	127	β	beta	142
Ο	Omicron	117	{	braceleft	173
Φ	Phi	106	}	braceright	175
Π	Pi	120	⎧	bracelefttp	354
Ψ	Psi	131	⎨	braceleftmid	355
ℜ	Rfraktur	302	⎩	braceleftbt	356
Ρ	Rho	122	⎫	bracerighttp	374
Σ	Sigma	123	⎬	bracerightmid	375
Τ	Tau	124	⎭	bracerightbt	376
Θ	Theta	121	⎪	braceex	357
Υ	Upsilon	125	[	bracketleft	133
ϒ	Upsilon1	241	]	bracketright	135
Ξ	Xi	130	⎡	bracketlefttp	351
Ζ	Zeta	132	⎢	bracketleftex	352
ℵ	aleph	300	⎣	bracketleftbt	353
α	alpha	141	⎤	bracketrighttp	371
&	ampersand	046	⎥	bracketrightex	372
∠	angle	320	⎦	bracketrightbt	373
〈	angleleft	341	•	bullet	267
〉	angleright	361	↵	carriagereturn	277
≈	approxequal	273	χ	chi	143

CHAR	NAME	CODE	CHAR	NAME	CODE
⊗	circlemultiply	304	⌡	integralbt	365
⊕	circleplus	305	∩	intersection	307
♣	club	247	ι	iota	151
:	colon	072	κ	kappa	153
,	comma	054	λ	lambda	154
≅	congruent	100	<	less	074
©	copyrightsans	343	≤	lessequal	243
©	copyrightserif	323	∧	logicaland	331
°	degree	260	¬	logicalnot	330
δ	delta	144	∨	logicalor	332
♦	diamond	250	◊	lozenge	340
÷	divide	270	−	minus	055
·	dotmath	327	′	minute	242
8	eight	070	μ	mu	155
∈	element	316	×	multiply	264
…	ellipsis	274	9	nine	071
∅	emptyset	306	∉	notelement	317
ε	epsilon	145	≠	notequal	271
=	equal	075	⊄	notsubset	313
≡	equivalence	272	ν	nu	156
η	eta	150	#	numbersign	043
!	exclam	041	ω	omega	167
∃	existential	044	ϖ	omega1	166
5	five	065	ο	omicron	157
ƒ	florin	246	1	one	061
4	four	064	(	parenleft	050
⁄	fraction	244	)	parenright	051
γ	gamma	147	⎛	parenlefttp	346
∇	gradient	321	⎜	parenleftex	347
>	greater	076	⎝	parenleftbt	350
≥	greaterequal	263	⎞	parenrighttp	366
♥	heart	251	⎟	parenrightex	367
∞	infinity	245	⎠	parenrightbt	370
∫	integral	362	∂	partialdiff	266
⌠	integraltp	363	%	percent	045
⎮	integralex	364	.	period	056

CHAR	NAME	CODE	CHAR	NAME	CODE
⊥	perpendicular	136	~	similar	176
ϕ	phi	146	6	six	066
φ	phi1	152	/	slash	057
π	pi	160		space	040
+	plus	053	♠	spade	252
±	plusminus	261	∍	suchthat	047
∏	product	325	∑	summation	345
⊂	propersubset	314	τ	tau	164
⊃	propersuperset	311	∴	therefore	134
∝	proportional	265	θ	theta	161
ψ	psi	171	ϑ	theta1	112
?	question	077	3	three	063
√	radical	326	™	trademarksans	344
‾	radicalex	140	™	trademarkserif	324
⊆	reflexsubset	315	2	two	062
⊇	reflexsuperset	312	_	underscore	137
®	registersans	342	∪	union	310
®	registerserif	322	∀	universal	042
ρ	rho	162	υ	upsilon	165
″	second	262	℘	weierstrass	303
;	semicolon	073	ξ	xi	170
7	seven	067	0	zero	060
σ	sigma	163	ζ	zeta	172
ς	sigma1	126			

D.4 ZapfDingbats Set and Encoding

CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE
	space	040	✣	a30	103	❆	a65	146	♠	a109	253
✁	a1	041	✤	a31	104	❇	a66	147	①	a120	254
✂	a2	042	✥	a32	105	❈	a67	150	②	a121	255
✃	a202	043	✦	a33	106	❉	a68	151	③	a122	256
✄	a3	044	✧	a34	107	❊	a69	152	④	a123	257
☎	a4	045	★	a35	110	❋	a70	153	⑤	a124	260
✆	a5	046	✩	a36	111	●	a71	154	⑥	a125	261
✇	a119	047	✪	a37	112	❍	a72	155	⑦	a126	262
✈	a118	050	✫	a38	113	■	a73	156	⑧	a127	263
✉	a117	051	✬	a39	114	❏	a74	157	⑨	a128	264
☛	a11	052	✭	a40	115	❐	a203	160	⑩	a129	265
☞	a12	053	✮	a41	116	❑	a75	161	❶	a130	266
✌	a13	054	✯	a42	117	❒	a204	162	❷	a131	267
✍	a14	055	✰	a43	120	▲	a76	163	❸	a132	270
✎	a15	056	✱	a44	121	▼	a77	164	❹	a133	271
✏	a16	057	✲	a45	122	◆	a78	165	❺	a134	272
✐	a105	060	✳	a46	123	❖	a79	166	❻	a135	273
✑	a17	061	✴	a47	124	◗	a81	167	❼	a136	274
✒	a18	062	✵	a48	125	❘	a82	170	❽	a137	275
✓	a19	063	✶	a49	126	❙	a83	171	❾	a138	276
✔	a20	064	✷	a50	127	❚	a84	172	❿	a139	277
✕	a21	065	✸	a51	130	❛	a97	173	➀	a140	300
✖	a22	066	✹	a52	131	❜	a98	174	➁	a141	301
✗	a23	067	✺	a53	132	❝	a99	175	➂	a142	302
✘	a24	070	✻	a54	133	❞	a100	176	➃	a143	303
✙	a25	071	✼	a55	134	❡	a101	241	➄	a144	304
✚	a26	072	✽	a56	135	❢	a102	242	➅	a145	305
✛	a27	073	✾	a57	136	❣	a103	243	➆	a146	306
✜	a28	074	✿	a58	137	❤	a104	244	➇	a147	307
✝	a6	075	❀	a59	140	❥	a106	245	➈	a148	310
✞	a7	076	❁	a60	141	❦	a107	246	➉	a149	311
✟	a8	077	❂	a61	142	❧	a108	247	➊	a150	312
✠	a9	100	❃	a62	143	♣	a112	250	➋	a151	313
✡	a10	101	❄	a63	144	♦	a111	251	➌	a152	314
✢	a29	102	❅	a64	145	♥	a110	252	➍	a153	315

CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE	CHAR	NAME	CODE
➎	a154	316	➚	a192	332	➦	a176	346	➳	a184	363
➏	a155	317	➛	a166	333	➧	a177	347	➴	a197	364
➐	a156	320	➜	a167	334	➨	a178	350	➵	a185	365
➑	a157	321	➝	a168	335	➩	a179	351	➶	a194	366
➒	a158	322	➞	a169	336	➪	a193	352	➷	a198	367
➓	a159	323	➟	a170	337	➫	a180	353	➸	a186	370
➔	a160	324	➠	a171	340	➬	a199	354	➹	a195	371
→	a161	325	➡	a172	341	➭	a181	355	➺	a187	372
↔	a163	326	➢	a173	342	➮	a200	356	➻	a188	373
↕	a164	327	➣	a162	343	➯	a182	357	➼	a189	374
➘	a196	330	➤	a174	344	➱	a201	361	➽	a190	375
➙	a165	331	➥	a175	345	➲	a183	362	➾	a191	376

APPENDIX E

PDF Name Registry

THIS APPENDIX DISCUSSES a registry, maintained for developers by Adobe Systems, that contains private names and formats used by PDF producers or Acrobat plug-in extensions.

Acrobat enables third parties to add private data to PDF documents and to add plug-in extensions that change viewer behavior based on this data. However, Acrobat users have certain expectations when opening a PDF document, no matter what plug-ins are available. PDF enforces certain restrictions on private data in order to meet these expectations.

A PDF producer or Acrobat viewer plug-in extension may define new types of action, destination, annotation, security, and file system handlers. If a user opens a PDF document and the plug-in that implements the new type of object is unavailable, the viewer will behave as described in Appendix H.

A PDF producer or Acrobat plug-in extension may also add keys to any PDF object that is implemented as a dictionary, except the file trailer dictionary (see Section 3.4.4, "File Trailer"). In addition, a PDF producer or Acrobat plug-in may create tags that indicate the role of marked-content operators *(PDF 1.2)*, as described in Section 10.5, "Marked Content."

To avoid conflicts with third-party names and with future versions of PDF, Adobe maintains a registry for certain private names and formats. Developers must only add private data that conforms to the registry rules. The registry includes three classes:

- *First class*. Names and data formats that are of value to a wide range of developers. All names defined in any version of the PDF specification are first-class names. Plug-in extensions that are publicly available should often use first-class names for their private data. First-class names and data formats must be registered with Adobe and will be made available for all developers to use. To submit a private name and format for consideration as first-class, use the Acrobat SDK feedback form at the following Web page:

<http://partners.adobe.com/asn/developer/feedback.jsp>

- *Second class*. Names that are applicable to a specific developer. (Adobe does not register second-class data formats.) Adobe distributes second-class names by registering developer-specific prefixes, which must be used as the first characters in the names of all private data added by the developer. Adobe will not register the same prefix to two different developers, thereby ensuring that different developers' second-class names will not conflict. It is the responsibility of the developer to ensure that it does not itself use the same name in conflicting ways. To register a developer-specific prefix, use the following Web page:

  <http://partners.adobe.com/asn/developer/pdfregister.jsp>

- *Third class*. Names that can be used only in files that will never be seen by other third parties, because they may conflict with third-class names defined by others. Third-class names all begin with a specific prefix reserved by Adobe for private plug-in extensions. This prefix, which is XX, must be used as the first characters in the names of all private data added by the developer. It is not necessary to contact Adobe to register third-class names.

Note: *New keys for the document information dictionary (see Section 10.2.1, "Document Information Dictionary") or a thread information dictionary (in the* **I** *entry of a thread dictionary; see Section 8.3.2, "Articles") need not be registered.*

APPENDIX F

Linearized PDF

A LINEARIZED PDF FILE is one that has been organized in a special way to enable efficient incremental access in a network environment. The file is valid PDF in all respects, and is compatible with all existing viewers and other PDF applications. Enhanced viewer applications can recognize that a PDF file has been linearized and can take advantage of that organization (as well as added "hint" information) to enhance viewing performance.

The Linearized PDF file organization is an optional feature available beginning in PDF 1.2. Its primary goal is to achieve the following behavior:

- When a document is opened, display the first page as quickly as possible. The first page to be viewed can be an arbitrary page of the document, not necessarily page 0 (though opening at page 0 is most common).
- When the user requests another page of an open document (for example, by going to the next page or by following a link to an arbitrary page), display that page as quickly as possible.
- When data for a page is delivered over a slow channel, display the page incrementally as it arrives. To the extent possible, display the most useful data first.
- Permit user interaction, such as following a link, to be performed even before the entire page has been received and displayed.

This behavior should be achieved for documents of arbitrary size. The total number of pages in the document should have little or no effect on the user-perceived performance of viewing any particular page.

The primary focus of Linearized PDF is optimized viewing of read-only PDF documents. It is intended that the Linearized PDF will be generated once and read many times. Incremental update is still permitted, but the resulting PDF is no longer linearized and subsequently will be treated as ordinary PDF. Linearizing it again may require reprocessing the entire file; see Section F.4.6, "Accessing an Updated File," for details.

Linearized PDF requires two additions to the PDF specification:

- Rules for the ordering of objects in the PDF file
- Additional data structures, called *hint tables*, that enable efficient navigation within the document

Both of these additions are relatively simple to describe; however, using them effectively requires a deeper understanding of their purpose. Consequently, this appendix goes considerably beyond a simple specification of these PDF extensions, to include background, motivation, and strategies.

- Section F.1, "Background and Assumptions," provides background information about the properties of the World Wide Web that are relevant to the design of Linearized PDF.
- Section F.2, "Linearized PDF Document Structure," specifies the file format and object-ordering requirements of Linearized PDF.
- Section F.3, "Hint Tables," specifies the detailed representation of the hint tables.
- Section F.4, "Access Strategies," outlines strategies for accessing Linearized PDF over a network, which in turn determine the optimal way to organize the PDF file itself.

The reader is assumed to be familiar with the basic architecture of the Web, including terms such as URL, HTTP, and MIME.

F.1 Background and Assumptions

The principal problem addressed by the Linearized PDF design is the access of PDF documents through the World Wide Web. This environment has the following important properties:

- The access protocol (HTTP) is a transaction consisting of a request and a response. The client presents a request in the form of a URL, and the server sends a response consisting of one or more MIME-tagged data blocks.

- After a transaction has completed, obtaining more data requires a new request-response transaction. The connection between client and server does not ordinarily persist beyond the end of a transaction, although some implementations may attempt to cache the open connection in order to expedite subsequent transactions with the same server.
- Round-trip delay can be significant. A request-response transaction can take up to several seconds, independent of the amount of data requested.
- The data rate may be limited. A typical bottleneck is a slow modem link between the client and the Internet service provider.

These properties are generally shared by other wide-area network architectures besides the Web. Also, CD-ROMs share some of these properties, since they have relatively slow seek times and limited data rates compared to magnetic media. The remainder of this appendix focuses on the Web.

There are some additional properties of the HTTP protocol that are relevant to the problem of accessing PDF files efficiently. These properties may not all be shared by other protocols or network environments.

- When a PDF file is initially accessed (such as by following a URL hyperlink from some other document), the file type is not known to the client. Therefore, the client initiates a transaction to retrieve the entire document and then inspects the MIME tag of the response as it arrives. Only at that point is the document known to be PDF. Additionally, with a properly configured server environment, the length of the document becomes known at that time.
- The client can abort a response while the transaction is still in progress, if it decides that the remainder of the data is not of immediate interest. In HTTP, aborting the transaction requires closing the connection, which will interfere with the strategy of caching the open connection between transactions.
- The client can request retrieval of portions of a document by specifying one or more byte ranges (by offset and count) in the HTTP request headers. Each range can be relative to either the beginning or the end of the file. The client can specify as many ranges as it wants in the request, and the response will consist of multiple blocks, each properly tagged.
- The client can initiate multiple concurrent transactions in an attempt to obtain multiple responses in parallel. This is commonly done, for instance, to retrieve inline images referenced from an HTML document. This strategy is not

always reliable and may backfire if the transactions interfere with each other by competing for scarce resources in the server or the communication channel.

Note: *Extensive experimentation has determined that having multiple concurrent transactions does not work very well for PDF in some important environments. Therefore, Linearized PDF is designed to enable good performance to be achieved using only one transaction at a time. In particular, this means that the client must have sufficient information to determine the byte ranges for all the objects required to display a given page of the PDF file, so that it can specify all those byte ranges in a single request.*

Finally, the following additional assumptions are made about the PDF viewer application and its local environment:

- The viewer application has plenty of local temporary storage available. It should rarely need to retrieve a given portion of a PDF document more than once from the server.
- The viewer application is able to display PDF data quickly once it has been received. The performance bottleneck is assumed to be in the transport system (throughput or round-trip delay), not in the processing of data after it arrives.

The consequence of these assumptions is that it may be advantageous for the client to do considerable extra work in order to minimize delays due to communications. Such work includes maintaining local caches and reordering actions according to when the needed data becomes available.

F.2 Linearized PDF Document Structure

Except as noted below, all elements of a Linearized PDF file are as specified in Section 3.4, “File Structure,” and all indirect objects in the file are numbered sequentially in two groups, based on their order of appearance in the file.

- The first group consists of the document catalog, certain other document-level objects, and all objects belonging to the first page of the document. These are numbered sequentially starting at the first object number after the last number of the second group. (The stream containing the hint tables, called a *hint* *stream*, may be numbered out of sequence; see Section F.2.5, "Hint Streams (Parts 5 and 10).")

- The second group consists of all remaining objects in the document, including all pages after the first, all shared objects (objects referenced from more than one page, not counting objects referenced from the first page), and so forth. These are numbered sequentially starting at 1.

These groups of objects are indexed by exactly two cross-reference table sections, located as shown in Example F.1. The composition of these groups is discussed in more detail in the sections that follow (ordered by the part number as shown in this example, with one section for parts 5 and 10). All objects have a generation number of 0.

Starting with PDF 1.5, PDF files may contain object streams (see Section 3.4.6, "Object Streams"). In linearized files containing object streams, the following conditions apply:

- Certain additional objects may not be contained in an object stream: the linearization dictionary, the document catalog, and page objects.
- Objects stored within object streams are given the highest range of object numbers within the main and first-page cross-reference sections.
- For files containing object streams, hint data may specify the location and size of the object streams only (or uncompressed objects), not the individual compressed objects. Similarly, shared object references should be made to the object stream containing a compressed object, not to the compressed object itself.
- Cross-reference streams (Section 3.4.7, "Cross-Reference Streams") may be used in place of traditional cross-reference tables. The logic described in this chapter still applies, with the appropriate syntactic changes.

Example F.1

Part 1: Header

```
%PDF−1.1          % …Binary characters…
```

Part 2: Linearization parameter dictionary

```
43 0 obj
   << /Linearized 1.0          % Version
      /L 54567                 % File length
      /H [475 598]             % Primary hint stream offset and length (part 5)
      /O 45                    % Object number of first page's page object (part 6)
      /E 5437                  % Offset of end of first page
      /N 11                    % Number of pages in document
      /T 52786                 % Offset of first entry in main cross-reference table (part 11)
   >>
endobj
```

Part 3: First-page cross-reference table and trailer

```
xref
43 14
0000000052 00000 n
0000000392 00000 n
0000001073 00000 n
…Cross-reference entries for remaining objects in the first page…
0000000475 00000 n
trailer
   << /Size 57                 % Total number of cross-reference table entries in document
      /Prev 52776              % Offset of main cross-reference table (part 11)
      /Root 44 0 R             % Indirect reference to catalog (part 4)
      …Any other entries, such as Info and Encrypt…        % (part 9)
   >>
startxref
0                              % Dummy cross-reference table offset
%%EOF
```

Part 4: Document catalog and other required document-level objects

```
44 0 obj
   << /Type /Catalog
      /Pages 42 0 R
   >>
endobj

…Other objects…
```

Part 5: Primary hint stream (may precede or follow part 6)

```
56 0 obj
    << /Length 457
        … Possibly other stream attributes, such as Filter …
        /S 221                  % Position of shared object hint table
        … Possibly entries for other hint tables …
    >>
stream
    … Page offset hint table …
    … Shared object hint table …
    … Possibly other hint tables …
endstream
endobj
```

Part 6: First-page section (may precede or follow part 5)

```
45 0 obj
    << /Type /Page
        …
    >>
endobj
```

… Outline hierarchy (if the PageMode value in the document catalog is UseOutlines) …

… Objects for first page, including both shared and nonshared objects …

Part 7: Remaining pages

```
1 0 obj
    << /Type /Page
        … Other page attributes, such as MediaBox, Parent, and Contents …
    >>
endobj
```

… Nonshared objects for this page …

… Each successive page followed by its nonshared objects …

… Last page followed by its nonshared objects …

Part 8: Shared objects for all pages except the first

… Shared objects …

Part 9: Objects not associated with pages, if any

… Other objects …

Part 10: Overflow hint stream (optional)

. . . Overflow hint stream . . .

Part 11: Main cross-reference table and trailer

```
xref
0  43
0000000000  65535  f
. . . Cross-reference entries for all except first page's objects . . .
trailer
    << /Size  43 >>              % Trailer need not contain other entries; in particular,
startxref                        %     it should not have a Prev entry
257                              % Offset of first-page cross-reference table (part 3)
%%EOF
```

F.2.1 Header (Part 1)

The Linearized PDF file begins with the standard header line (see Section 3.4.1, "File Header"). Linearization is independent of PDF version number and can be applied to any PDF file of version 1.1 or greater.

The "binary characters" following the percent sign on the second line are characters with codes 128 or greater, as recommended in Section 3.4.1, "File Header."

F.2.2 Linearization Parameter Dictionary (Part 2)

Following the header, the first object in the body of the file (part 2) must be an indirect dictionary object, the *linearization parameter dictionary*, containing the parameters listed in Table F.1. All values in this dictionary must be direct objects. Note that there are no references to this dictionary anywhere in the document. (However, there is a normal entry for it in the first-page cross-reference table, part 3.)

The linearization parameter dictionary must be entirely contained within the first 1024 bytes of the PDF file. This limits the amount of data a viewer application must read before deciding whether the file is linearized.

F.2.3 First-Page Cross-Reference Table and Trailer (Part 3)

Part 3 contains the cross-reference table for all the first page's objects (discussed in Section F.2.6, "First-Page Section (Part 6)") as well as for the document catalog and document-level objects appearing before the first page (discussed in Section F.2.4, "Document Catalog and Document-Level Objects (Part 4)"). Additionally, it contains entries for the linearization parameter dictionary (at the beginning) and the primary hint stream (at the end). This table is a valid cross-reference table as defined in Section 3.4.3, "Cross-Reference Table," although its position in the file is unconventional. It consists of a single cross-reference subsection, with no free entries.

***Note:** In PDF 1.5 and greater, cross-reference streams (see Section 3.4.7, "Cross-Reference Streams") may be used in linearized files in place of traditional cross-reference tables. The logic described in this section still applies, with the appropriate syntactic changes.*

Below the table is the first-page trailer. The **startxref** line at the end of the trailer gives the offset of the first-page cross-reference table. The trailer's **Prev** entry gives the offset of the main cross-reference table near the end of the file. Again, this is valid PDF syntax, although the trailers are linked in an unusual order. A PDF viewer application that is unaware of linearization interprets the first-page cross-reference table as an update to an original document that is indexed by the main cross-reference table.

The first-page trailer must contain valid **Size** and **Root** entries, as well as any other entries needed to display the document. The **Size** value must be the combined number of entries in both the first-page cross-reference table and the main cross-reference table.

The first-page trailer may optionally end with **startxref**, an integer, and %%EOF, just as in an ordinary trailer. This information is ignored.

TABLE F.1 Entries in the linearization parameter dictionary

PARAMETER	TYPE	VALUE
Linearized	number	*(Required)* A version identification for the linearized format. As usual, a change in the integer part indicates an incompatible change in the linearized format, while a change in the fractional part indicates a backward-compatible change. The current version is 1.0.

PARAMETER	TYPE	VALUE
L	integer	*(Required)* The length of the entire file in bytes. This must be exactly equal to the actual length of the PDF file. A mismatch indicates that the file is not linearized and must be treated as ordinary PDF, ignoring linearization information. (If the mismatch resulted from appending an update, the linearization information may still be correct but requires validation; see Section F.4.6, "Accessing an Updated File," for details.)
H	array	*(Required)* An array of either two or four integers, [$offset_1$ $length_1$] or [$offset_1$ $length_1$ $offset_2$ $length_2$]. $offset_1$ is the offset of the primary hint stream from the beginning of the file. (This is the beginning of the stream object, not the beginning of the stream data.) $length_1$ is the length of this stream, including stream object overhead. If the value of the primary hint stream dictionary's **Length** entry is an indirect reference, the object it refers to must immediately follow the stream object, and $length_1$ also includes the length of the indirect length object, including object overhead. (See implementation note 155 in Appendix H.) If there is an overflow hint stream, $offset_2$ and $length_2$ specify its offset and length. (See implementation note 156 in Appendix H.)
O	integer	*(Required)* The object number of the first page's page object.
E	integer	*(Required)* The offset of the end of the first page (the end of part 6 in Example F.1), relative to the beginning of the file. (See implementation note 157 in Appendix H.)
N	integer	*(Required)* The number of pages in the document.
T	integer	*(Required)* In documents that use standard main cross-reference tables (including hybrid-reference files; see *"Compatibility with PDF 1.4" on page 85*), this entry represents the offset of the white-space character preceding the first entry of the main cross-reference table (the entry for object number 0), relative to the beginning of the file. Note that this differs from the **Prev** entry in the first-page trailer, which gives the location of the **xref** line that precedes the table. In PDF 1.5 and later documents that use cross-reference streams exclusively (see Section 3.4.7, "Cross-Reference Streams"), this entry represents the offset of the main cross-reference stream object.
P	integer	*(Optional)* The page number of the first page (see Section F.2.6, "First-Page Section (Part 6)"). Default value: 0.

F.2.4 Document Catalog and Document-Level Objects (Part 4)

Following the first-page cross-reference table and trailer are the catalog dictionary and other objects that are required when the document is opened. These additional objects (constituting part 4) include the values of the following entries, if they are present and are indirect objects:

- The **ViewerPreferences** entry in the catalog.
- The **PageMode** entry in the catalog. (Note that if the value of **PageMode** is UseOutlines, the outline hierarchy is located in part 6; otherwise, the outline hierarchy, if any, is located in part 9. See Section F.2.9, "Other Objects (Part 9)" for details.)
- The **Threads** entry in the catalog, along with all thread dictionaries it refers to. This does not include the threads' information dictionaries or the individual bead dictionaries belonging to the threads.
- The **OpenAction** entry in the catalog.
- The **AcroForm** entry in the catalog. Only the top-level interactive form dictionary is needed, not the objects that it refers to.
- The **Encrypt** entry in the first-page trailer dictionary. All values in the encryption dictionary must be located here also.

Objects that are not ordinarily needed when the document is opened should not be located here but instead should be at the end of the file; see Section F.2.9, "Other Objects (Part 9)." This includes objects such as page tree nodes, the document information dictionary, and the definitions for named destinations.

Note that the objects located here are indexed by the first-page cross-reference table, even though they are not logically part of the first page.

F.2.5 Hint Streams (Parts 5 and 10)

The core of the linearization information is stored in data structures known as *hint tables*, whose format is described in Section F.3, "Hint Tables." They provide indexing information that enables the client to construct a single request for all the objects that are needed to display any page of the document or to retrieve certain other information efficiently. The hint tables may contain additional information to optimize access by plug-in extensions to application-specific data.

The hint tables are not logically part of the information content of the document; they can be derived from the document. Any action that changes the document—for instance, appending an incremental update—will invalidate the hint tables. The document will remain a valid PDF file but will no longer be linearized; see Section F.4.6, "Accessing an Updated File," for details.

The hint tables are binary data structures that are enclosed in a stream object. Syntactically, this stream is a normal PDF indirect object. However, there are no references to the stream anywhere in the document, so it is not logically part of the document; an operation that regenerates the document may remove the stream.

Usually, all the hint tables are contained in a single stream, known as the *primary hint stream*. Optionally, there may be an additional stream containing more hints, known as the *overflow hint stream*. The contents of the two hint streams are to be concatenated and treated as if they were a single unbroken stream.

The primary hint stream, which is required, is shown as part 5 in Example F.1. The order of this part and the first-page section, shown as part 6, may be reversed; see Section F.4, "Access Strategies," for considerations on the choice of placement. The overflow hint stream, part 10, is optional. (See implementation note 156 in Appendix H.)

The location and length of the primary hint stream, and of the overflow hint stream if present, are given in the linearization parameter dictionary at the beginning of the file.

The hint streams are assigned the last object numbers in the file—that is, after the object number for the last object in the first page. Their cross-reference table entries are at the end of the first-page cross-reference table. This object number assignment is independent of the physical locations of the hint streams in the file. (This convention keeps their object numbers out of the way of the numbering of the linearized objects.)

With one exception, the values of all entries in the hint streams' dictionaries must be direct objects, and may contain no indirect object references. The exception is the stream dictionary's **Length** entry (see the discussion of the **H** entry in Table F.1).

In addition to the standard stream attributes, the dictionary of the primary hint stream contains entries giving the position of the beginning of each hint table in the stream. These positions are given in bytes relative to the beginning of the stream data (after decoding filters, if any, are applied) and with the overflow hint stream concatenated if present. The dictionary of the overflow hint stream should not contain these entries. The keys designating the standard hint tables in the primary hint stream's dictionary are listed in Table F.2; Section F.3, "Hint Tables," documents the format of these hint tables. Additionally, there is a required page offset hint table, which must be the first table in the stream and must start at offset 0.

TABLE F.2 Standard hint tables

KEY	HINT TABLE
S	*(Required)* Shared object hint table (see Section F.3.2, "Shared Object Hint Table")
T	*(Present only if thumbnail images exist)* Thumbnail hint table (see Section F.3.3, "Thumbnail Hint Table")
O	*(Present only if a document outline exists)* Outline hint table (see Section F.3.4, "Generic Hint Tables")
A	*(Present only if article threads exist)* Thread information hint table (see Section F.3.4, "Generic Hint Tables")
E	*(Present only if named destinations exist)* Named destination hint table (see Section F.3.4, "Generic Hint Tables")
V	*(Present only if an interactive form dictionary exists)* Interactive form hint table (see Section F.3.5, "Extended Generic Hint Tables")
I	*(Present only if a document information dictionary exists)* Information dictionary hint table (see Section F.3.4, "Generic Hint Tables")
C	*(Present only if a logical structure hierarchy exists; PDF 1.3)* Logical structure hint table (see Section F.3.5, "Extended Generic Hint Tables")
L	*(PDF 1.3)* Page label hint table (see Section F.3.4, "Generic Hint Tables")
R	*(Present only if a renditions name tree exists; PDF 1.5)* Renditions name tree hint table (see Section F.3.5, "Extended Generic Hint Tables")
B	*(Present only if embedded file streams exist; PDF 1.5)* Embedded file stream hint table (see Section F.3.6, "Embedded File Stream Hint Tables")

New keys may be registered for additional hint tables required for new PDF features or for application-specific data accessed by plug-in extensions. See Appendix E for further information.

F.2.6 First-Page Section (Part 6)

As mentioned earlier, the section containing objects belonging to the first page of the document may either precede or follow the primary hint stream. The starting file offset and length of this section can be determined from the hint tables. In addition, the **E** entry in the linearization parameter dictionary specifies the end of the first page (as an offset relative to the beginning of the file), and the **O** entry gives the object number of the first page's page object.

This part of the file contains all the objects needed to display the first page of the document. Ordinarily, the "first page" is page 0—that is, the leftmost leaf page node in the page tree. However, if the document catalog contains an **OpenAction** entry that specifies opening at some page other than page 0, then that page is the "first page" and should be located here. The page number of the first page is given in the **P** entry of the linearization parameter dictionary. (See also implementation note 158 in Appendix H.)

The objects contained here should include the following:

- The page object for the first page. This must be the first object in this part of the file. Its object number is given in the linearization parameter dictionary. This page object must explicitly specify all required attributes, such as **Resources** and **MediaBox**; the attributes cannot be inherited from ancestor page tree nodes.
- The entire outline hierarchy, if the value of the **PageMode** entry in the catalog is UseOutlines. (If the **PageMode** entry is omitted or has some other value and the document has an outline hierarchy, the outline hierarchy appears in part 9; see Section F.2.9, "Other Objects (Part 9)" for details.)
- All objects that the page object refers to, to an arbitrary depth, except page tree nodes or other page objects. This includes objects referred to by its **Contents**, **Resources**, **Annots**, and **B** entries, but not **Thumb**.

The order of objects referenced from the page object should facilitate early user interaction and incremental display of the page data as it arrives. The following order is recommended:

1. The **Annots** array and all annotation dictionaries, to a depth sufficient to allow those annotations to be activated. Information required to draw the annotation can be deferred until later, since annotations are always drawn on top of (hence after) the contents.
2. The **B** (beads) array and all bead dictionaries, if any, for this page. If any beads exist for this page, the **B** array is required to be present in the page dictionary. Additionally, each bead in the thread (not just the first bead) must contain a **T** entry referring to the associated thread dictionary.
3. The resource dictionary, but not the resource objects contained in the dictionary.
4. Resource objects, other than the types listed below, in the order that they are first referenced (directly or indirectly) from the content stream. If the contents are represented as an array of streams, each resource object should precede the stream in which it is first referenced. Note that **Font**, **FontDescriptor**, and **Encoding** resources should be included here, but not substitutable font files referenced from font descriptors (see the last item below).
5. The page contents (**Contents**). If large, this should be represented as an array of indirect references to content streams, which in turn are interleaved with the resources they require. If small, the entire contents should be a single content stream preceding the resources.
6. Image XObjects, in the order that they are first referenced. Images are assumed to be large and slow to transfer, so the viewer application defers rendering images until all the other contents have been displayed.
7. **FontFile** streams, which contain the actual definitions of embedded fonts. These are assumed to be large and slow to transfer, so the viewer application uses substitute fonts until the real ones have arrived. Only those fonts for which substitution is possible can be deferred in this way. (Currently, this includes any Type 1 or TrueType font that has a font descriptor with the Nonsymbolic flag set, indicating the Adobe standard Latin character set).

See Section F.4, "Access Strategies," for additional discussion about object order and incremental drawing strategies.

F.2.7 Remaining Pages (Part 7)

Part 7 of the Linearized PDF file contains the page objects and nonshared objects for all remaining pages of the file, with the objects for each page grouped together. The pages are contiguous and are ordered by page number. If the first page of the file is not page 0, this section starts with page 0 and skips over the first page when its position in the sequence is reached.

For each page, the objects required to display that page are grouped together, except for resources and other objects that are shared with other pages. Shared objects are located in the shared objects section (part 8). The starting file offset and length of any page can be determined from the hint tables.

The recommended order of objects within a page is essentially the same as in the first page. In particular, the page object must be the first object in each section.

In most cases, unlike for the first page, there will be little benefit from interleaving contents with resources. This is because most resources other than images—fonts in particular—are shared among multiple pages and therefore reside in the shared objects section. Image XObjects usually are not shared, but they should appear at the end of the page's section of the file, since rendering of images is deferred.

F.2.8 Shared Objects (Part 8)

Part 8 of the file contains objects, primarily named resources, that are referenced from more than one page but that are not referenced (directly or indirectly) from the first page. The hint tables contain an index of these objects. For more information on named resources, see Section 3.7.2, "Resource Dictionaries."

The order of these objects is essentially arbitrary. However, wherever a resource consists of a multiple-level structure, all components of the structure should be grouped together. If only the top-level object is referenced from outside the group, the entire group can be described by a single entry in the shared object hint table. This helps to minimize the size of the shared object hint table and the number of individual references from entries in the page offset hint table. (See also implementation note 159 in Appendix H.)

F.2.9 Other Objects (Part 9)

Following the shared objects are any other objects that are part of the document but are not required for displaying pages. These objects are divided into functional categories. Objects within each of these categories should be grouped together; the relative order of the categories is unimportant.

- *The page tree*. This can be located here, since the viewer application never needs to consult it. Note that all **Resources** attributes and other inheritable attributes of the page objects must be pushed down and replicated in each of the leaf page objects (but they may contain indirect references to shared objects).
- *Thumbnail images*. These should simply be ordered by page number. Note that the thumbnail image for page 0 should be first, even if the first page of the document is some page other than 0. Each thumbnail image consists of one or more objects, which may refer to objects in the thumbnail shared objects section (see the next item).
- *Thumbnail shared objects*. These are objects that are shared among some or all thumbnail images and are not referenced from any other objects.
- *The outline hierarchy*, if not located in part 6. The order of objects should be the same as the order in which they are displayed by the viewer application. This is a preorder traversal of the outline tree, skipping over any subtree that is closed (that is, whose parent's **Count** value is negative); following that should be the subtrees that were skipped over, in the order in which they would have appeared if they were all open.
- *Thread information dictionaries*, referenced from the **I** entries of thread dictionaries. Note that the thread dictionaries themselves are located with the document catalog, and the bead dictionaries with the individual pages.
- *Named destinations*. These objects include the value of the **Dests** or **Names** entry in the document catalog and all the destination objects that it refers to. See Section F.4.2, "Opening at an Arbitrary Page."
- *The document information dictionary* and the objects contained within it.
- *The interactive form field hierarchy*. This does not include the top-level interactive form dictionary, which is located with the document catalog.
- *Other entries* in the document catalog that are not referenced from any page.
- *(PDF 1.3) The logical structure hierarchy.*

- *(PDF 1.5) The renditions name tree hierarchy.*
- *(PDF 1.5) Embedded file streams.*

F.2.10 Main Cross-Reference and Trailer (Part 11)

Part 11 is the cross-reference table for all objects in the PDF file except those listed in the first-page cross-reference table (part 3). As indicated earlier, this cross-reference table plays the role of the original cross-reference table for the file (before any updates are appended). It must conform to the following rules:

- It consists of a single cross-reference subsection, beginning at object number 0.
- The first entry (for object number 0) must be a free entry.
- The remaining entries are for in-use objects, which are numbered consecutively starting at 1.

As indicated earlier, the **startxref** line gives the offset of the first-page cross-reference table. The **Prev** entry of the first-page trailer gives the offset of the main cross-reference table. The main trailer has no **Prev** entry, and in fact does not need to contain any entries other than **Size**.

Note: *In PDF 1.5 and greater, cross-reference streams (see Section 3.4.7, "Cross-Reference Streams") may be used in linearized files in place of traditional cross-reference tables. The logic described in this chapter still applies, with the appropriate syntactic changes.*

F.3 Hint Tables

The core of the linearization information is stored in two or more hint tables, as indicated by the attributes of the primary hint stream (see Section F.2.5, "Hint Streams (Parts 5 and 10)"). The format of the standard hint tables is described in this section.

There can be additional hint tables for application-specific data that is accessed by plug-in extensions. A generic format for such hint tables is defined; see Section F.3.4, "Generic Hint Tables." Alternatively, the format of a hint table can be private to the application; see Appendix E for further information.

Each hint table consists of a portion of the stream, beginning at the position in the stream indicated by the corresponding stream attribute. Additionally, there is a required page offset hint table, which must be the first table in the stream and must start at offset 0. (If there is an overflow hint stream, its contents are to be appended seamlessly to the primary hint stream; hint table positions are relative to the beginning of this combined stream.) In general, this byte stream is treated as a bit stream, high-order bit first, which is then subdivided into fields of arbitrary width without regard to byte boundaries. However, each hint table begins at a byte boundary.

The hint tables are designed to encode the required information as compactly as possible. Interpreting the hint tables requires reading them sequentially; they are not designed for random access. The client is expected to read and decode the tables once and retain the information for as long as the document remains open.

A hint table encodes the positions of various objects in the file. The representation is either explicit (an offset from the beginning of the file) or implicit (accumulated lengths of preceding objects). Regardless of the representation, the resulting positions must be interpreted as if the primary hint stream itself were not present. That is, a position greater than the *hint stream offset* must have the *hint stream length* added to it in order to determine the actual offset relative to the beginning of the file. (The hint stream offset and hint stream length are the values $offset_1$ and $length_1$ in the **H** array in the linearization parameter dictionary at the beginning of the file.)

The reason for this rule is that the length of the primary hint stream depends on the information contained within the hint tables, and this is not known until after they have been generated. Any information contained in the hint tables must not depend on knowing the primary hint stream's length in advance.

Note that this rule applies only to offsets given in the hint tables and not to offsets given in the cross-reference tables or linearization parameter dictionary. Also, the offset and length of the overflow hint stream, if present, need not be taken into account, since this object follows all other objects in the file.

***Note:** In linearized files that use object streams (Section 3.4.6, "Object Streams), the position specified in a hint table for a compressed object is to be interpreted as a byte range in which the object can be found, not as a precise offset. Viewer applications should locate the object via a cross-reference stream, as it would if the hint table were not present.*

F.3.1 Page Offset Hint Table

The page offset hint table provides information required for locating each page. Additionally, for each page except the first, it also enumerates all shared objects that the page references, directly or indirectly.

This table begins with a header section, described in Table F.3, followed by one or more per-page entries, described in Table F.4. Note that the items making up each per-page entry are not contiguous; they are broken up with items from entries for other pages. The order of items making up the per-page entries is as follows:

1. Item 1 for all pages, in page order starting with the first page
2. Item 2 for all pages, in page order starting with the first page
3. Item 3 for all pages, in page order starting with the first page
4. Item 4 for all shared objects in the second page, followed by item 4 for all shared objects in the third page, and so on
5. Item 5 for all shared objects in the second page, followed by item 5 for all shared objects in the third page, and so on
6. Item 6 for all pages, in page order starting with the first page
7. Item 7 for all pages, in page order starting with the first page

Note: *All the "bits needed" items in Table F.3, such as item 3, may have values in the range 0 through 32. Although that range requires only 6 bits, 16-bit numbers are used.*

TABLE F.3 Page offset hint table, header section

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The least number of objects in a page (including the page object itself).
2	32	The location of the first page's page object.
3	16	The number of bits needed to represent the difference between the greatest and least number of objects in a page.
4	32	The least length of a page in bytes. This is the least length from the beginning of a page object to the last byte of the last object used by that page.

ITEM	SIZE (BITS)	DESCRIPTION
5	16	The number of bits needed to represent the difference between the greatest and least length of a page, in bytes.
6	32	The least offset of the start of any content stream, relative to the beginning of its page. (See implementation note 160 in Appendix H.)
7	16	The number of bits needed to represent the difference between the greatest and least offset to the start of the content stream. (See implementation note 160 in Appendix H.)
8	32	The least content stream length. (See implementation note 161 in Appendix H.)
9	16	The number of bits needed to represent the difference between the greatest and least content stream length. (See implementation note 161 in Appendix H.)
10	16	The number of bits needed to represent the greatest number of shared object references.
11	16	The number of bits needed to represent the numerically greatest shared object identifier used by the pages (discussed further in Table F.4, item 4).
12	16	The number of bits needed to represent the numerator of the fractional position for each shared object reference. For each shared object referenced from a page, there is an indication of where in the page's content stream the object is first referenced. That position is given as the numerator of a fraction, whose denominator is specified once for the entire document (in the next item in this table). The fraction is explained in more detail in Table F.4, item 5.
13	16	The denominator of the fractional position for each shared object reference.

TABLE F.4 Page offset hint table, per-page entry

ITEM	SIZE (BITS)	DESCRIPTION
1	See Table F.3, item 3	A number that, when added to the least number of objects in a page (Table F.3, item 1), gives the number of objects in the page. The first object of the first page has an object number that is the value of the **O** entry in the linearization parameter dictionary at the beginning of the file. The first object of the second page has an object number of 1. Object numbers for subsequent pages can be determined by accumulating the number of objects in all previous pages.

ITEM	SIZE (BITS)	DESCRIPTION
2	See Table F.3, item 5	A number that, when added to the least page length (Table F.3, item 4), gives the length of the page in bytes. The location of the first object of the first page can be determined from its object number (the **O** entry in the linearization parameter dictionary) and the cross-reference table entry for that object (see Section F.2.3, “First-Page Cross-Reference Table and Trailer (Part 3)”). The locations of subsequent pages can be determined by accumulating the lengths of all previous pages. Note that it is necessary to skip over the primary hint stream, wherever it is located.
3	See Table F.3, item 10	The number of shared objects referenced from the page. Note that this must be 0 for the first page, and that the next two items start with the second page.
4	See Table F.3, item 11	*(One item for each shared object referenced from the page)* A *shared object identifier*—that is, an index into the shared object hint table (described in Section F.3.2, “Shared Object Hint Table”). Note that a single entry in the shared object hint table can designate a group of shared objects, only one of which is referenced from outside the group. That is, shared object identifiers are not directly related to object numbers. This identifier combines with the numerators provided in item 5 to form a *shared object reference.*

ITEM	SIZE (BITS)	DESCRIPTION
5	See Table F.3, item 12	*(One item for each shared object referenced from the page)* The numerator of the fractional position for each shared object reference, in the same order as the preceding item. The fraction indicates where in the page's content stream the shared object is first referenced. This item is interpreted as the numerator of a fraction whose denominator is specified once for the entire document (Table F.3, item 13). If the denominator is d, a numerator ranging from 0 to $d - 1$ indicates the corresponding portion of the page's content stream. For example, if the denominator is 4, a numerator of 0, 1, 2, or 3 indicates that the first reference lies in the first, second, third, or fourth quarter of the content stream, respectively. There are two (or more) other possible values for the numerator, which indicate that the shared object is not referenced from the content stream but is needed by annotations or other objects that are drawn after the contents. The value d indicates that the shared object is needed before image XObjects and other nonshared objects that are at the end of the page. A value of $d + 1$ or greater indicates that the shared object is needed after those objects. This method of dividing the page into fractions is only approximate. Determining the first reference to a shared object entails inspecting the unencoded content stream. The relationship between positions in the unencoded and encoded streams is not necessarily linear.
6	See Table F.3, item 7	A number that, when added to the least offset to the start of the content stream (Table F.3, item 6), gives the offset in bytes of the start of the page's content stream, relative to the beginning of the page. This is the offset of the stream object, not the stream data. (See implementation note 160 in Appendix H.)
7	See Table F.3, item 9	A number that, when added to the least content stream length (Table F.3, item 8), gives the length of the page's content stream in bytes. This includes object overhead preceding and following the stream data. (See implementation note 161 in Appendix H.)

F.3.2 Shared Object Hint Table

The shared object hint table gives information required to locate shared objects (see Section F.2.8, "Shared Objects (Part 8)"). Shared objects can be physically located in either of two places: objects that are referenced from the first page are located with the first-page objects (part 6); all other shared objects are located in the shared objects section (part 8).

A single entry in the shared object hint table can actually describe a group of adjacent objects, under the following condition: Only the first object in the group is referenced from outside the group; the remaining objects in the group are referenced only from other objects in the same group. The objects in a group must have adjacent object numbers.

The page offset hint table, interactive form hint table, and logical structure hint table refer to an entry in the shared object hint table by a simple index that is its sequential position in the table, counting from 0.

The shared object hint table consists of a header section (Table F.5), followed by one or more shared object group entries (Table F.6). There are two sequences of shared object group entries: the ones for objects located in the first page, followed by the ones for objects located in the shared objects section. The entries have the same format in both cases. Note that the items making up each shared object group entry are not contiguous; they are broken up with items from entries for other shared object groups. The order of items in each sequence is as follows:

1. Item 1 for the first group, item 1 for the second group, and so on
2. Item 2 for the first group, item 2 for the second group, and so on
3. Item 3 for the first group, item 3 for the second group, and so on
4. Item 4 for the first group, item 4 for the second group, and so on

All objects associated with the first page (part 6) have entries in the shared object hint table, whether or not they are actually shared. The first entry refers to the beginning of the first page and has an object count and length that span all the initial nonshared objects. The next entry refers to a group of shared objects. Subsequent entries span additional groups of either shared or nonshared objects consecutively, until all shared objects in the first page have been enumerated. (The entries that refer to nonshared objects will never be used.)

TABLE F.5 Shared object hint table, header section

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The object number of the first object in the shared objects section (part 8).

ITEM	SIZE (BITS)	DESCRIPTION
2	32	The location of the first object in the shared objects section.
3	32	The number of shared object entries for the first page (including nonshared objects, as noted above).
4	32	The number of shared object entries for the shared objects section. This includes the number of shared object entries for the first page (that is, the value of item 3).
5	16	The number of bits needed to represent the greatest number of objects in a shared object group. (See also implementation note 162 in Appendix H.)
6	32	The least length of a shared object group in bytes.
7	16	The number of bits needed to represent the difference between the greatest and least length of a shared object group, in bytes.

TABLE F.6 Shared object hint table, shared object group entry

ITEM	SIZE (BITS)	DESCRIPTION
1	See Table F.5, item 7	A number that, when added to the least shared object group length (Table F.5, item 6), gives the length of the object group in bytes. The location of the first object of the first page is given in the page offset hint table, header section (Table F.3, item 4). The locations of subsequent object groups can be determined by accumulating the lengths of all previous object groups until all shared objects in the first page have been enumerated. Following that, the location of the first object in the shared objects section can be obtained from the header section of the shared object hint table (Table F.5, item 2).
2	1	A flag indicating whether the shared object signature (item 3) is present; its value is 1 if the signature is present and 0 if it is absent. (See also implementation note 163 in Appendix H.)
3	128	*(Only if item 2 is 1)* The *shared object signature*, a 16-byte MD5 hash that uniquely identifies the resource that the group of objects represents. This is intended to enable the client to substitute a locally cached copy of the resource instead of reading it from the PDF file. Note that this signature is unrelated to signature fields in interactive forms, as defined in the section "Signature Fields" on page 636.

ITEM	SIZE (BITS)	DESCRIPTION
4	See Table F.5, item 5	A number equal to 1 less than the number of objects in the group. The first object of the first page is the one whose object number is given by the **O** entry in the linearization parameter dictionary at the beginning of the file. Object numbers for subsequent entries can be determined by accumulating the number of objects in all previous entries, until all shared objects in the first page have been enumerated. Following that, the first object in the shared objects section has a number that can be obtained from the header section of the shared object hint table (Table F.5, item 1). (See also implementation note 164 in Appendix H.)

***Note:** In a document consisting of only one page, all of that page's objects are nevertheless treated as if they were shared; the shared object hint table reflects this. (See implementation note 165 in Appendix H.)*

F.3.3 Thumbnail Hint Table

The thumbnail hint table consists of a header section (Table F.7), followed by the thumbnails section, which includes one or more per-page entries (Table F.8), each of which describes the thumbnail image for a single page. The entries are in page number order starting with page 0, even if the document catalog contains an **OpenAction** entry that specifies opening at some page other than page 0. Thumbnail images may exist for some pages and not for others.

TABLE F.7 Thumbnail hint table, header section

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The object number of the first thumbnail image (that is, the thumbnail image that is described by the first entry in the thumbnails section).
2	32	The location of the first thumbnail image.
3	32	The number of pages that have thumbnail images.
4	16	The number of bits needed to represent the greatest number of consecutive pages that do not have a thumbnail image.
5	32	The least length of a thumbnail image in bytes.
6	16	The number of bits needed to represent the difference between the greatest and least length of a thumbnail image.

ITEM	SIZE (BITS)	DESCRIPTION
7	32	The least number of objects in a thumbnail image.
8	16	The number of bits needed to represent the difference between the greatest and least number of objects in a thumbnail image.
9	32	The object number of the first object in the thumbnail shared objects section (a subsection of part 9). These are objects (color spaces, for example) that are referenced from some or all thumbnail objects and are not referenced from any other objects. The thumbnail shared objects are undifferentiated; there is no indication of which shared objects are referenced from any given page's thumbnail image.
10	32	The location of the first object in the thumbnail shared objects section.
11	32	The number of thumbnail shared objects.
12	32	The length of the thumbnail shared objects section in bytes.

TABLE F.8 Thumbnail hint table, per-page entry

ITEM	SIZE (BITS)	DESCRIPTION
1	See Table F.7, item 4	*(Optional)* The number of preceding pages lacking a thumbnail image. This indicates how many pages without a thumbnail image lie between the previous entry's page and this one.
2	See Table F.7, item 8	A number that, when added to the least number of objects in a thumbnail image (Table F.7, item 7), gives the number of objects in this page's thumbnail image.
3	See Table F.7, item 6	A number that, when added to the least length of a thumbnail image (Table F.7, item 5), gives the length of this page's thumbnail image in bytes.

The order of items in Table F.8 is as follows:

1. Item 1 for all pages, in page order starting with the first page
2. Item 2 for all pages, in page order starting with the first page
3. Item 3 for all pages, in page order starting with the first page

F.3.4 Generic Hint Tables

Certain categories of objects are associated with the document as a whole rather than with individual pages (see Section F.2.9, "Other Objects (Part 9)"), and it is sometimes useful to provide hints for accessing those objects efficiently. For each category of hints, there is a separate entry in the primary hint stream giving the starting position of the table within the stream (see Section F.2.5, "Hint Streams (Parts 5 and 10)").

Such hints may be represented by a generic hint table, which describes a single group of objects that are located together in the PDF file. The entries in this table are listed in Table F.9. This representation is used for the following hint tables, if needed:

- Outline hint table
- Thread information hint table
- Named destination hint table
- Information dictionary hint table
- Page label hint table

Generic hint tables may also be useful for application-specific objects accessed by plug-in extensions. It is considerably more convenient for a plug-in to use the generic hint representation than to specify custom hints.

TABLE F.9 Generic hint table

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The object number of the first object in the group.
2	32	The location of the first object in the group.
3	32	The number of objects in the group.
4	32	The length of the object group in bytes.

F.3.5 Extended Generic Hint Tables

An extended generic hint table begins with the same entries as in a generic hint table, followed by three additional entries, as shown in Table F.10. It is used to provide hints for accessing objects that reference shared objects. As of PDF 1.5, the following hint tables, if needed, use the extended generic format:

- Interactive form hint table
- Logical structure hint table
- Renditions name tree hint table

Note: *Embedded file streams should not be referred to by this hint table, even if they are reachable from nodes in the renditions name tree; instead they should use the hint table described in Section F.3.6, "Embedded File Stream Hint Tables."*

TABLE F.10 Extended generic hint table

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The object number of the first object in the group.
2	32	The location of the first object in the group.
3	32	The number of objects in the group.
4	32	The length of the object group in bytes.
5	32	The number of shared object references.
6	16	The number of bits needed to represent the numerically greatest shared object identifier used by the objects in the group.
7…	See Table F.3, item 11	Starting with item 7, each of the remaining items in this table is a shared object identifier—that is, an index into the shared object hint table (described in Section F.3.2, "Shared Object Hint Table").

F.3.6 Embedded File Stream Hint Tables

The embedded file streams hint table allows a viewer application to locate all byte ranges of a PDF file needed to access its embedded file streams. An embedded file stream may be grouped with other objects that are referenced by it; all objects in such a group must have adjacent object numbers. (A group may contain no objects at all, if it contains shared object references.)

This hint table has a header section (see Table F.11) which has general information about the embedded file stream groups. It is followed by the entries in Table F.12. Each of the items in Table F.12 is repeated for each embedded file stream group (the number of groups being represented by item 3 in Table F.11). That is, the order of items in Table F.12 is: item 1 for the first group, item 1 for the second group, and so on; item 2 for the first group, item 2 for the second group, and so on; repeated for the 5 items.

TABLE F.11 Embedded file stream hint table, header section

ITEM	SIZE (BITS)	DESCRIPTION
1	32	The object number of the first object in the first embedded file stream group.
2	32	The location of the first object in the first embedded file stream group.
3	32	The number of embedded file stream groups referenced by this hint table.
4	16	The number of bits needed to represent the highest object number corresponding to an embedded file stream object.
5	16	The number of bits needed to represent the greatest number of objects in an embedded file stream group.
6	16	The number of bits needed to represent the greatest length of an embedded file stream group, in bytes.
7	16	The number of bits needed to represent the greatest number of shared object references present in any embedded file stream group.

TABLE F.12 Embedded file stream hint table, per-embedded file stream group entries

ITEM	SIZE (BITS)	DESCRIPTION
1	See Table F.11, item 4	The object number of the embedded file stream that this entry is associated with.
2	See Table F.11, item 5	The number of objects in this embedded file streams group. This item may be 0, meaning that there are only shared object references. In this case, item 4 for this group must be greater than zero and item 3 must be zero.
3	See Table F.11, item 6	The length of this embedded file stream group, in bytes. This item may be 0, meaning that there are only shared object references. In this case, item 4 for this group must be greater than zero and item 2 must be zero.
4	See Table F.11, item 7	The number of shared objects referenced by this embedded file stream group.

ITEM	SIZE (BITS)	DESCRIPTION
5	See Table F.3, item 11	A bit-packed list of shared object identifiers; that is, indices into the shared object hint table (see Section F.3.2, "Shared Object Hint Table"). Item 4 for this group specifies how many shared object identifiers are associated with the group.

F.4 Access Strategies

This section outlines how the client can take advantage of the structure of a Linearized PDF file in order to retrieve and display it efficiently. This material is not formally a part of the Linearized PDF specification, but it may help explain the rationale for the organization.

F.4.1 Opening at the First Page

As described earlier, when a document is initially accessed, a request is issued to retrieve the entire file, starting at the beginning. Consequently, Linearized PDF is organized so that all the data required to display the first page is at the beginning of the file. This includes all resources that are referenced from the first page, whether or not they are also referenced from other pages.

The first page is usually but not necessarily page 0. If the document catalog contains an **OpenAction** entry that specifies opening at some page other than page 0, that page will be the one physically located at the beginning of the document. Thus, opening a document at the default place (rather than a specific destination) requires simply waiting for the first-page data to arrive; no additional transactions are required.

In an ordinary PDF viewer application, opening a document requires first positioning to the end to obtain the **startxref** line. Since a Linearized PDF file has the first page's cross-reference table at the beginning, reading the **startxref** line is not necessary. All that is required is to verify that the file length given in the linearization parameter dictionary at the beginning of the file matches the actual length of the file, indicating that no updates have been appended to the PDF file.

The primary hint stream is located either before or after the first-page section. This means that it will also be retrieved as part of the initial sequential read of the file. The client is expected to interpret and retain all the information in the hint tables. They are reasonably compact and are not designed to be obtained from the file in random pieces.

The client must now decide whether to continue reading the remainder of the document sequentially or to abort the initial transaction and access subsequent pages using separate transactions requesting byte ranges. This decision is a function of the size of the file, the data rate of the channel, and the overhead cost of a transaction.

F.4.2 Opening at an Arbitrary Page

The viewer application may be requested to open a PDF file at an arbitrary page. The page can be specified in one of three ways:

- By page number (remote go-to action, integer page specifier)
- By named destination (remote go-to action, name or string page specifier)
- By article thread (thread action)

Additionally, an indexed search results in opening a document by page number. Handling this case efficiently is especially important.

As indicated above, when the document is initially opened, it is retrieved sequentially starting at the beginning. As soon as the hint tables have been received, the client has sufficient information to request retrieval of any page of the document given its page number. Therefore, it can abort the initial transaction and issue a new transaction for the target page, as described in Section F.4.3, "Going to Another Page of an Open Document."

The position of the primary hint stream (part 5) with respect to the first-page section (part 6) determines how quickly this can be done. If the primary hint stream precedes the first-page section, the initial transaction can be aborted very quickly; however, this is at the cost of increased delay when opening the document at the first page. On the other hand, if the primary hint stream follows the first-page section, displaying the first page is quicker (since the hint tables are not needed for that), but opening at an arbitrary page is delayed by the time required to receive the first page. The decision whether to favor opening at the first page or opening at an arbitrary page must be made at the time a PDF file is linearized.

If an overflow hint stream exists, obtaining it requires issuing an additional transaction. For this reason, inclusion of an overflow hint stream in Linearized PDF, although permitted, is not recommended. The feature exists to allow the linearizer to write the PDF file with space reserved for a primary hint stream of an estimated size, and then go back and fill in the hint tables. If the estimate is too small, the linearizer can append an overflow stream containing the remaining hint table data. This enables the PDF file to be written in one pass, which may be an advantage if the performance of writing PDF is considered important.

Opening at a named destination requires the viewer application first to read the entire **Dests** or **Names** dictionary, for which a hint is present. Using this information, it is possible to determine the page containing the specific destination identified by the name.

Opening to an article requires the viewer application first to read the entire **Threads** array, which is located with the document catalog at the beginning of the document. Using this information, it is possible to determine the page containing the first bead of any thread. Opening at other than the first bead of a thread requires chaining through all the beads until the desired one is reached; there are no hints to accelerate this.

F.4.3 Going to Another Page of an Open Document

Given a page number and the information in the hint tables, it is now straightforward for the client to construct a single request to retrieve any arbitrary page of the document. The request should include:

- The objects of the page itself, whose byte range can be determined from the entry in the page offset hint table.
- The portion of the main cross-reference table referring to those objects. This can be computed from main cross-reference table location (the **T** entry in the linearization parameter dictionary) and the cumulative object number in the page offset hint table.
- The shared objects referenced from the page, whose byte ranges can be determined from information in the shared object hint table.
- The portion or portions of the main cross-reference table referring to those objects, as described above.

The purpose of the fractions in the page offset hint table is to enable the client to schedule retrieval of the page in a way that allows incremental display of the data as it arrives. It accomplishes this by constructing a request that interleaves pieces of the page contents with the shared resources that the contents refer to. This serves much the same purpose as the physical interleaving that is done for the first page.

F.4.4 Drawing a Page Incrementally

The ordering of objects in pages and the organization of the hint tables are intended to allow progressive update of the display and early opportunities for user interaction when the data is arriving slowly. The viewer application must recognize instances in which the targets of indirect object references have not yet arrived and, where possible, rearrange the order in which it acts on the objects in the page.

The following sequence of actions is recommended:

1. Activate the annotations, but do not draw them yet. Also activate the cursor feedback for any article threads in the page.
2. Begin drawing the contents. Whenever there is a reference to an image XObject that has not yet arrived, skip over it. Whenever there is a reference to a font whose definition is an embedded font file that has not yet arrived, draw the text using a substitute font (if that is possible).
3. Draw the annotations.
4. Draw the images as they arrive, together with anything that overlaps them.
5. Once the embedded font definitions have arrived, redraw the text using the correct fonts, together with anything that overlaps the text.

The last two steps should be done using an off-screen buffer if possible, to avoid objectionable flashing during the redraw process.

On encountering a reference XObject (see Section 4.9.3, "Reference XObjects"), the viewer application may choose to initially display the object itself as a proxy and defer the retrieval and rendering of the imported content. Note that, since all XObjects in a Linearized PDF file follow the content stream of the page on which they appear, their retrieval is already deferred; the use of a reference XObject will result in an additional level of deferral.

F.4.5 Following an Article Thread

As indicated earlier, the bead dictionaries for any article thread that visits a given page are located with that page. This enables the bead rectangles to be activated and proper cursor feedback to be shown.

If the user follows a thread, the viewer application can obtain the object number from the **N** or **P** entry of the bead dictionary. This identifies a target bead, which is located with the page to which it belongs. Given this object number, the viewer application can then go to that page, as discussed in Section F.4.3, "Going to Another Page of an Open Document."

F.4.6 Accessing an Updated File

As stated earlier, if a Linearized PDF file subsequently has an incremental update appended to it, the linearization and hints are no longer valid. Actually, this is not necessarily true, but the viewer application must do some additional work to validate them.

When the viewer application sees that the file is longer than the length given in the linearization parameter dictionary, it must issue an additional transaction to read everything that was appended. It must then analyze the objects in that update to see whether any of them modify objects that are in the first page or that are the targets of hints. If so, it must augment its internal data structures as necessary to take the updates into account.

For a PDF file that has received only a small update, this approach may be worthwhile. Accessing the file this way will be quicker than accessing it without hints or retrieving the entire file before displaying any of it.

APPENDIX G

Example PDF Files

THIS APPENDIX PRESENTS several examples showing the structure of actual PDF files:

- A minimal file that can serve as a starting point for creating other PDF files (and that is the basis of later examples)
- A simple example that shows a text string—the classic "Hello World"—and a simple graphics example that draws lines and shapes
- A fragment of a PDF file that illustrates the structure of the page tree for a large document and, similarly, two fragments that illustrate the structure of an outline hierarchy
- Finally, an example showing the structure of a PDF file as it is updated several times, illustrating multiple body sections, cross-reference sections, and trailers

Note: *The* ***Length*** *values of stream objects in the examples and the byte addresses in cross-reference tables are not necessarily accurate.*

G.1 Minimal PDF File

Example G.1 is a PDF file that does not draw anything; it is almost the minimum acceptable PDF file. It is not strictly the minimum acceptable because it contains an outline dictionary (**Outlines** in the document catalog) with a zero count (in which case this object would normally be omitted); a page content stream (**Contents** in the page object); and a resource dictionary (**Resources** in the page object) containing a **ProcSet** array. These objects were included to make this file useful as a starting point for creating other, more realistic PDF files.

Table G.1 lists the objects present in this example.

TABLE G.1 Objects in minimal example

OBJECT NUMBER	OBJECT TYPE
1	**Catalog** (document catalog)
2	**Outlines** (outline dictionary)
3	**Pages** (page tree node)
4	**Page** (page object)
5	Content stream
6	Procedure set array

Note: *When using Example G.1 as a starting point for creating other files, remember to update the* ***ProcSet*** *array as needed (see Section 10.1, "Procedure Sets"). Also, remember that the cross-reference table entries may need to have a trailing space (see Section 3.4.3, "Cross-Reference Table").*

Example G.1

```
%PDF–1.4
1 0 obj
    << /Type /Catalog
       /Outlines 2 0 R
       /Pages 3 0 R
    >>
endobj

2 0 obj
    << /Type Outlines
       /Count 0
    >>
endobj

3 0 obj
    << /Type /Pages
       /Kids [4 0 R]
       /Count 1
    >>
endobj
```

```
4 0 obj
   << /Type /Page
      /Parent 3 0 R
      /MediaBox [0 0 612 792]
      /Contents 5 0 R
      /Resources << /ProcSet 6 0 R >>
   >>
endobj

5 0 obj
   << /Length 35 >>
stream
…Page-marking operators…
endstream
endobj

6 0 obj
   [/PDF]
endobj

xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n

trailer
   << /Size 7
      /Root 1 0 R
   >>
startxref
408
%%EOF
```

G.2 Simple Text String Example

Example G.2 is the classic "Hello World" example built from the preceding example. It shows a single line of text consisting of the string Hello World, illustrating the use of fonts and several text-related PDF operators. The string is displayed in 24-point Helvetica; because Helvetica is one of the standard 14 fonts, no font descriptor is needed.

Table G.2 lists the objects present in this example.

TABLE G.2 Objects in simple text string example

OBJECT NUMBER	OBJECT TYPE
1	**Catalog** (document catalog)
2	**Outlines** (outline dictionary)
3	**Pages** (page tree node)
4	**Page** (page object)
5	Content stream
6	Procedure set array
7	**Font** (Type 1 font)

Example G.2

```
%PDF–1.4
1  0  obj
    <<  /Type  /Catalog
         /Outlines  2 0 R
         /Pages  3 0 R
    >>
endobj

2  0  obj
    <<  /Type  /Outlines
         /Count  0
    >>
endobj
```

```
3  0  obj
    <<  /Type  /Pages
         /Kids  [4 0 R]
         /Count  1
    >>
endobj

4  0  obj
    <<  /Type  /Page
         /Parent  3 0 R
         /MediaBox  [0  0  612  792]
         /Contents  5 0 R
         /Resources  <<  /ProcSet  6 0 R
                             /Font  <<  /F1  7 0 R  >>
                        >>
    >>
endobj

5  0  obj
    << /Length  73  >>
stream
    BT
        /F1  24  Tf
        100  100  Td
        (Hello World)  Tj
    ET
endstream
endobj

6  0  obj
    [/PDF  /Text]
endobj

7  0  obj
    <<  /Type  /Font
         /Subtype  /Type1
         /Name  /F1
         /BaseFont  /Helvetica
         /Encoding  /MacRomanEncoding
    >>
endobj
```

```
xref
0  8
0000000000  65535  f
0000000009  00000  n
0000000074  00000  n
0000000120  00000  n
0000000179  00000  n
0000000364  00000  n
0000000466  00000  n
0000000496  00000  n

trailer
    <<  /Size  8
         /Root  1 0 R
    >>
startxref
625
%%EOF
```

G.3 Simple Graphics Example

Example G.3 draws a thin black line segment, a thick black dashed line segment, a filled and stroked rectangle, and a filled and stroked cubic Bézier curve. Table G.3 lists the objects present in this example, and Figure G.1 shows the resulting output. (Each shape has a red border, and the rectangle is filled with light blue.)

TABLE G.3 Objects in simple graphics example

OBJECT NUMBER	OBJECT TYPE
1	**Catalog** (document catalog)
2	**Outlines** (outline dictionary)
3	**Pages** (page tree node)
4	**Page** (page object)
5	Content stream
6	Procedure set array

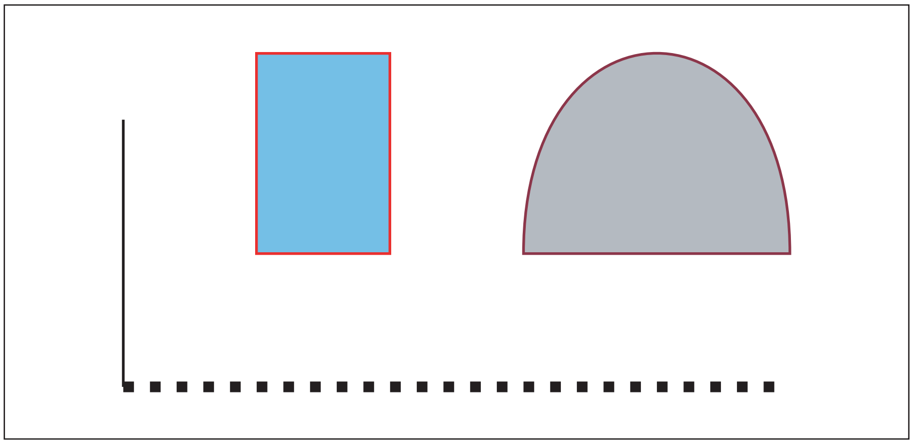


FIGURE G.1 *Output of Example G.3*

Example G.3

```
%PDF–1.4
1 0 obj
    << /Type /Catalog
       /Outlines 2 0 R
       /Pages 3 0 R
    >>
endobj

2 0 obj
    << /Type /Outlines
       /Count 0
    >>
endobj

3 0 obj
    << /Type /Pages
       /Kids [4 0 R]
       /Count 1
    >>
endobj
```

```
4 0 obj
   << /Type /Page
      /Parent 3 0 R
      /MediaBox [0 0 612 792]
      /Contents 5 0 R
      /Resources << /ProcSet 6 0 R >>
   >>
endobj

5 0 obj
   << /Length 883 >>
stream
   % Draw a black line segment, using the default line width.
   150 250 m
   150 350 l
   S

   % Draw a thicker, dashed line segment.
   4 w                                   % Set line width to 4 points
   [4 6] 0 d                             % Set dash pattern to 4 units on, 6 units off
   150 250 m
   400 250 l
   S

   [] 0 d                                % Reset dash pattern to a solid line
   1 w                                   % Reset line width to 1 unit

   % Draw a rectangle with a 1–unit red border, filled with light blue.
   1.0 0.0 0.0 RG                        % Red for stroke color
   0.5 0.75 1.0 rg                       % Light blue for fill color
   200 300 50 75 re
   B

   % Draw a curve filled with gray and with a colored border.
   0.5 0.1 0.2 RG
   0.7 g
   300 300 m
   300 400 400 400 400 300 c
   b
endstream
endobj

6 0 obj
   [/PDF]
endobj
```

```
xref
0  7
0000000000  65535  f
0000000009  00000  n
0000000074  00000  n
0000000120  00000  n
0000000179  00000  n
0000000300  00000  n
0000001532  00000  n

trailer
    <<  /Size  7
         /Root  1 0 R
    >>
startxref
1556
%%EOF
```

G.4 Page Tree Example

Example G.4 is a fragment of a PDF file illustrating the structure of the page tree for a large document. It contains the page tree nodes for a 62-page document; Figure G.2 shows the structure of this page tree. Numbers in the figure are object numbers corresponding to the objects in the example.

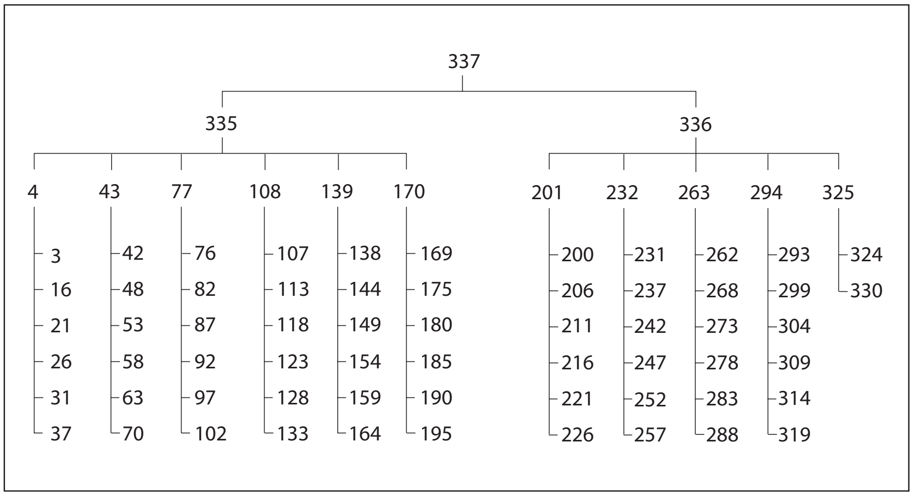


FIGURE G.2 *Page tree for Example G.4*

Example G.4

```
337 0 obj
    << /Type /Pages
       /Kids [ 335 0 R
               336 0 R
             ]
       /Count 62
    >>
endobj

335 0 obj
    << /Type /Pages
       /Parent 337 0 R
       /Kids [ 4 0 R
               43 0 R
               77 0 R
               108 0 R
               139 0 R
               170 0 R
             ]
       /Count 36
    >>
endobj

336 0 obj
    << /Type /Pages
       /Parent 337 0 R
       /Kids [ 201 0 R
               232 0 R
               263 0 R
               294 0 R
               325 0 R
             ]
       /Count 26
    >>
endobj
```

```
4  0  obj
   <<  /Type  /Pages
       /Parent  335 0 R
       /Kids  [  3 0 R
                 16 0 R
                 21 0 R
                 26 0 R
                 31 0 R
                 37 0 R
              ]
       /Count  6
   >>
endobj

43  0  obj
   <<  /Type  /Pages
       /Parent  335 0 R
       /Kids  [  42 0 R
                 48 0 R
                 53 0 R
                 58 0 R
                 63 0 R
                 70 0 R
              ]
       /Count  6
   >>
endobj

77  0  obj
   <<  /Type  /Pages
       /Parent  335 0 R
       /Kids  [  76 0 R
                 82 0 R
                 87 0 R
                 92 0 R
                 97 0 R
                 102 0 R
              ]
       /Count  6
   >>
endobj
```

```
108  0  obj
    <<  /Type  /Pages
        /Parent  335 0 R
        /Kids  [  107 0 R
                  113 0 R
                  118 0 R
                  123 0 R
                  128 0 R
                  133 0 R
               ]
        /Count  6
    >>
endobj

139  0  obj
    <<  /Type  /Pages
        /Parent  335 0 R
        /Kids  [  138 0 R
                  144 0 R
                  149 0 R
                  154 0 R
                  159 0 R
                  164 0 R
               ]
        /Count  6
    >>
endobj

170  0  obj
    <<  /Type  /Pages
        /Parent  335 0 R
        /Kids  [  169 0 R
                  175 0 R
                  180 0 R
                  185 0 R
                  190 0 R
                  195 0 R
               ]
        /Count  6
    >>
endobj
```

```
201 0 obj
    <<  /Type /Pages
        /Parent 336 0 R
        /Kids [ 200 0 R
                206 0 R
                211 0 R
                216 0 R
                221 0 R
                226 0 R
              ]
        /Count 6
    >>
endobj

232 0 obj
    <<  /Type /Pages
        /Parent 336 0 R
        /Kids [ 231 0 R
                237 0 R
                242 0 R
                247 0 R
                252 0 R
                257 0 R
              ]
        /Count 6
    >>
endobj

263 0 obj
    <<  /Type /Pages
        /Parent 336 0 R
        /Kids [ 262 0 R
                268 0 R
                273 0 R
                278 0 R
                283 0 R
                288 0 R
              ]
        /Count 6
    >>
endobj
```

```
294  0  obj
    <<  /Type  /Pages
        /Parent  336 0 R
        /Kids  [   293 0 R
                   299 0 R
                   304 0 R
                   309 0 R
                   314 0 R
                   319 0 R
               ]
        /Count  6
    >>
endobj
```

```
325  0  obj
    <<  /Type  /Pages
        /Parent  336 0 R
        /Kids  [   324 0 R
                   330 0 R
               ]
        /Count  2
    >>
endobj
```

G.5 Outline Hierarchy Example

This section from a PDF file illustrates the structure of an outline hierarchy with six items. Example G.5 shows the outline with all items open, as illustrated in Figure G.3.

On-screen appearance	Object number	Count
	21	6
🗋 Document	22	4
🗋 Section 1	25	0
🗋 Section 2	26	1
🗋 Subsection 1	27	0
🗋 Section 3	28	0
🗋 Summary	29	0

FIGURE G.3 *Document outline as displayed in Example G.5*

Example G.5

```
21  0  obj
    <<  /Type  /Outlines
        /First  22 0 R
        /Last  29 0 R
        /Count  6
    >>
endobj

22  0  obj
    <<  /Title  (Document)
        /Parent  21 0 R
        /Next  29 0 R
        /First  25 0 R
        /Last  28 0 R
        /Count  4
        /Dest  [3 0 R  /XYZ  0  792  0]
    >>
endobj

25  0  obj
    <<  /Title  (Section 1)
        /Parent  22 0 R
        /Next  26 0 R
        /Dest  [3 0 R  /XYZ  null  701  null]
    >>
endobj

26  0  obj
    <<  /Title  (Section 2)
        /Parent  22 0 R
        /Prev  25 0 R
        /Next  28 0 R
        /First  27 0 R
        /Last  27 0 R
        /Count  1
        /Dest  [3 0 R  /XYZ  null  680  null]
    >>
endobj
```

```
27 0 obj
   << /Title (Subsection 1)
      /Parent 26 0 R
      /Dest [3 0 R /XYZ null 670 null]
   >>
endobj

28 0 obj
   << /Title (Section 3)
      /Parent 22 0 R
      /Prev 26 0 R
      /Dest [7 0 R /XYZ null 500 null]
   >>
endobj

29 0 obj
   << /Title (Summary)
      /Parent 21 0 R
      /Prev 22 0 R
      /Dest [8 0 R /XYZ null 199 null]
   >>
endobj
```

Example G.6 is the same as Example G.5, except that one of the outline items has been closed in the display. The outline appears as shown in Figure G.4.

On-screen appearance	Object number	Count
	21	5
🗋 Document	22	3
🗋 Section 1	25	0
🗋 Section 2	26	−1
🗋 Section 3	28	0
🗋 Summary	29	0

FIGURE G.4 *Document outline as displayed in Example G.6*

Example G.6

```
21 0 obj
    << /Type /Outlines
       /First 22 0 R
       /Last 29 0 R
       /Count 5
    >>
endobj

22 0 obj
    << /Title (Document)
       /Parent 21 0 R
       /Next 29 0 R
       /First 25 0 R
       /Last 28 0 R
       /Count 3
       /Dest [3 0 R /XYZ 0 792 0]
    >>
endobj

25 0 obj
    << /Title (Section 1)
       /Parent 22 0 R
       /Next 26 0 R
       /Dest [3 0 R /XYZ null 701 null]
    >>
endobj

26 0 obj
    << /Title (Section 2)
       /Parent 22 0 R
       /Prev 25 0 R
       /Next 28 0 R
       /First 27 0 R
       /Last 27 0 R
       /Count −1
       /Dest [3 0 R /XYZ null 680 null]
    >>
endobj
```

```
27 0 obj
    << /Title (Subsection 1)
         /Parent 26 0 R
         /Dest [3 0 R /XYZ null 670 null]
    >>
endobj

28 0 obj
    << /Title (Section 3)
         /Parent 22 0 R
         /Prev 26 0 R
         /Dest [7 0 R /XYZ null 500 null]
    >>
endobj

29 0 obj
    << /Title (Summary)
         /Parent 21 0 R
         /Prev 22 0 R
         /Dest [8 0 R /XYZ null 199 null]
    >>
endobj
```

G.6 Updating Example

This example shows the structure of a PDF file as it is updated several times; it illustrates multiple body sections, cross-reference sections, and trailers. In addition, it shows that once an object has been assigned an object identifier, it keeps that identifier until the object is deleted, even if the object is altered. Finally, the example illustrates the reuse of cross-reference entries for objects that have been deleted, along with the incrementing of the generation number after an object has been deleted.

The original file is that shown in Example G.1 on page 920. The updates are divided into four stages, with the file saved after each:

1. Four text annotations are added.
2. The text of one of the annotations is altered.
3. Two of the text annotations are deleted.
4. Three text annotations are added.

The sections following show the segments added to the file at each stage. Throughout this example, objects are referred to by their object identifiers, which are made up of the object number and the generation number, rather than simply by their object numbers as in earlier examples. This is necessary because the example reuses object numbers, so the objects they denote are not unique.

Note: *The tables in these sections show only those objects that are modified during the updating process. Objects from Example G.1 that are not altered during the update are not shown.*

G.6.1 Stage 1: Add Four Text Annotations

Four text annotations are added to the initial file and the file is saved. Table G.4 lists the objects involved in this update.

TABLE G.4 Object usage after adding four text annotations

OBJECT IDENTIFIER	OBJECT TYPE
4 0	**Page** (page object)
7 0	Annotation array
8 0	**Annot** (annotation dictionary)
9 0	**Annot** (annotation dictionary)
10 0	**Annot** (annotation dictionary)
11 0	**Annot** (annotation dictionary)

Example G.7 shows the lines added to the file by this update. The page object is updated because an **Annots** entry has been added to it. Note that the file's trailer now contains a **Prev** entry, which points to the original cross-reference section in the file, while the **startxref** value at the end of the trailer points to the cross-reference section added by the update.

Example G.7

```
4  0  obj
   <<  /Type  /Page
       /Parent  3 0 R
       /MediaBox  [0  0  612  792]
       /Contents  5 0 R
       /Resources  <<  /ProcSet  6 0 R  >>
       /Annots  7 0 R
   >>
endobj

7  0  obj
   [  8 0 R
      9 0 R
      10 0 R
      11 0 R
   ]
endobj

8  0  obj
   <<  /Type  /Annot
       /Subtype  /Text
       /Rect  [44  616  162  735]
       /Contents  (Text #1)
       /Open  true
   >>
endobj

9  0  obj
   <<  /Type  /Annot
       /Subtype  /Text
       /Rect  [224  668  457  735]
       /Contents  (Text #2)
       /Open  false
   >>
endobj

10  0  obj
   <<  /Type  /Annot
       /Subtype  /Text
       /Rect  [239  393  328  622]
       /Contents  (Text #3)
       /Open  true
   >>
endobj
```

```
11 0 obj
   << /Type /Annot
      /Subtype /Text
      /Rect [34 398 225 575]
      /Contents (Text #4)
      /Open false
   >>
endobj

xref
0 1
0000000000 65535 f
4 1
0000000632 00000 n
7 5
0000000810 00000 n
0000000883 00000 n
0000001024 00000 n
0000001167 00000 n
0000001309 00000 n

trailer
   << /Size 12
      /Root 1 0 R
      /Prev 408
   >>
startxref
1452
%%EOF
```

G.6.2 Stage 2: Modify Text of One Annotation

One text annotation is modified and the file is saved. Example G.8 shows the lines added to the file by this update. Note that the file now contains two copies of the object with identifier 10 0 (the text annotation that was modified) and that the added cross-reference section points to the more recent version of the object. This added cross-reference section contains one subsection, containing only an entry for the object that was modified. In addition, the **Prev** entry in the file's trailer has been updated to point to the cross-reference section added in the previous stage, while the **startxref** value at the end of the trailer points to the newly added cross-reference section.

Example G.8

```
10 0 obj
    << /Type /Annot
         /Subtype /Text
         /Rect [239 393 328 622]
         /Contents (Modified Text #3)
         /Open true
    >>
endobj

xref
0 1
0000000000 65535 f
10 1
0000001703 00000 n

trailer
    << /Size 12
         /Root 1 0 R
         /Prev 1452
    >>
startxref
1855
%%EOF
```

G.6.3 Stage 3: Delete Two Annotations

Two text annotation are deleted and the file is saved. Table G.5 lists the objects updated.

TABLE G.5 Object usage after deleting two text annotations

OBJECT IDENTIFIER	OBJECT TYPE
7 0	Annotation array
8 0	Free
9 0	Free

The **Annots** array is the only object that is written in this update. It is updated because it now contains two annotations fewer.

Example G.9 shows the lines added when the file was saved. Note that objects with identifiers 8 0 and 9 0 have been deleted, as can be seen from the fact that their entries in the cross-reference section end with the keyword **f**.

Example G.9

```
7 0 obj
   [ 10 0 R
     11 0 R
   ]
endobj

xref
0 1
0000000008 65535 f
7 3
0000001978 00000 n
0000000009 00001 f
0000000000 00001 f

trailer
   <<  /Size 12
       /Root 1 0 R
       /Prev 1855
   >>
startxref
2027
%%EOF
```

The cross-reference section added at this stage contains four entries, representing object number 0, the **Annots** array, and the two deleted text annotations.

- The cross-reference entry for object number 0 is updated because it is the head of the linked list of free entries and must now point to the entry for the newly freed object number 8. The entry for object number 8 points to the entry for object number 9 (the next free entry), while the entry for object number 9 is the last free entry in the cross-reference table, indicated by the fact that it points back to object number 0.
- The entries for the two deleted text annotations are marked as free and as having generation numbers of 1, which will be used for any objects that reuse these cross-reference entries. Keep in mind that, although the two objects have been deleted, they are still present in the file. It is the cross-reference table that records the fact that they have been deleted.

The **Prev** entry in the trailer has again been updated, so that it points to the cross-reference section added at the previous stage, and the **startxref** value points to the newly added cross-reference section.

G.6.4 Stage 4: Add Three Annotations

Finally, three new text annotations are added to the file. Table G.6 lists the objects involved in this update.

TABLE G.6 Object usage after adding three text annotations

OBJECT IDENTIFIER	OBJECT TYPE
7 0	Annotation array
8 1	**Annot** (annotation dictionary)
9 1	**Annot** (annotation dictionary)
12 0	**Annot** (annotation dictionary)

Object numbers 8 and 9, which were used for the two annotations deleted in the previous stage, have been reused; however, the new objects have been given a generation number of 1. In addition, the third text annotation added has been assigned the previously unused object identifier of 12 0.

Example G.10 shows the lines added to the file by this update. The added cross-reference section contains five entries, corresponding to object number 0, the **Annots** array, and the three annotations added. The entry for object number 0 is updated because the previously free entries for object numbers 8 and 9 have been reused. The entry for object number 0 now shows that there are no free entries in the cross-reference table. The **Annots** array is updated to reflect the addition of the three text annotations.

Example G.10

```
7 0 obj
   [ 10 0 R
     11 0 R
     8 1 R
     9 1 R
     12 0 R
   ]
endobj
```

```
8 1 obj
    << /Type /Annot
         /Subtype /Text
         /Rect [58 657 172 742]
         /Contents (New Text #1)
         /Open true
    >>
endobj

9 1 obj
    << /Type /Annot
         /Subtype /Text
         /Rect [389 459 570 537]
         /Contents (New Text #2)
         /Open false
    >>
endobj

12 0 obj
    << /Type /Annot
         /Subtype /Text
         /Rect [44 253 473 337]
         /Contents (New Text #3\203a longer text annotation which we will continue \
onto a second line)
         /Open true
    >>
endobj

xref
0 1
0000000000 65535 f
7 3
0000002216 00000 n
0000002302 00001 n
0000002447 00001 n
12 1
0000002594 00000 n

trailer
    << /Size 13
         /Root 1 0 R
         /Prev 2027
    >>
startxref
2814
%%EOF
```

The annotation with object identifier 12 0 illustrates splitting a long text string across multiple lines, as well as the technique for including nonstandard characters in a string. In this case, the character is an ellipsis (…), which is character code 203 (octal) in **PDFDocEncoding**, the encoding used for text annotations.

As in previous updates, the trailer's **Prev** entry and **startxref** value have been updated.

APPENDIX H

Compatibility and Implementation Notes

THE GOAL OF the Adobe Acrobat family of products is to enable people to exchange and view electronic documents easily and reliably. Ideally, this means that any Acrobat viewer application should be able to display the contents of any PDF file, even if the PDF file was created long before or long after the viewer application itself. Of course, new versions of viewer applications are introduced to provide additional capabilities not present before. Furthermore, beginning with Acrobat 2.0, viewer applications may accept plug-in extensions, making some Acrobat 2.0 (and later) viewers more capable than others, depending on what extensions are present.

Both the viewer applications and PDF itself have been designed to enable users to view everything in the document that the viewer application understands and to ignore or inform the user about objects not understood. The decision whether to ignore or inform the user is made on a feature-by-feature basis.

The original PDF specification did not define how a viewer application should behave when it reads a file that does not conform to the specification. This appendix provides that information. The PDF version associated with a file determines how it should be treated when a viewer application encounters a problem.

In addition, this appendix includes notes on the Adobe Acrobat implementation for details that are not strictly defined by the PDF specifications.

H.1 PDF Version Numbers

PDF version numbers take the form *M.m*, where *M* is the major and *m* the minor version number. Adobe increments the major version number when the PDF specification changes in such a way that existing viewer applications will be unlikely to read a document without serious errors that prevent pages from being viewed. The minor version number is incremented if the changes do not prevent existing viewer applications from continuing to work, such as the addition of new page description operators. The version number does not change at all if PDF changes in a way that existing viewer applications are unlikely to detect. Such changes might include the addition of private data, such as additional entries in the document catalog, that can be gracefully ignored by applications that do not understand it.

The header in the first line of a PDF file specifies a PDF version (see Section 3.4.1, "File Header"). In PDF 1.4, a PDF version can also be specified in the **Version** entry of the document catalog, essentially updating the version associated with the file by overriding the one specified in the file header (see Section 3.6.1, "Document Catalog"). As described in the following paragraphs, the viewer application's behavior upon opening or saving a document depends on what it perceives to be the document's PDF version (compared to the viewer's native file format—for example, PDF 1.3 for Acrobat 4.0—which is also referred to as the viewer's PDF version). Viewers that are not PDF 1.4–aware may perceive the document's version incorrectly, because they will look for it only in the PDF file's header and will not see the version (if any) specified in the document catalog.

An Acrobat viewer will attempt to read any PDF file, even if the file's version is more recent than that of the viewer itself. It will read without errors any file that does not require a plug-in extension, even if the file's version is older than the viewer's. Some documents may require a plug-in to display an annotation, follow a link, or execute an action. Viewer behavior in this situation is described below in Section H.3, "Implementation Notes." However, a plug-in is never required in order to display the contents of a page.

If a viewer application opens a document with a major version number newer than it expects, it warns the user that it is unlikely to be able to read the document successfully and that the user will not be able to change or save the document. At the first error related to document processing, the viewer notifies the user that an error has occurred but that no further errors will be reported. (Some errors are always reported, including file I/O errors, extension loading errors, out-of-memory errors, and notifications that a command has failed.) Processing continues if possible. Acrobat does not permit a document with a newer than expected major version number to be inserted into another document.

If a viewer application opens a document with a minor version number newer than it expects, it notifies the user that the document may contain information the viewer does not understand. (This describes the behavior in Acrobat 5.0 and later; versions prior to that do not so notify the user.) If the viewer encounters an error, it notifies the user that the document version is newer than expected, that an error has occurred, and that no further errors will be reported. Acrobat permits a document with a newer minor version to be inserted into another document.

Whether and how the version of a document changes when the document is modified and saved depends on several factors. If the document has a newer version than expected, the viewer will not alter the version—that is, a document's version will never be changed to an older version. If the document has an older version than expected, the viewer will update the document's version to match the viewer's version. If a user modifies a document by inserting another document into it, the saved document's version will be whichever is the most recent of the viewer's version, the document's original version, and the inserted document's version.

If the version of a document changes, viewers that are not PDF 1.4–aware will not be able to save the document using an incremental update, because updating the header requires rewriting the entire file. Among other disadvantages, this can cause existing digital signatures to become invalid. Since viewers that are PDF 1.4–aware can use the **Version** entry in the document catalog to update the document's version, they can incrementally save the document (and will do so if necessary to preserve existing signatures). For example, if an Acrobat 5.0 user modifies a document having a PDF version earlier than 1.4, the document may be updated incrementally when saved (with the updated version of 1.4 in the document catalog); however, if an Acrobat 4.0 user modifies a document having a PDF version earlier than 1.3, the entire file will be rewritten when saved (with a new header indicating version 1.3).

Again, the discussion of viewer behavior above applies to what the viewer perceives to be a document's PDF version, which may be different from the document's actual version if the viewer does not look for the **Version** entry in the document's catalog (a PDF 1.4 feature). One consequence of this is that a file may be rewritten when it could have been incrementally updated. For example, suppose an Acrobat 4.0 user opens a document having a version of 1.4 (newer than expected), specified in the catalog's **Version** entry. Acrobat 4.0 will determine the version by looking only at the document's header. There are two cases to consider:

- The header specifies version 1.2 or earlier. If the user alters and saves the document, the viewer will update the document's version to match its own, by rewriting the file with a new header indicating version 1.3.
- The header specifies version 1.3 or later. If the user alters and saves the document, the viewer will allow the file to be incrementally updated, since it does not believe the version needs updating.

In both cases, the version number in the document catalog will be maintained at 1.4, so later versions of Acrobat will recognize the correct version number.

H.2 Feature Compatibility

Many PDF features are introduced simply by adding new entries to existing dictionaries. Earlier versions of viewer applications will not notice the existence of such entries and will behave as if they were not there. Such new features are therefore both forward- and backward-compatible. Likewise, adding entries not described in the PDF specification to dictionary objects does not affect the viewers' behavior. (See Appendix E for information on how to choose key names that are compatible with future versions of PDF.)

In some cases, a new feature is impossible to ignore, because doing so would preclude some vital operation such as viewing or printing a page. For instance, if a page's content stream is encoded with some new type of filter, then there is no way to view or print the page, even though the content stream (if decoded) would be perfectly understood by the viewer. There is little choice but to give an error in cases like these. Such new features are forward-compatible, but not backward-compatible.

There are a few cases in which new features are defined in a way that earlier viewer versions will ignore, but the output will be degraded in some way without any error indication. The most significant example of this is transparency. All of the transparency features introduced in PDF 1.4 are defined as new entries in existing dictionaries (including the graphics state parameter dictionary). A viewer that does not understand transparency will treat transparency group XObjects as if they were opaque form XObjects. This is a significant enough deviation from the intended behavior that it is worth pointing out as a compatibility issue (and so is covered in implementation notes in this appendix).

If a PDF document undergoes editing by an application that does not understand some of the features that the document uses, the occurrences of those features may or may not survive. If a dictionary object such as an annotation is copied into another document during a page insertion (or, beginning with Acrobat 2.0, during a page extraction), all entries are copied. If a value is an indirect reference to another object, that object may be copied as well, depending on the entry.

H.3 Implementation Notes

This section gives notes on the implementation of Adobe Acrobat and on compatibility between different versions of PDF. The notes are listed in the order of the sections to which they refer in the main text.

1.2, "Introduction to PDF 1.5 Features"

1. The native file formats of Adobe Acrobat products are PDF 1.2 for Acrobat 3.0, PDF 1.3 for Acrobat 4.0, PDF 1.4 for Acrobat 5.0, and PDF 1.5 for Acrobat 6.0.

3.1.2, "Comments"

2. Acrobat viewers do not preserve comments when saving a file.

3.2.4, "Name Objects"

3. In PDF 1.1, the number sign character (#) could be used as part of a name (for example, /A#B), and the specifications did not specifically prohibit embedded spaces (although Adobe producer applications did not provide a way to write names containing them). In PDF 1.2, the number sign became an escape character, preceding two hexadecimal digits. Thus a 3-character name A-space-B can now be written as /A#20B (since 20 is the hexadecimal code for the space character). This means that the name /A#B is no longer valid, since the number sign is not followed by two hexadecimal digits. A name object with this value must be written as /A#23B, since 23 is the hexadecimal code for the character #.

4. In cases where a PostScript name must be preserved, or where a string is permitted in PostScript but not in PDF, the Acrobat Distiller application uses the # convention as necessary. When an Acrobat viewer generates PostScript, it "inverts" the convention by writing a string, where that is permitted, or a name otherwise. For example, if the string (Adobe Green) were used as a key in a dictionary, the Distiller program would use the name /Adobe#20Green and the viewer would generate (Adobe Green).

5. In Acrobat 4.0 and earlier versions, a name object being treated as text will typically be interpreted in a host platform encoding, which depends on the operating system and the local language. For Asian languages, this encoding may be something like Shift-JIS or Big Five. Consequently, it will be necessary to distinguish between names encoded this way and ones encoded as UTF-8. Fortunately, UTF-8 encoding is very stylized and its use can usually be recognized. A name that is found not to conform to UTF-8 encoding rules can instead be interpreted according to host platform encoding.

3.2.7, "Stream Objects"

6. When a stream specifies an external file, PDF 1.1 parsers ignore the file and always use the bytes between **stream** and **endstream**.

7. Acrobat viewers accept the name **DP** as an abbreviation for the **DecodeParms** key in any stream dictionary. If both **DP** and **DecodeParms** entries are present, **DecodeParms** takes precedence.

3.2.9, "Indirect Objects"

8. Acrobat viewers require that the name object used as a key in a dictionary entry be a direct object; an indirect object reference to a name is not accepted.

3.3, "Filters"

9. Acrobat viewers accept the abbreviated filter names shown in Table H.1 in addition to the standard ones. Although the abbreviated names are intended for use only in the context of inline images (see Section 4.8.6, "Inline Images"), they are also accepted as filter names in any stream object.

TABLE H.1 Abbreviations for standard filter names

STANDARD FILTER NAME	ABBREVIATION
ASCIIHexDecode	**AHx**
ASCII85Decode	**A85**
LZWDecode	**LZW**
FlateDecode *(PDF 1.2)*	**Fl** (uppercase F, lowercase L)
RunLengthDecode	**RL**
CCITTFaxDecode	**CCF**
DCTDecode	**DCT**

10. If an unrecognized filter is encountered, Acrobat viewers report the context in which the filter was found. If errors occur while a page is being displayed, only the first error is reported. The subsequent behavior depends on the context, as described in Table H.2. Acrobat operations that process pages, such as the Find command and the Create Thumbnails command, stop as soon as an error occurs.

TABLE H.2 Acrobat behavior with unknown filters

CONTEXT	BEHAVIOR
Content stream	Page processing stops.
Indexed color space	The image does not appear, but page processing continues.
Image resource	The image does not appear, but page processing continues.
Inline image	Page processing stops.
Thumbnail image	An error is reported and no more thumbnail images are displayed, but the thumbnails can be deleted and created again.
Form XObject	The form does not appear, but page processing continues.
Type 3 glyph description	The glyph does not appear, but page processing continues. The text position is adjusted based on the glyph width.
Embedded font	The viewer behaves as if the font is not embedded.

3.3.7, "DCTDecode Filter"

11. Acrobat 4.0 and later viewers do not support the combination of the **DCTDecode** filter with any other filter if the encoded data uses the progressive JPEG format. If a version of the Acrobat viewer earlier than 4.0 encounters **DCTDecode** data encoded in progressive JPEG format, an error occurs that will be handled according to Table H.2.

3.4, "File Structure"

12. The restriction on line length is not enforced by any Acrobat viewer.

3.4.1, "File Header"

13. Acrobat viewers require only that the header appear somewhere within the first 1024 bytes of the file.
14. Acrobat viewers will also accept a header of the form

%!PS–Adobe–*N.n* PDF–*M.m*

3.4.3, "Cross-Reference Table"

15. Acrobat viewers do not enforce the restriction on object numbers existing in more than one subsection, and use the entry in the first subsection where the object number is encountered. However, overlap is explicitly prohibited in cross-reference streams in PDF 1.5.
16. Acrobat viewers do not raise an error in cases where there are gaps in the sequence of object numbers between cross-reference subsections. The missing object numbers are treated as free objects.

3.4.4, "File Trailer"

17. Acrobat viewers require only that the %%EOF marker appear somewhere within the last 1024 bytes of the file.

3.4.6, "Object Streams"

18. When creating or saving PDF files, Acrobat 6.0 limits the number of objects in individual object streams to 100 for linearized files and 200 for non-linearized files.

3.5, "Encryption"

19. An Acrobat viewer will fail to open a document encrypted with a **V** value defined in a version of PDF that the viewer does not understand.

3.5.2, "Standard Security Handler" (Standard Encryption Dictionary)

20. Acrobat viewers implement this limited mode of printing as "Print As Image," except on UNIX systems, where this feature is not available.

3.5.2, "Standard Security Handler" (Password Algorithms)

21. In Acrobat 2.0 and 2.1 viewers, the standard security handler uses the empty string if there is no owner password in step 1 of Algorithm 3.3.

3.5.4, "Crypt Filters"

22. In Acrobat 6.0. crypt filter usage is limited to metadata streams and embedded files.
23. In Acrobat 6.0, when strings and streams in an encrypted document are edited, those streams and strings are encrypted with the **StmF** and **StrF** filters, respectively.
24. In Acrobat 6.0, support for embedded file authorization is limited to embedded files specified by **EmbeddedFiles** in the name dictionary (see Table 3.28). In the file specification dictionary (see Section 3.10.2, "File Specification Dictionaries"), related files (**RF**) must use the same crypt filter as the embedded file (**EF**).

3.6.1, "Document Catalog"

25. Acrobat 5.0 and Acrobat 6.0 avoid adding a **Version** entry to the document catalog and do so only if they must. Once they have done so:
    - Acrobat 5.0 never removes the **Version** entry. For documents containing a **Version** entry, Acrobat 5.0 attempts to ensure that the version specified in the header matches the version specified in the **Version** entry; if this is not possible, it at least ensures that the latter is later than (and therefore overrides) the version specified in the header.
    - Acrobat 6.0 removes the **Version** entry when doing a full (non-incremental) save of the document.

26. An earlier version of this specification documented the **PageLayout** entry as being in the viewer preferences dictionary (see Section 8.1, "Viewer Preferences"); it is actually implemented in the document catalog instead.

27. In PDF 1.2, an additional entry in the document catalog, named **AA**, was defined but was never implemented. The **AA** entry that is newly introduced in PDF 1.4 is entirely different from the one that was contemplated for PDF 1.2.

3.6.2, "Page Tree" (Page Objects)

28. In PDF 1.2, an additional entry in the page object, named **Hid**, was defined but was never implemented. Beginning with PDF 1.3, this entry is obsolete and should be ignored.

29. Acrobat 5.0 and later viewers do not accept a **Contents** array containing no elements.

30. In a document containing articles, if the first page with an article bead does not have a **B** entry, Acrobat viewers rebuild the **B** array for all pages of the document.

31. In PDF 1.2, additional-actions dictionaries were inheritable; beginning with PDF 1.3, they no longer are.

3.7.1, "Content Streams"

32. Acrobat viewers report an error the first time they find an unknown operator or an operator with too few operands, but continue processing the content stream. No further errors are reported.

3.9.1, "Type 0 (Sampled) Functions"

33. When printing, Acrobat performs only linear interpolation, regardless of the value of the **Order** entry.

3.9.2, "Type 2 (Exponential Interpolation) Functions"

34. Since Type 2 functions are not defined in PDF 1.2 or earlier versions, Acrobat 3.0 (whose native file format is PDF 1.2) will report an error, "Invalid Function Resource," if it encounters a function of this type.

3.9.3, "Type 3 (Stitching) Functions"

35. Since Type 3 functions are not defined in PDF 1.2 or earlier versions, Acrobat 3.0 (whose native file format is PDF 1.2) will report an error, "Invalid Function Resource," if it encounters a function of this type.

3.9.4, "Type 4 (PostScript Calculator) Functions"

36. Since Type 4 functions are not defined in PDF 1.2 or earlier versions, Acrobat 3.0 (whose native file format is PDF 1.2) will report an error, "Invalid Function Resource," if it encounters a function of this type.

37. Acrobat uses single-precision floating-point numbers for all real-number operations in a type 4 function.

3.10.2, "File Specification Dictionaries"

38. In Acrobat 5, file specifications accessed through **EmbeddedFiles** have a **Type** entry whose value is **F** instead of the correct **Filespec**. Acrobat 6.0 and later accept a file specification whose **Type** entry is either **Filespec** or **F**.

4.5.2, "Color Space Families"

39. If an Acrobat viewer encounters an unknown color space family name, it displays an error specifying the name, but reports no further errors thereafter.

4.5.5, "Special Color Spaces" (DeviceN Color Spaces)

40. Acrobat viewers support the special meaning of **None** only when a **DeviceN** color space is used as a base color for an indexed color space. For all other uses of **DeviceN**, **None** is treated as a regular spot color name.

4.5.5, "Special Color Spaces" (Multitone Examples)

41. This method of representing multitones is used by Adobe Photoshop 5.0.2 and subsequent versions when exporting EPS files. Beginning with version 4.0, Acrobat exports Level 3 EPS files using this method, and can also export Level 1 EPS files that use the "Level 1 separation" conventions of Adobe Technical Note #5044, *Color Separation Conventions for PostScript Language Programs*. These conventions are used to emit multitone images

as calls to “customcolorimage” with overprinting, which can then be placed in page layout applications such as Adobe PageMaker®, Adobe In-Design, and QuarkXPress.

4.6, “Patterns”

42. Acrobat viewers prior to version 4.0 do not display patterns on the screen, although they do print them to PostScript output devices.

4.7, “External Objects”

43. If an Acrobat viewer encounters an XObject of an unknown type, it displays an error specifying the type of XObject, but reports no further errors thereafter.

4.8.4, “Image Dictionaries”

44. Image XObjects in PDF 1.2 and earlier versions are all implicitly unmasked images. A PDF consumer that does not recognize the **Mask** entry will treat the image as unmasked without raising an error.
45. All Acrobat viewers ignore the **Name** entry in an image dictionary.

4.8.5, “Masked Images”

46. Explicit masking and color key masking are features of PostScript LanguageLevel 3. Acrobat 4.0 and later versions do not attempt to emulate the effect of masked images when printing to LanguageLevel 1 or LanguageLevel 2 output devices; they print the base image without the mask.

    The Acrobat 4.0 viewer will display masked images, but only when the amount of data in the mask is below a certain limit. Above that, the viewer will display the base image without the mask.

4.9.1, “Form Dictionaries”

47. All Acrobat viewers ignore the **Name** entry in a form dictionary.

4.9.3, "Reference XObjects

48. Acrobat 6.0 and earlier viewers do not implement reference XObjects. The proxy is always used for viewing and printing.

5.2.5, "Text Rendering Mode"

49. In Acrobat 4.05 and earlier versions, text-showing operators such as **Tj** first perform the fills for all the glyphs in the string being shown, followed by the strokes for all the glyphs. This produces incorrect results if glyphs overlap.

5.3.2, "Text-Showing Operators"

50. In versions of Acrobat prior to 3.0, the horizontal coordinate of the text position after the **TJ** operator paints a character glyph and moves by any specified offset must not be less than it was before the glyph was painted.

51. In Acrobat 4.0 and earlier viewers, position adjustments specified by numbers in a **TJ** array are performed incorrectly if the horizontal scaling parameter, T_h, is different from its default value of 100.

5.5.1, "Type 1 Fonts"

52. All Acrobat viewers ignore the **Name** entry in a font dictionary.

53. Acrobat 5.0 and later viewers use the glyph widths stored in the font dictionary to override the widths of glyphs in the font program itself, which improves the consistency of the display and printing of the document. This addresses the situation in which the font program used by the viewer application is different from the one used by the application that produced the document.

    The font program with the altered glyph widths might be embedded or not. If it is embedded, its widths should exactly match the widths in the font dictionary. If the font program is not embedded, Acrobat will override the widths in the font program on the viewer application's system with the widths specified in the font dictionary.

    It is important that the widths in the font dictionary match the actual glyph widths of the font program that was used to produce the document. Consumers of PDF files depend on these widths in many different contexts, including viewing, printing, "fauxing" (font substitution), reflow, and word search. These operations may malfunction if arbitrary adjustments are made to the widths so that they do not represent the glyph widths intended by the PDF producer.

It is recommended that diagnostic and preflight tools check the glyph widths in the font dictionary against those in an embedded font program and flag any inconsistencies. It would also be helpful if the tools could optionally check for consistency with the widths in font programs that are not embedded; this is useful for checking a PDF file immediately after it is produced, when the original font programs are still available.

Note: *This implementation note is also referred to in Section 5.6.3, "CIDFonts" (Glyph Metrics in CIDFonts).*

5.5.1, "Type 1 Fonts" (Standard Type 1 Fonts)

54. Acrobat 3.0 and earlier viewers may ignore attempts to override the standard fonts. Also, Acrobat 4.0 and earlier viewers incorrectly allow substitution fonts, such as TimesNewRoman and ArialMT, to be specified without **FirstChar**, **LastChar**, **Widths**, and **FontDescriptor** entries.

    Table H.3 shows the complete list of font names that are accepted as the names of standard fonts. The first column shows the proper one (for example, Helvetica); the second column shows the alternative if one exists (for example, Arial).

TABLE H.3 Names of standard fonts

STANDARD NAME	ALTERNATIVE
Courier	CourierNew
Courier–Oblique	CourierNew,Italic
Courier–Bold	CourierNew,Bold
Courier–BoldOblique	CourierNew,BoldItalic
Helvetica	Arial
Helvetica–Oblique	Arial,Italic
Helvetica–Bold	Arial,Bold
Helvetica–BoldOblique	Arial,BoldItalic

STANDARD NAME	ALTERNATIVE
Times–Roman	TimesNewRoman
Times–Italic	TimesNewRoman,Italic
Times–Bold	TimesNewRoman,Bold
Times–BoldItalic	TimesNewRoman,BoldItalic
Symbol	
ZapfDingbats	

5.5.3, “Font Subsets”

55. For Acrobat 3.0 and earlier viewers, all font subsets whose **BaseFont** names differ only in their tags should have the same font descriptor values and should map character names to glyphs in the same way; otherwise, glyphs may be shown unpredictably. This restriction is eliminated in Acrobat 4.0.

5.5.4, “Type 3 Fonts”

56. In principle, the value of the **Encoding** entry could also be the name of a predefined encoding or an encoding dictionary whose **BaseEncoding** entry is a predefined encoding. However, Acrobat 4.0 and earlier viewers do not implement this correctly.

57. For compatibility with Acrobat 2.0 and 2.1, the names of resources in a Type 3 font’s resource dictionary must match those in the page object’s resource dictionary for all pages in which the font is referenced. If backward compatibility is not required, any valid names may be used.

5.6.4, “CMaps”

58. Embedded CMap files, other than **ToUnicode** CMaps, do not work properly in Acrobat 4.0 viewers; this has been corrected in Acrobat 4.05.

59. Japanese fonts included with Acrobat 6.0 contain only glyphs from the Adobe Japan1-4 character collection. Documents that use fonts containing additional glyphs from the Adobe-Japan1-5 collection must embed those fonts in order to ensure proper display and printing.

5.7, "Font Descriptors"

60. Acrobat viewers prior to version 3.0 ignore the **FontFile3** entry. If a font uses the Adobe standard Latin character set (as defined in Section D.1, "Latin Character Set and Encodings"), Acrobat creates a substitute font. Otherwise, Acrobat displays an error message (once per document) and substitutes any characters in the font with the bullet character.

5.8, "Embedded Font Programs"

61. For simple fonts, font substitution is performed using multiple master Type 1 fonts. This substitution can be performed only for fonts that use the Adobe standard Latin character set (as defined in Section D.1, "Latin Character Set and Encodings"). In Acrobat 3.0.1 and later, Type 0 fonts that use a CMap whose **CIDSystemInfo** dictionary defines the Adobe-GB1, Adobe-CNS1 Adobe-Japan1, or Adobe-Korea1 character collection can also be substituted. To make a document portable, it is necessary to embed fonts that cannot be substituted. The only exceptions are the Symbol and ZapfDingbats fonts, which are assumed to be present.

6.4.2, "Spot Functions"

62. When the Acrobat Distiller encounters a call to the PostScript **setscreen** or **sethalftone** operator that includes a spot function, it compares the PostScript code defining the spot function with that of the predefined spot functions shown in Table 6.1. If the code matches one of the predefined functions, Distiller puts the name of that function into the halftone dictionary; Acrobat will then use that function when printing the PDF document to a PostScript output device. If the code does not match any of the predefined spot functions, Distiller samples the specified spot function and generates a function for the halftone dictionary; when printing to a PostScript device, Acrobat will generate a spot function that interpolates values from that function.

    When producing PDF version 1.3 or later, Distiller represents the spot function using a Type 4 (PostScript calculator) function whenever possible (see Section 3.9.4, "Type 4 (PostScript Calculator) Functions"). In this case, Acrobat will use this function directly when printing the document.

6.5.4, “Automatic Stroke Adjustment”

63. When drawing to the screen, Acrobat 6.0 always performs automatic stroke adjustment, regardless of the value of the **SA** entry in the graphics state parameter dictionary.

7.5.2, “Specifying Blending Color Space and Blend Mode”

64. PDF 1.3 or earlier viewers will ignore all transparency-related graphics state parameters (blend mode, soft mask, alpha constant, and alpha source). All graphics objects will be painted opaquely.

***Note:** This implementation note is also referred to in Sections 7.5.3, “Specifying Shape and Opacity” (Mask Shape and Opacity, Constant Shape and Opacity) and 7.5.4, “Specifying Soft Masks” (Soft-Mask Dictionaries).*

7.5.3, “Specifying Shape and Opacity” (Mask Shape and Opacity)

65. PDF 1.3 or earlier viewers will ignore the **SMask** entry in an image dictionary. All images will be painted opaquely.

***Note:** This implementation note is also referred to in Section 7.5.4, “Specifying Soft Masks” (Soft-Mask Images).*

8.1, “Viewer Preferences”

66. Earlier versions of the PDF specification erroneously described an additional entry, **PageLayout**, as being in the viewer preferences dictionary; it is actually implemented in the document catalog instead (see Section 3.6.1, “Document Catalog”).

8.2.2, “Document Outline”

67. In PDF 1.2, an additional entry in the outline item dictionary, named **AA**, was defined but was never implemented. Beginning with PDF 1.3, this entry is obsolete and should be ignored.
68. Acrobat viewers report an error when a user activates an outline item whose destination is of an unknown type.

8.3.1, "Page Labels"

69. Acrobat viewers up to version 3.0 ignore the **PageLabels** entry and label pages with decimal numbers starting at 1.

8.4, "Annotations"

70. In PDF 1.5, the order of moving the keyboard focus between annotations on a page with the tab key can be made explicit by means of the page's **Tabs** entry (see Table 3.27). In prior versions, the tab order was not explicitly specified and depended upon the viewer. In Acrobat 4.0, the order includes only widget annotations and is determined by their order in the page's **Annots** array. In Acrobat 5.0, the order includes all annotations: widgets come first and are ordered as in Acrobat 4.0; other annotations come after widgets and are ordered by rows. Acrobat 6.0 has the same behavior as Acrobat 5.0 for documents that do not contain a **Tabs** entry. For documents that have a **Tabs** entry, Acrobat 6.0 re-orders widgets in the **Annots** array to match the specified order (row, column, or structure) so that the tab order for widgets is preserved when the document is opened by earlier viewers.

8.4.1, "Annotation Dictionaries"

71. Acrobat viewers update the annotation dictionary's **M** entry only for text annotations.
72. Acrobat 2.0 and 2.1 viewers ignore the annotation dictionary's **BS**, **AP**, and **AS** entries.
73. All versions of Acrobat through 6.0 ignore the **AP** entry when drawing the appearance of link annotations.
74. Acrobat viewers ignore the horizontal and vertical corner radii in the annotation dictionary's **Border** entry; the border is always drawn with square corners.
75. Acrobat viewers support a maximum of ten elements in the dash array of the annotation dictionary's **Border** entry.

8.4.2, "Annotation Flags"

76. Acrobat viewers prior to version 3.0 ignore an annotation's Hidden and Print flags. Annotations that should be hidden are shown; annotations that should be printed are not printed. Acrobat 3.0 ignores the Print flag for text and link annotations.

77. Acrobat 5.0 obeys the Locked flag only for widget annotations. In Acrobat 6.0, markup annotations support it as well.

78. In Acrobat 6.0, the ToggleNoView flag is applicable to mouse-over and selection events.

8.4.3, "Border Styles"

79. If an Acrobat viewer encounters a border style it does not recognize, the border style defaults to S (Solid).

8.4.4, "Appearance Streams"

80. Acrobat 5.0 treats the annotation appearance as an isolated group, whether or not a **Group** entry is present. This behavior is corrected in Acrobat 6.0.

8.4.5, "Annotation Types"

81. Acrobat viewers display annotations whose types they do not recognize in closed form, with an icon containing a question mark. Such an annotation can be selected, moved, or deleted, but if the user attempts to activate it, an alert appears giving the annotation type and reporting that a required plug-in is unavailable.

8.4.5, "Annotation Types" (Link Annotations)

82. Acrobat viewers report an error when a user activates a link annotation whose destination is of an unknown type.

83. When a link annotation specifies a value of P for the **H** entry (highlighting mode), Acrobat viewers display the link appearance with a beveled border, ignoring any down appearance (see Section 8.4.4, "Appearance Streams") that is defined.

8.4.5, "Annotation Types" (Text Markup Annotations)

84. In Acrobat 4.0 and later versions, the text is oriented with respect to the vertex with the smallest *y* value (or the leftmost of those, if there are two such vertices) and the next vertex in a counterclockwise direction, regardless of whether these are the first two points in the **QuadPoints** array.

8.4.5, "Annotation Types" (Ink Annotations)

85. Acrobat viewers always connect the points along each path with straight lines.

8.5.2, "Trigger Events"

86. In PDF 1.2, the additional-actions dictionary could contain entries named **NP** (next page), **PP** (previous page), **FP** (first page), and **LP** (last page). The actions associated with these entries were never implemented; beginning with PDF 1.3, these entries are obsolete and should be ignored.
87. In PDF 1.2, additional-actions dictionaries were inheritable; beginning with PDF 1.3, they no longer are.
88. In Acrobat 3.0, the **O** and **C** events in a page object's additional-actions dictionary are ignored if the document is not being displayed in a page-oriented layout mode. Beginning with Acrobat 4.0, the actions associated with these events are executed if the document is in a page-oriented or single-column layout; they are ignored if it is in a multiple-column layout.

8.5.3, "Action Types" (Launch Actions)

89. The Acrobat viewer for the Windows platform uses the Windows function ShellExecute to launch an application. The **Win** dictionary entries correspond to the parameters of ShellExecute.

8.5.3, "Action Types" (URI Actions)

90. URI actions are resolved by the Acrobat WebLink plug-in extension.
91. If the appropriate plug-in extension (WebLink) is not present, Acrobat viewers report the following error when a link annotation that uses a URI action is activated: "The plug-in required by this URI action is not available."

8.5.3, "Action Types" (Sound Actions)

92. In PDF 1.2, the value of the **Sound** entry was allowed to be a file specification. Beginning with PDF 1.3, this is no longer supported, but the same effect can be achieved by using an external stream.

93. Acrobat viewers mute the sound if the value of **Volume** is negative; otherwise, this entry is ignored.

94. Acrobat 6.0 does not support the **Synchronous** entry.

95. Acrobat 5.0 and earlier viewers do not support the **Mix** entry.

8.5.3, "Action Types" (Movie Actions)

96. Acrobat viewers prior to version 3.0 report an error when they encounter an action of type **Movie**.

8.5.3, "Action Types" (Hide Actions)

97. Acrobat viewers prior to version 3.0 report the following error when encountering an action of type **Hide**: "The plug-in needed for this **Hide** action is not available."

98. In Acrobat viewers, the change in an annotation's Hidden flag as a result of a hide action is temporary, in the sense that the user can subsequently close the document without being prompted to save changes, and the effect of the hide action will be lost. However, if the user does explicitly save the document before closing, such changes *will* be saved and will thus become permanent.

8.5.3, "Action Types" (Named Actions)

99. Acrobat viewers prior to version 3.0 report the following error when encountering an action of type **Named**: "The plug-in needed for this **Named** action is not available."

100. Acrobat viewers extend the list of named actions in Table 8.53 to include most of the menu item names available in the viewer.

8.6.1, "Interactive Form Dictionary"

101. Acrobat viewers may insert additional entries in the **DR** resource dictionary, such as **Encoding**, as a convenience for keeping track of objects being used to construct form fields. Such objects are not actually resources and are not referenced from the appearance stream.

102. In Acrobat, markup annotations can also make use of the resources in the **DR** dictionary.

8.6.2, "Field Dictionaries" (Field Names)

103. Beginning in Acrobat 3.0, partial field names may not contain a period.

104. Acrobat versions 3.0 and later do not support Unicode encoding of field names.

8.6.2, "Field Dictionaries" (Variable Text)

105. In PDF 1.2, an additional entry in the field dictionary, **DR**, was defined but was never implemented. Beginning with PDF 1.5, this entry is obsolete and should be ignored.

106. If the **MK** entry is present in the field's widget annotation dictionary (see Table 8.35), Acrobat viewers regenerate the entire XObject appearance stream. If **MK** is not present, the contents of the stream outside /Tx BMC ... EMC are preserved.

8.6.2, "Field Dictionaries" (Rich Text Strings)

107. To select a font specified by attributes in a rich text string, Acrobat 6.0 follows these steps, choosing the first appropriate font it finds:

   a. A font in the default resource dictionary (specified by the document's **DR** entry; see Table 8.58) whose font descriptor information matches the font specification in the rich text string. "Font Characteristics" on page 761 describes how this matching is done.

b. A matching font installed on the user's system, ignoring generic font families.

c. A font on the user's system that matches the generic font family, if specified.

d. A standard font (see implementation note 54) that most closely matches the other font specification properties and is appropriate for the current input locale.

8.6.2, "Field Dictionaries" (Button Fields)

108. The behavior of Acrobat has changed in the situation where a checkbox or radio button field have multiple children that have the same export value. In Acrobat 4, such buttons always turned off and on in unison. In Acrobat 5, the behavior of radio buttons was changed to mimic HTML, so that turning on a radio button always turned off its siblings regardless of export value. In Acrobat 6.0, the RadiosInUnison flag allows the document author to choose between these behaviors.

8.6.3, "Field Types" (Choice Fields)

109. In Acrobat 3.0, the **Opt** array must be homogenous: its elements must be either all text strings or all arrays.

8.6.4, "Form Actions" (Submit-Form Actions)

110. In Acrobat viewers, if the response to a submit-form action uses Forms Data Format (FDF), then the URL must end in #FDF so that it will be recognized as such by the Acrobat software and handled properly. Conversely, if the response is in any other format, the URL should not end in #FDF.

8.6.4, "Form Actions" (Import-Data Actions)

111. Acrobat viewers set the **F** entry to a relative file specification locating the FDF file with respect to the current PDF document file. If the designated FDF file is not found when the import-data action is performed, Acrobat tries to locate the file in a few "well-known" locations depending on the host platform. On the Windows platform, for example, it looks in the

directory from which Acrobat was loaded, the current directory, the System directory, the Windows directory, and any directories listed in the PATH environment variable; on Mac OS, it looks in the Preferences folder and the Acrobat folder.

112. When performing an import-data action, Acrobat viewers import the contents of the FDF file into the current document's interactive form, ignoring the **F** and **ID** entries in the FDF dictionary of the FDF file itself.

8.6.4, "Form Actions" (JavaScript Actions)

113. Because JavaScript 1.2 is not Unicode-compatible, **PDFDocEncoding** and the Unicode encoding are translated to a platform-specific encoding prior to interpretation by the JavaScript engine.

8.6.6, "Forms Data Format" (FDF Header)

114. Because a bug in versions of Acrobat prior to 5.0 prevents them from accepting any other version number, the FDF file header is permanently frozen at version 1.2. All further updates to the version number will be made via the **Version** entry in the FDF catalog dictionary instead.

8.6.6, "Forms Data Format" (FDF Catalog)

115. The Acrobat implementation of interactive forms displays the value of the **Status** entry, if any, in an alert note when importing an FDF file.

116. The only **Encoding** value supported by Acrobat 4.0 is Shift–JIS. Acrobat 5 supports Shift–JIS, UHC, GBK, and BigFive. If any other value is specified, the default, **PDFDocEncoding**, will be used.

8.6.6, "Forms Data Format" (FDF Fields)

117. Of all the possible entries shown in Table 8.87 on page 653, Acrobat 3.0 will export only the **V** entry when generating FDF, and Acrobat 4.0 and later versions will export only the **V** and **AP** entries. It will, however, import FDF files containing fields using any of the described entries.

118. If the FDF dictionary in an FDF file received as a result of a submit-form action contains an **F** entry specifying a form other than the one currently being displayed, Acrobat fetches the specified form before importing the FDF file.

119. When exporting a form to an FDF file, Acrobat sets the **F** entry in the FDF dictionary to a relative file specification giving the location of the FDF file relative to that of the file from which it was exported.

120. If an FDF file being imported contains fields whose fully qualified names are not present in the form, Acrobat will discard those fields. This feature can be useful, for example, if an FDF file containing commonly used fields (such as name and address) is used to populate various types of form, not all of which necessarily include all of the fields available in the FDF file.

121. As shown in Table 8.87 on page 653, the only required entry in the field dictionary is **T**. One possible use for exporting FDF with fields containing **T** entries but no **V** entries is to indicate to a server which fields are desired in the FDF files returned in response. For example, a server accessing a database might use this information to decide whether to transmit all fields in a record or just some selected ones. As noted in implementation note 120 above, the Acrobat implementation of interactive forms will ignore fields in the imported FDF file that do not exist in the form.

122. The Acrobat implementation of forms allows the option of submitting the data in a submit-form action in HTML Form format. This is for the benefit of existing server scripts written to process such forms. Note, however, that any such existing scripts that generate new HTML forms in response will need to be modified to generate FDF instead.

123. When scaling a button's appearance to the bounds of an annotation, versions of Acrobat prior to 6.0 always took into account the line width used to draw the border, even when no border was being drawn, Starting with Acrobat 6.0, the **FB** entry in the icon fit dictionary (see Table 8.88 on page 656) allows the option of ignoring the line width.

8.6.6, "Forms Data Format" (FDF Pages)

124. Acrobat renames fields by prepending a page number, a template name, and an ordinal number to the field name. The ordinal number corresponds to the order in which the template is applied to a page, with 0 being the first template specified for the page. For example, if the first template used on the fifth page has the name Template and has the **Rename** flag set to **true**, fields defined in that template will be renamed by prepending the character string P5.Template_0. to their field names.

125. Adobe Extreme® printing systems require that the **Rename** flag be **true**.

8.7, “Digital Signatures”

126. Acrobat computes a byte range digest only when the signature dictionary is referenced from a signature field. In Acrobat 6.0, the only cases where there is no byte range signature (that is, there is only an object signature) are FDF file signatures and usage rights signatures (referenced from the **UR** entry of a permissions dictionary).

9.1, “Multimedia”

127. The following media formats are recommended for use in authoring cross-platform PDF files intended for consumption by Acrobat 6.0.

TABLE H.4 Recommended media types

COMMON EXTENSION	COMMON MIME TYPE	DESCRIPTION
.aiff	audio/aiff	Audio Interchange File Format
.au	audio/basic	NeXT/Sun™ Audio Format
.avi	video/avi	AVI (Audio/Video Interleaved)
.mid	audio/midi	MIDI (Musical Instrument Digital Interface)
.mov	video/quicktime	QuickTime
.mp3	audio/x-mp3	MPEG Audio Layer-3
.mp4	audio/mp4	MPEG-4 Audio
.mp4	video/mp4	MPEG-4 Video
.mpeg	video/mpeg	MPEG-2 Video
.smil	application/smil	Synchronized Multimedia Integration Language
.swf	application/x-shockwave-flash	Macromedia Flash

128. If the **CT** entry is not present, Acrobat requires a **PL** entry to be present that specifies at least one player that can be used.

9.2, “Sounds”

129. Acrobat supports a maximum of two sound channels.

9.3, “Movies”

130. Acrobat viewers do not support the value of **Aspect**.

131. Acrobat viewers support only the **DeviceRGB** and **DeviceGray** color spaces for poster image XObjects. For indexed color spaces with a base color space of **DeviceRGB** (see “Indexed Color Spaces” on page 232”), Acrobat 5 viewers incorrectly treat *hival* as the number of colors, rather than the number of colors - 1. Acrobat 6.0 can handle this case properly, as well as the correct value of *hival*; for compatibility with 5.0 viewers, it is necessary to specify *hival* as the number of colors.

     Also, Acrobat viewers do not support authoring or rendering posters when the value of **Poster** is **true**.

132. Acrobat viewers treat a **FWScale** value of [999 1] as full screen.

133. Acrobat viewers never allow any portion of a floating window to be off-screen.

9.4, “Alternate Presentations”

134. The PDF language contains no direct method of initiating an alternate presentation-defined slideshow. Instead, a slideshow is invoked by a JavaScript call that is typically triggered by an interactive form element (see Section 8.6, “Interactive Forms”). Refer to Adobe Technical Note #5431, *Acrobat JavaScript Scripting Reference* (see the Bibliography) for information on starting and stopping a slideshow using JavaScript.

135. The only type of slideshow supported in Acrobat 5.1 and later is an SVG slideshow (MIME content type image/svg+xml). Acrobat supports the *Scalable Vector Graphics (SVG) 1.0 Specification* specification defined by the World Wide Web Consortium (see the Bibliography), with implementation notes on Adobe’s support of SVG available at <http://www.adobe.com/svg/>.

     All resources must be either image XObjects (see Section 4.8.4, “Image Dictionaries”) or embedded file streams (see Section 3.10.3, “Embedded File Streams”).

- Image XObjects used for slideshows must use the **DCTDecode** filter and an RGB color space. Color profile information must be specified in the image XObject dictionary as well as embedded within the JPEG stream.
- Embedded audio files must be of type .wav (supported on Windows only, MIME type audio/x-wav) or .mp3 (MIME type audio/mpeg, documented at <http://www.chiariglione.org/mpeg/index.htm>).
- Embedded video must be QuickTime-compatible files of type .avi (MIME type video/ms-video) or .mov (MIME type video/quicktime, documented at <http://developer.apple.com/documentation/QuickTime/PDF/QTFileFormat.pdf>). To play video, a QuickTime player (version 3 or later) must be installed.

10.1, "Procedure Sets"

136. Acrobat viewers prior to version 5.0 respond to requests for unknown procedure sets by warning the user that a required procedure set is unavailable and canceling the printing operation. Acrobat 5.0 ignores procedure sets.

10.2, "Metadata"

137. Acrobat viewers display the document's metadata in the Document Properties dialog box and impose a limit of 255 bytes on any string representing one of those values.

10.2.2, "Metadata Streams"

138. For backward compatibility, applications that create PDF 1.4 documents should include the metadata for a document in the document information dictionary as well as in the document's metadata stream. Applications that support PDF 1.4 should check for the existence of a metadata stream and synchronize the information in it with that in the document information dictionary. The Adobe metadata framework provides a date stamp for metadata expressed in the framework. If this date stamp is equal to or later than the document modification date recorded in the document information dictionary, the metadata stream can be taken as authoritative. If, however, the document modification date recorded in the document information dictionary is later than the metadata stream's date stamp, it is likely that the document has been saved by an application that is not aware

of PDF 1.4 metadata streams. In this case, information stored in the document information dictionary should be taken to override any semantically equivalent items in the metadata stream.

10.3, "File Identifiers"

139. Although the **ID** entry is not required, all Adobe applications that produce PDF files include this entry. Acrobat adds this entry when saving a file if it is not already present.

140. Adobe applications pass the suggested information to the MD5 message digest algorithm to calculate file identifiers. Note that the calculation of the file identifier need not be reproducible; all that matters is that the identifier is likely to be unique. For example, two implementations of this algorithm might use different formats for the current time; this will cause them to produce different file identifiers for the same file created at the same time, but does not affect the uniqueness of the identifier.

10.9.2, "Content Database" (Digital Identifiers)

141. The Acrobat Web Capture plug-in treats external streams referenced within a PDF file as auxiliary data. Such streams are not used in generating the digital identifier.

10.9.3, "Content Sets" (Image Sets)

142. In Acrobat 4.0 and later versions, if the indirect reference to an image XObject is not removed from the **O** array when its reference count reaches 0, the XObject will never be garbage-collected during a save operation. The image set's reference to the XObject may thus be considered a weak one that is relevant only for caching purposes; when the last strong reference goes away, so does the weak one.

10.9.4, "Source Information" (URL Alias Dictionaries)

143. Acrobat viewers use an indirect object reference to a shared string for each URL in a URL alias dictionary; these strings can then be shared among the chains and with other data structures. It is recommended that other PDF viewer applications adopt this same implementation.

10.10.1, "Page Boundaries"

144. Acrobat provides various user-specified options for determining how the region specified by the crop box is to be imposed on the output medium during printing. Although these options have varied from one Acrobat version to another, the default behavior is as follows:

    1. Select the media size and orientation according to the operating system's Print Setup dialog. (Acrobat itself has no direct control over this process.)
    2. Compute an effective crop box by first clipping it with the media box, then rotating the page according to the page object's **Rotate** entry, if specified.
    3. Center the crop box on the medium, rotating it if necessary to enable it to fit in both dimensions.
    4. Optionally, scale the page up or down so that the crop box coincides with the edges of the medium in the more restrictive dimension.

    The description above applies only in simple printing workflows that lack any other information about how PDF pages are to be imposed on the output medium. In some workflows, there will be additional information, either in the PDF file itself (**BleedBox**, **TrimBox**, or **ArtBox**) or in a separate job ticket (such as JDF or PJTF). In these circumstances, other rules will apply, which depend on the details of the workflow.

    Consequently, it is recommended that PDF files initially be created with the crop box the same as the media box (or equivalently, with the crop box omitted). This ensures that if the page is printed on that size medium, the crop box will coincide with the edges of the medium, producing predictable and dependable positioning of the page contents. On the other hand, if the page is printed on a different size medium, the page may be repositioned or scaled in implementation-defined or user-specified ways.

10.10.4, "Output Intents"

145. Acrobat viewers do not make use of the "to CIE" *(AToB)* information in an output intent's ICC profile.

146. Acrobat 5.0 does not make direct use of the destination profile in the output intent dictionary, but third-party plug-in extensions might do so. Acrobat 6.0 does make use of this profile.

10.10.5, "Trapping Support" (Trap Network Annotations)

147. Older viewers may fail to maintain the trap network annotation's required position at the end of the **Annots** array.

148. Older viewers may fail to validate trap networks before printing.

149. In Acrobat 4.0, saving a PDF file with the Optimize option selected would cause a page's trap networks to be incorrectly invalidated even if the contents of the page had not been changed. This occurred because the new, optimized content stream generated for the page differed from the original content stream still referenced by the trap network annotation's **Version** array. This problem has been corrected in later versions of Acrobat.

10.10.6, "Open Prepress Interface (OPI)"

150. The Acrobat 3.0 Distiller application converts OPI comments into OPI dictionaries; when the Acrobat 3.0 viewer prints a PDF file to a PostScript file or printer, it converts the OPI dictionary back to OPI comments. However, the OPI information has no effect on the displayed image or form XObject.

151. Acrobat viewer and Distiller applications prior to version 4.0 do not support OPI 2.0.

152. In Acrobat 3.0, the value of the **F** entry in an OPI dictionary must be a string.

C.1, "General Implementation Limits"

153. Acrobat viewers prior to 5.0 use the PostScript **save** and **restore** operators, rather than **gsave** and **grestore**, to implement **q** and **Q** and are subject to a nesting limit of 12 levels.

154. In Acrobat viewers prior to version 4.0, the minimum allowed page size is 72 by 72 units in default user space (1 inch by 1 inch); the maximum is 3240 by 3240 units (45 by 45 inches).

F.2.2, "Linearization Parameter Dictionary (Part 2)"

155. Acrobat requires a white-space character to follow the left bracket (`[`) character that begins the **H** array.

156. Acrobat does not currently support reading or writing files that have an overflow hint stream.

Note: This implementation note is also referred to in Section F.2.5, "Hint Streams (Parts 5 and 10)."

157. Acrobat generates a value for the **E** parameter that incorrectly includes an object beyond the end of the first page as if it were part of the first page.

F.2.6, "First-Page Section (Part 6)"

158. Acrobat always treats page 0 as the first page for linearization, regardless of the value of **OpenAction**.

F.2.8, "Shared Objects (Part 8)"

159. Acrobat does not generate shared object groups containing more than one object.

F.3.1, "Page Offset Hint Table"

160. In Acrobat, items 6 and 7 in the header section of the page offset hint table are set to 0. As a result, item 6 of the per-page entry effectively does not exist; its value is taken to be 0. That is, the sequence of bytes constituting the content stream for a page is described as if the content stream were the first object in the page, even though it is not.

161. Acrobat 4.0 and later versions always set item 8 equal to 0. They also set item 9 equal to the value of item 5, and set item 7 of each per-page hint table entry (Table F.4) to be the same as item 2 of the per-page entry. Acrobat ignores all of these entries when reading the file.

F.3.2, "Shared Object Hint Table"

162. In Acrobat, item 5 in the header section of the shared objects hint table is unused and is always set to 0.

163. MD5 signatures are not implemented in Acrobat; item 2 in a shared object group entry must be 0.

164. Acrobat does not support more than one shared object in a group; item 4 in a shared object group entry should always be 0.

165. In a document consisting of only one page, items 1 and 2 in the shared object hint table are not meaningful; Acrobat writes unpredictable values for these items.

APPENDIX I

Computation of Object Digests

This appendix describes the algorithm for computing object digests (discussed in Section 8.7, "Digital Signatures"). The computation uses a hashing method, specified by the **DigestMethod** entry of the signature reference dictionary (see Table 8.94). Its value can be **SHA1** for the Secure Hash Algorithm 1 (SHA-1) or **MD5** for the MD5 message-digest algorithm; see the Bibliography. Both algorithms operate on an arbitrary-length stream of bytes to produce a *digest* of fixed length (16 bytes for MD5, 20 bytes for SHA-1).

The following sections describe how the stream of bytes to be digested is generated, starting with a specific object within a PDF file. A PDF object is *digested* by recursively traversing the object hierarchy beginning with the given object. Objects encountered during the traversal are categorized as basic PDF types, described in Section I.1, "Basic Object Types," or more complex types, described in Section I.2, "Selective Computation." Each object is digested as it is processed. Not all objects may be included, depending on the transform method and parameters (see Section 8.7.2, "Transform Methods") that are being used.

I.1 Basic Object Types

The basic PDF object types are listed in Table I.1. For each type, the following data is digested:

- a single-byte type identifier
- other bytes representing the value of the data, as described in Table I.1

Dictionaries and arrays can contain indirect references to other objects; therefore, the data can be recursive. To prevent infinite recursions, the algorithm keeps track of all indirect objects visited during a recursive descent into a given object.

When it an encounters an object that has already been visited, it adds the type identifier followed by a 4-byte value for the number -1 (0xFFFFFFFF).

TABLE I.1 Data added to object digest for basic object types

OBJECT TYPE	TYPE IDENTIFIER	REMAINING VALUES ADDED TO DIGEST
Null	0	Nothing.
Integer	1	The unsigned 4-byte value of this integer, most significant byte first.
Real	2	The 4-byte integer corresponding to the integral part of the rounded value of the object.
Boolean	3	0x01 for **true**; 0x00 for **false**.
Name	4	An unsigned 4-byte integer (most significant byte first) representing the length of the name, followed by byte array containing the string representing the name (following expansion of any escape characters, and excluding the leading "/" character).
String	5	An unsigned 4-byte integer (most significant byte first) representing the length of the string, followed by the sequence of bytes corresponding to the string.
Dictionary	6	An unsigned 4-byte value (most significant byte first) specifying the number of entries in the dictionary, followed by the key-value pairs of the dictionary, sorted by lexicographic order of the keys (for comparison purposes, the key names are treated as binary byte sequences). The values may involve recursion; see above. Special treatment is given to certain dictionaries, when the transform method is anything but **Identity** (see Section 8.7.2, "Transform Methods"). For these dictionaries (which include catalog, page, named page, form field, annotation, action and additional-actions dictionaries), all key-value pairs are not digested; rather, only the values of specified entries are digested; see Section I.2, "Selective Computation," for details.
Array	7	An unsigned 4-byte value (most significant byte first) specifying the number of entries in the array, followed by the individual entries, in order. Individual entries may involve recursion. Specific entries may be excluded when dictated by the transform method and parameters (for example, annotation dictionaries in a page's **Annots** array).

OBJECT TYPE	TYPE IDENTIFIER	REMAINING VALUES ADDED TO DIGEST
Stream	8	The following values, in order: • An unsigned 4-byte value (most significant byte first) specifying the number of entries in the stream dictionary • The following key-value pairs in the stream dictionary, if present, sorted as follows: **DecodeParms**, **F**, **FDecodeParms**, **FFilter**, **Filter** and **Length**. • An unsigned 4-byte value (most significant byte first) specifying the length of the stream • The stream data.

I.2 Selective Computation

There is a set of special objects that, when encountered in an object calculation, are not treated as described in the previous section. These objects are described in the following sections. For each of them:

- A selective list of entries is chosen.
- Only the value of the entry is digested; the key is not included.

When the transform method is **DocMDP** (see "DocMDP" on page 666)or **UR** (see "UR" on page 667), the object digest is computed over the entire document (see Section I.2.1, "Document"). The calculation varies depending on the transform parameters, which may specify whether form fields or annotations are included, for example.

When the transform method is **FieldMDP** (see "FieldMDP" on page 669), the transform parameters indicate specific form fields over which the object digest should be computed. For each form field, the digest calculation is performed as specified in Section I.2.6, "Form Fields."

When the transform method is **Identity**, selective computation is not used. All objects are processed as basic object types as described in Section I.1, "Basic Object Types."

I.2.1 Document

When calculating a digest for the document, the following items are included, in order:

- The values of following entries in the document catalog (see Table 3.25), if present: **AA**, **Legal**; and **Perms**.
- The value of the following entries in the document information dictionary (see Table 10.2), if present: **Title**, **Author**, **Keywords**, and **Subject**.
- All page objects in the document, in page order, as described in Section I.2.2, "Page Objects."
- All named pages specified in the **Pages** name tree, sorted by name, as described in Section I.2.3, "Named Pages."
- All embedded files specified in the **EmbeddedFiles** name tree, sorted by name, as described in Section I.2.4, "Embedded Files."

I.2.2 Page Objects

For page objects (see Table 3.27), the digest includes the values of the following entries, in order, if present. For entries listed as inheritable, their values may be inherited from ancestor nodes in the page tree if not specified explicitly.

- **MediaBox** (inheritable)
- **CropBox** (inheritable)
- **Resources** (inheritable)
- **Contents**
- **Rotate** (inheritable)
- **AA**
- **Annots**. This entry consists of an array of dictionaries for annotations on the page. They are sorted by the value of the **NM** entry; if **NM** is not present, a globally unique ID (GUID) is supplied as **NM**.

  For each annotation, if it is a widget, the values added to the digest are those specified in Section I.2.6, "Form Fields." If it is any other kind of annotation, the values added to the digest are those specified in Section I.2.5, "Annotation Dictionaries." However, when the transform parameters specify that annotations may be modified (for example, when the value of **P** is 3 for the **DocMDP** transform method), annotation dictionaries other than widgets are not included.

Note: *Pages that have a* ***TemplateInstantiated*** *entry are not included in the digest, when the transform method indicates that page template instantiation is permitted. Instead, a separate calculation is performed to compare instantiated pages with their associated named pages; see Section I.2.9, "Page Template Verification."*

I.2.3 Named Pages

For named pages (see Section 8.6.5, "Named Pages"), only the **Contents** and **Annots** entries are digested, as shown in Section I.2.2, "Page Objects," above.

I.2.4 Embedded Files

The document's embedded files (as specified in the **EmbeddedFiles** name tree) are sorted by name. For each embedded file, the following values are digested, in order:

- Then name of the embedded file.
- The stream corresponding to the file.

I.2.5 Annotation Dictionaries

For annotation dictionaries (see Table 8.38), the values of the following entries are digested, in order, if present:

- **Contents**

  Note: *A content stream of the form "()" (parentheses enclosing nothing) is considered nonexistent content and is not included.*
- **T**
- **F**
- **A**
- **AA**
- **Dest**
- **QuadPoints**

- **Inklist**
- **Name**
- **FS** (If **FS** refers to the contents of a remote file, the contents of that file are not digested)
- **Sound**
- If **Movie** is present, the value of its **F** and **Poster** entries
- For stamp annotations only, the value of the **AP** entry.

I.2.6 Form Fields

For form fields (see Table 8.60), the values of the following entries are digested, in order, if present:

***Note:** The **A**, **AA**, **Rect** and **F** entries are from the annotation dictionary (see Table 8.11); all others are from the form field dictionary (see Tables 8.60 and 8.72).*

- **T** (the unqualified name)
- **FT** (inheritable)
- **DV** (inheritable)
- **V**. This value is included only in the cases where the transform method and parameters specify that form field fill-in is not allowed, or that this particular field is locked.
- **A** (inheritable)
- **AA** (inheritable)
- **Rect**
- **F** (the annotation flags). If necessary, the flag values are obtained by traveling the inheritance hierarchy.
- **Lock** (signature fields only)
- **SV** (signature fields only)

I.2.7 Actions

For most actions (see Section 8.5, "Actions"), the values of the following entries in the action dictionary are digested, in order, if present: **S** (required), **D**, **F**, **NewWindow**, **O**, **P**, **B**, **Base**, **Sound**, **Vol**, **Annot**, **T**, **H**, **N**, **JS** and **URI**.

Rendition actions (see "Rendition Actions" on page 609) are treated differently than the other types. The data that is digested is media data that is nested in several levels of objects, as follows:

- The rendition action's **R** entry, if present, specifies a rendition object (see Section 9.1.2, "Renditions") whose **S** entry determines whether it is a media rendition or a selector rendition.
- Selector renditions have an **R** entry specifying an array of renditions, which may themselves be selector renditions. This array is searched recursively for all media renditions, which are then processed as specified below.
- Media renditions have a **C** entry that refers to a media clip dictionary. If the **S** entry of the media clip is **MCD** (media clip data), then the **D** entry specifies the data that is digested (see Table 9.9).

I.2.8 Additional-Actions

Additional-actions dictionaries (see Section 8.5.2, "Trigger Events") can be the value of the **AA** entry of a catalog, page, annotation or field dictionary. If the additional action is valid, the values of the following entries in the additional-actions dictionary are digested, in order, if present: **E**, **X**, **D**, **U**, **Fo**, **Bl**, **O**, **C**, **K**, **F**, **V**, **C**, **DC**, **WS**, **DS**, **WP**, and **DP**.

I.2.9 Page Template Verification

In some cases, the permissions granted allow page template instantiation; this occurs when the value of **P** in the **DocMDP** transform parameters dictionary is 2 or 3 (see Table 8.95) or the value of **Form** in the **UR** transform parameters is **SpawnTemplate** (see Table 8.96). In such cases, it is necessary to compute the object digest such that its value changes when new pages have been added to the document, but not when pages have been instantiated from named pages (templates).

To accomplish this, the document object digest does not include pages that have a value for the **TemplateInstantiated** entry (see Table 3.27), indicating that they are instantiated from a named page. At the time the signature is verified:

- An object digest is computed for every named page in the document.
- Using the same method, an object digest is computed for every page in the document that has a **TemplateInstantiated** entry and matched against the digest for the corresponding named page.
- Verification succeeds only if the digests match for all instantiated pages in the document.

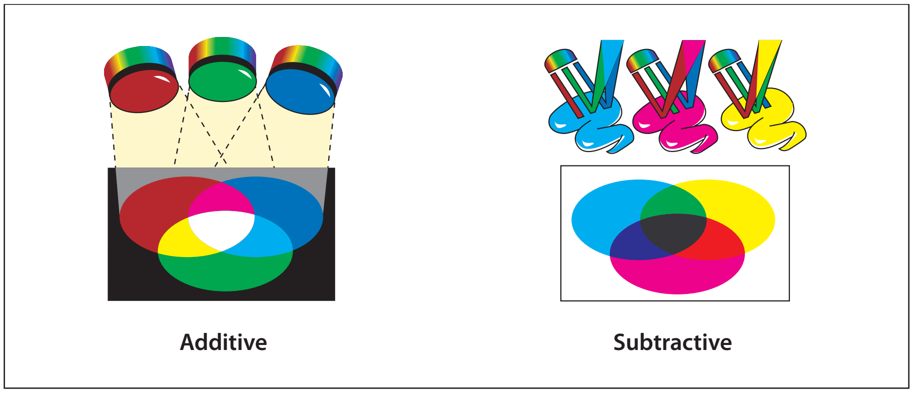


PLATE 1 *Additive and subtractive color (Section 4.5.3, "Device Color Spaces," page 211)*

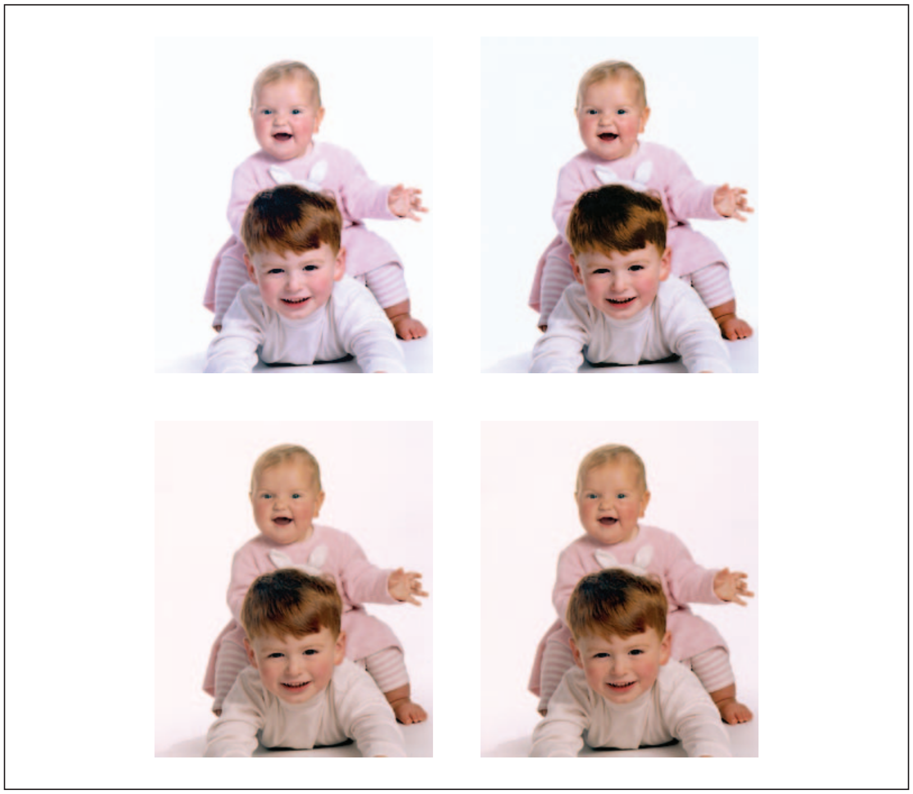


PLATE 2 *Uncalibrated color (Section 4.5.4, "CIE-Based Color Spaces," page 214)*

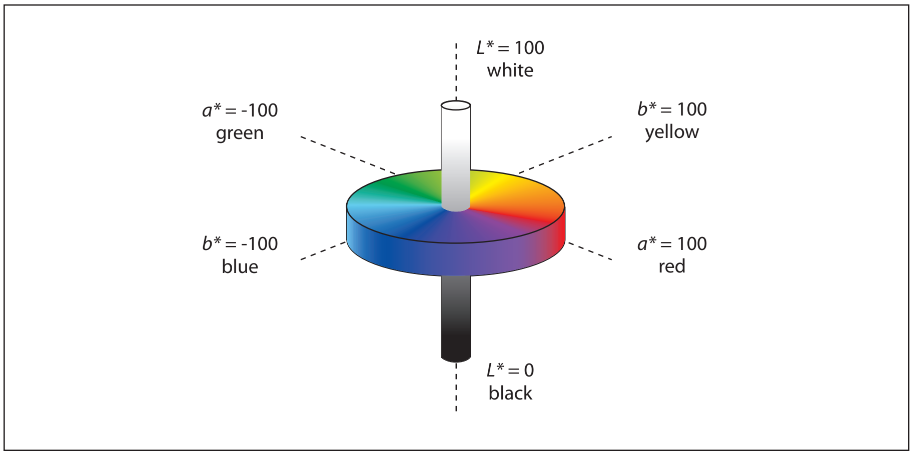


PLATE 3 ***Lab*** *color space ("Lab Color Spaces," page 220)*

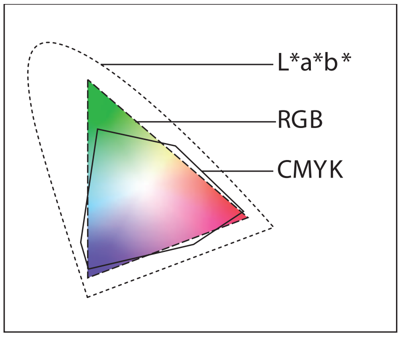


PLATE 4 *Color gamuts ("Lab Color Spaces," page 220)*

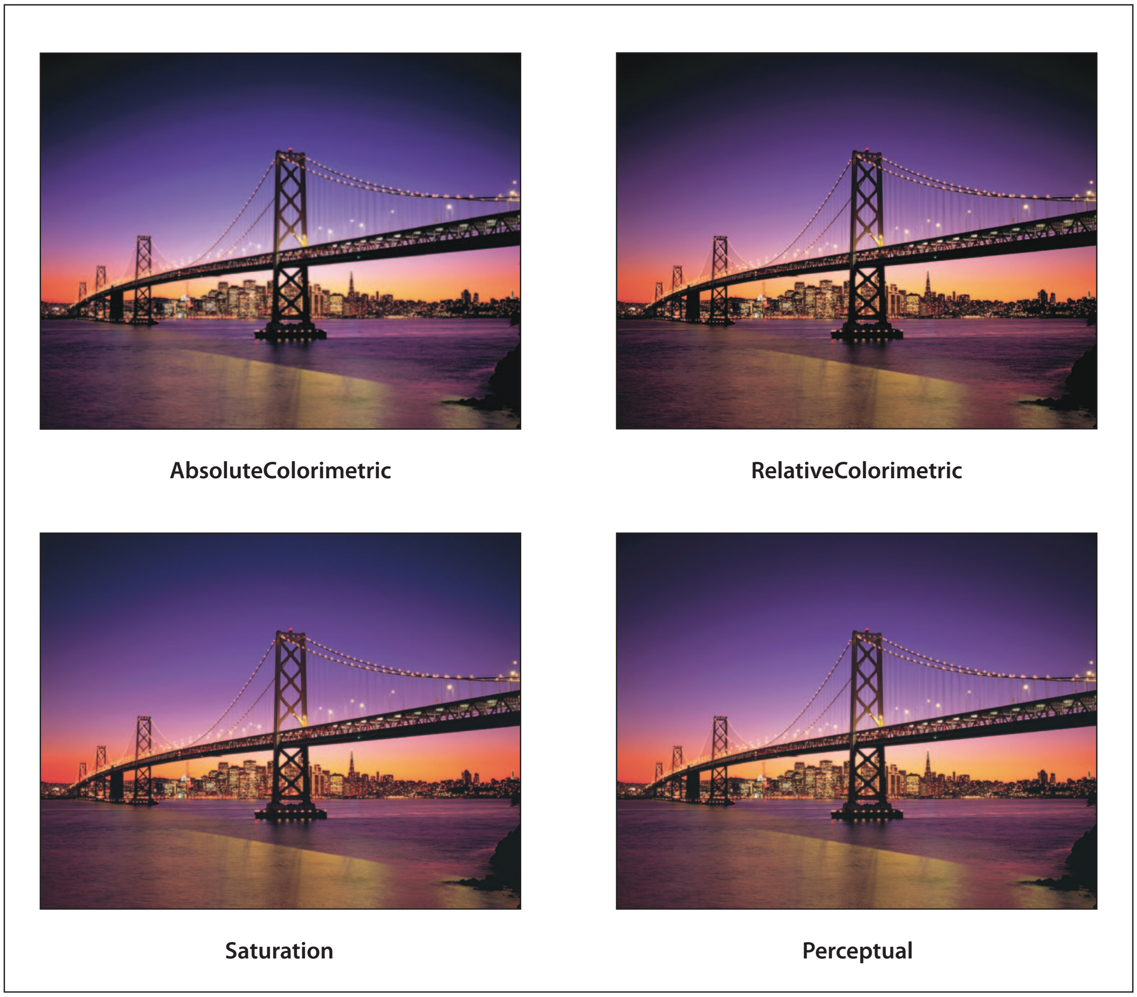


PLATE 5 *Rendering intents ("Rendering Intents," page 230)*

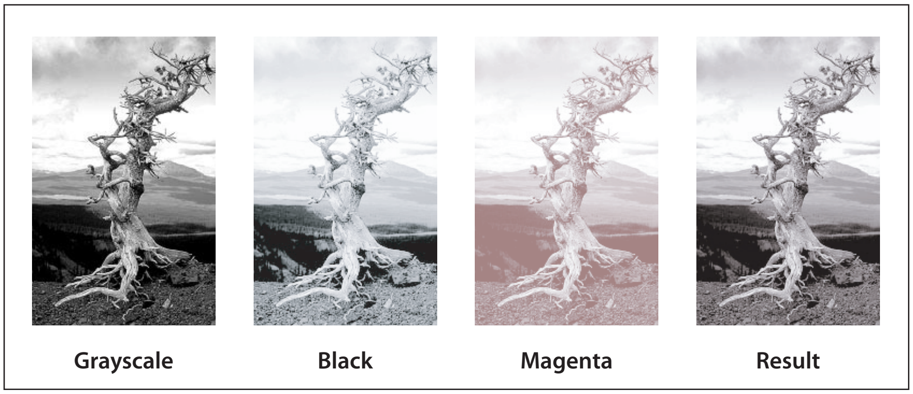


PLATE 6 *Duotone image ("DeviceN Color Spaces," page 238)*

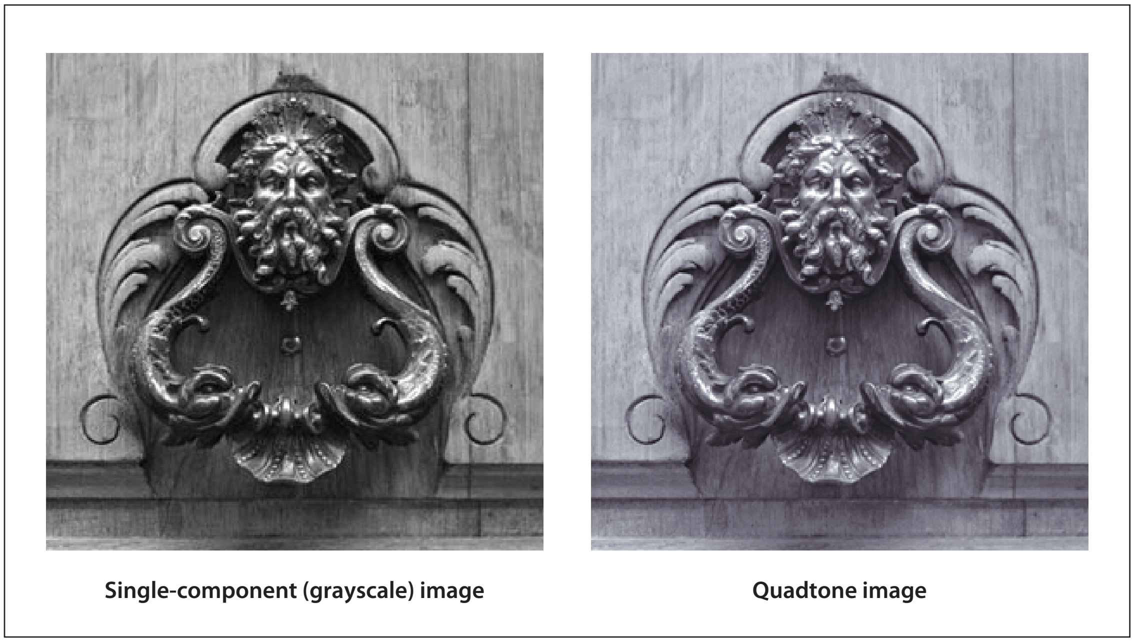


PLATE 7 *Quadtone image ("DeviceN Color Spaces," page 238)*

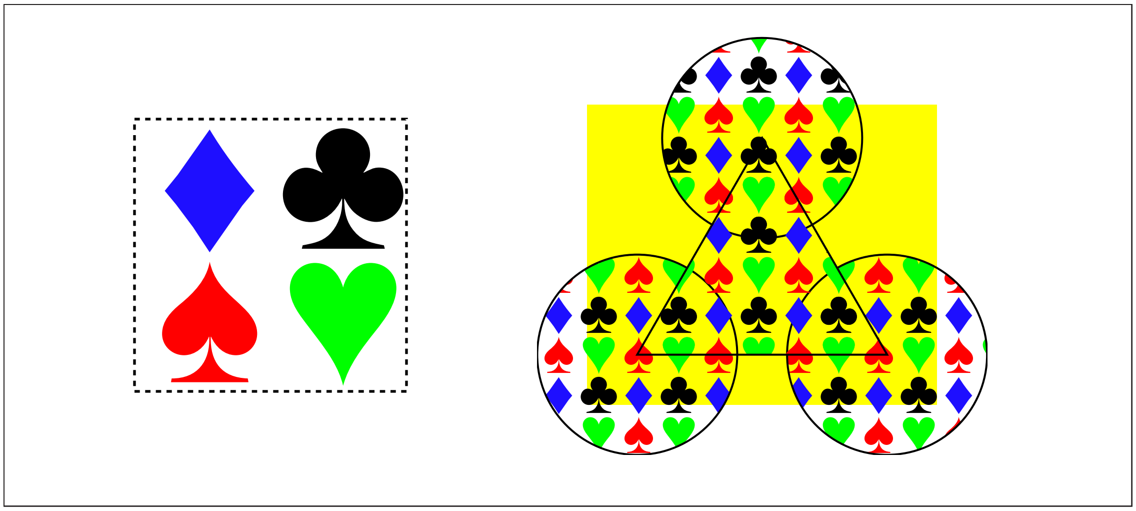


PLATE 8 *Colored tiling pattern ("Colored Tiling Patterns," page 258)*

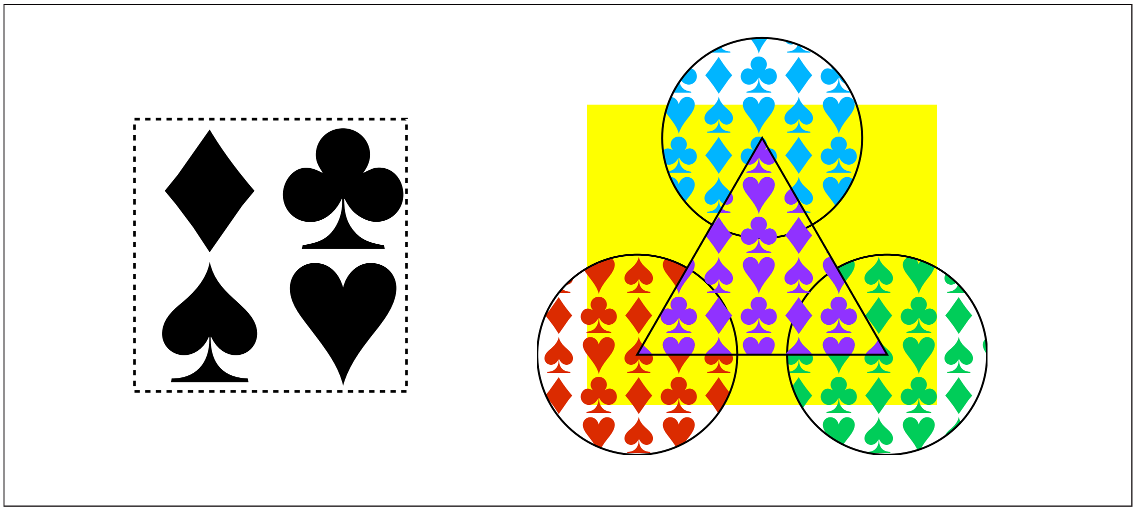


PLATE 9 *Uncolored tiling pattern ("Uncolored Tiling Patterns," page 262)*

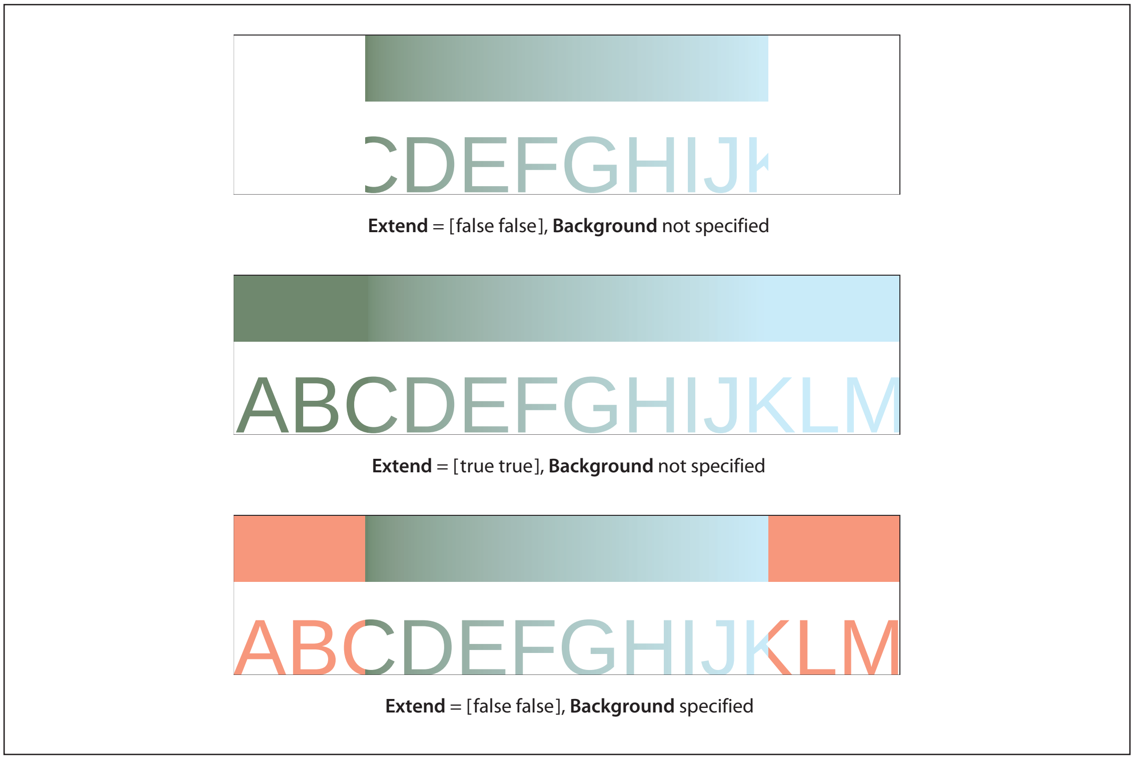

CDEFGHIJK
Extend = [false false], Background not specified
ABCDEFGHIJKLM
Extend = [true true], Background not specified
ABCDEFGHIJKLM
Extend = [false false], Background specified

PLATE 10 *Axial shading ("Type 2 (Axial) Shadings," page 273)*

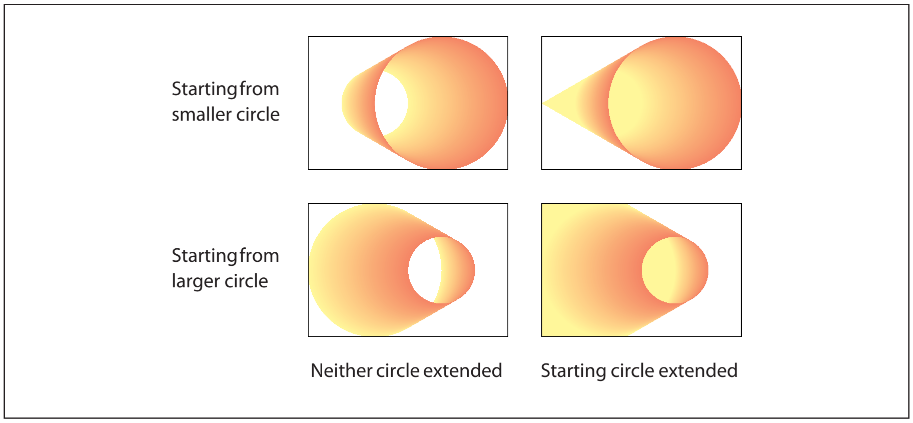


PLATE 11 *Radial shadings depicting a cone ("Type 3 (Radial) Shadings," page 275)*

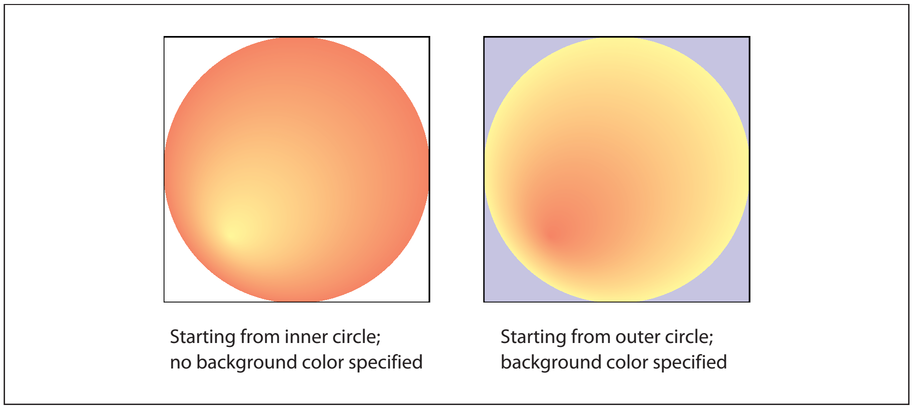

Starting from inner circle;
no background color specified
Starting from outer circle;
background color specified

PLATE 12 *Radial shadings depicting a sphere ("Type 3 (Radial) Shadings," page 275)*

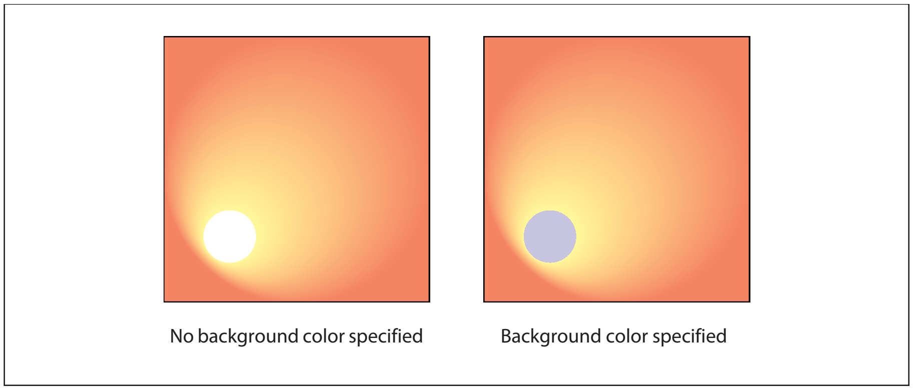


PLATE 13 *Radial shadings with extension ("Type 3 (Radial) Shadings," page 276)*

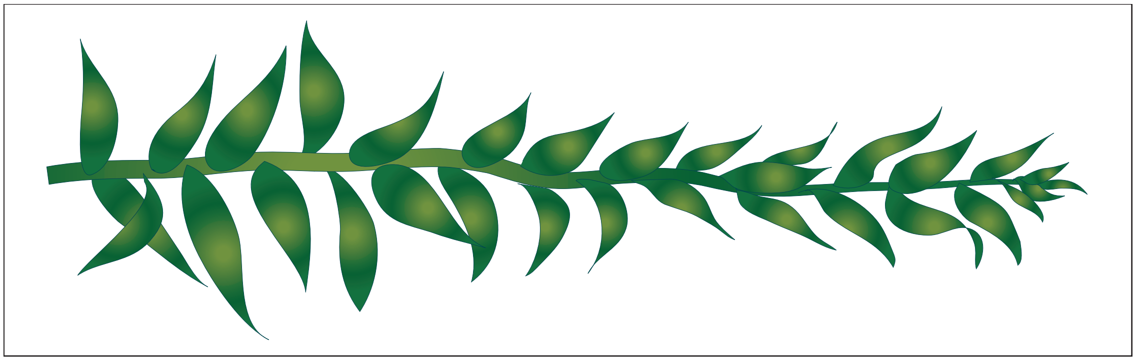


PLATE 14 *Radial shading effect ("Type 3 (Radial) Shadings," page 276)*

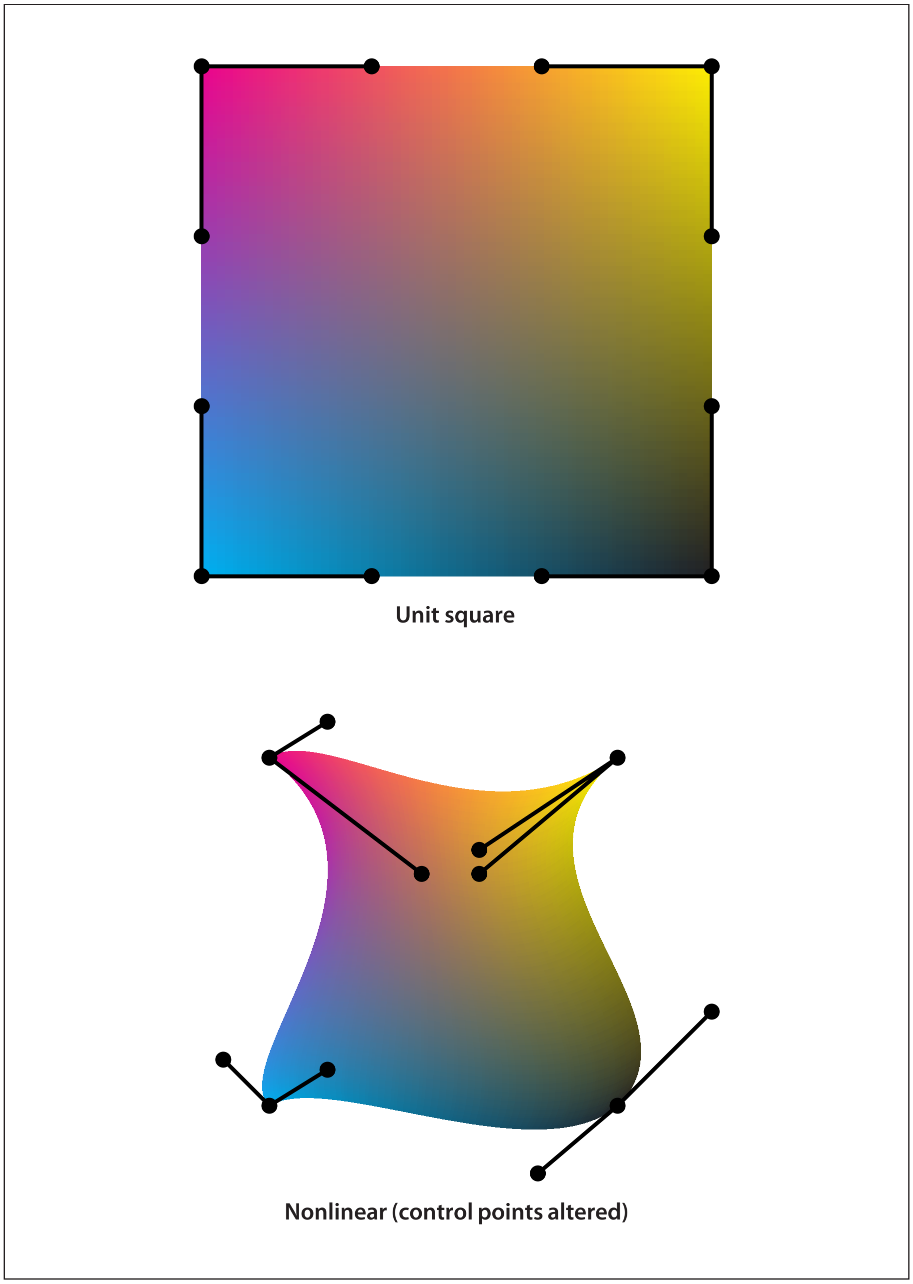


PLATE 15 *Coons patch mesh ("Type 6 Shadings (Coons Patch Meshes)," page 284)*

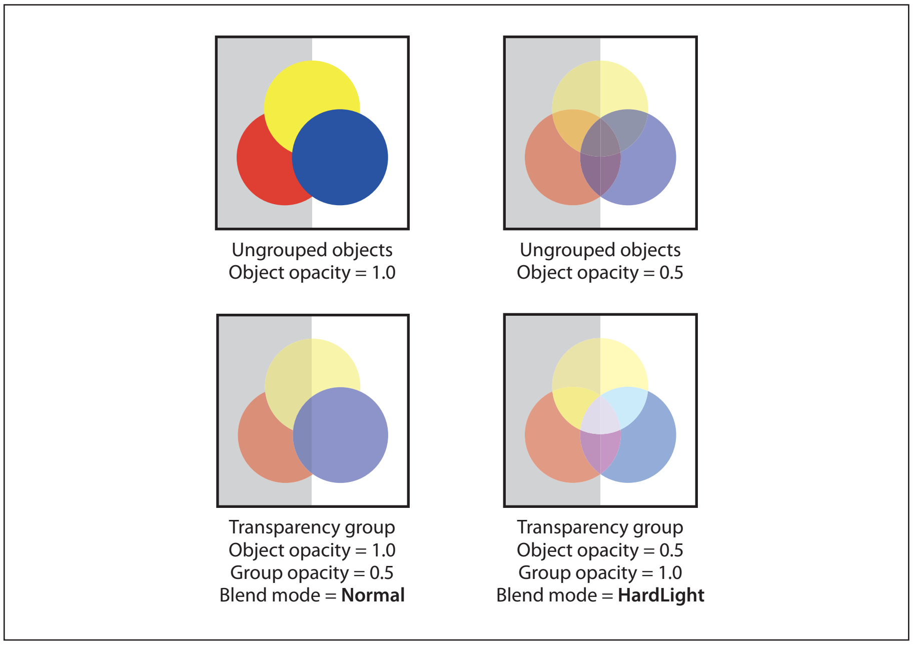


PLATE 16 *Transparency groups (Section 7.1, "Overview of Transparency," page 475)*

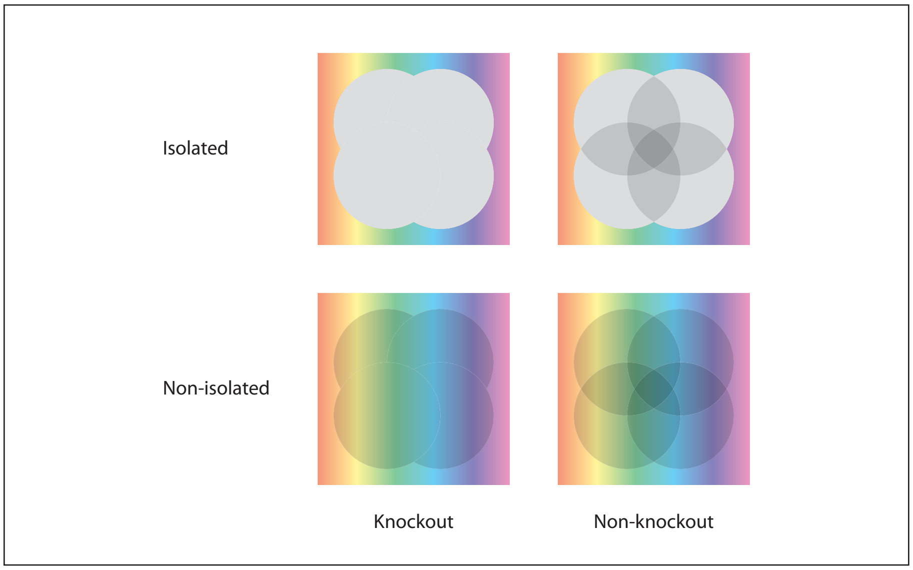
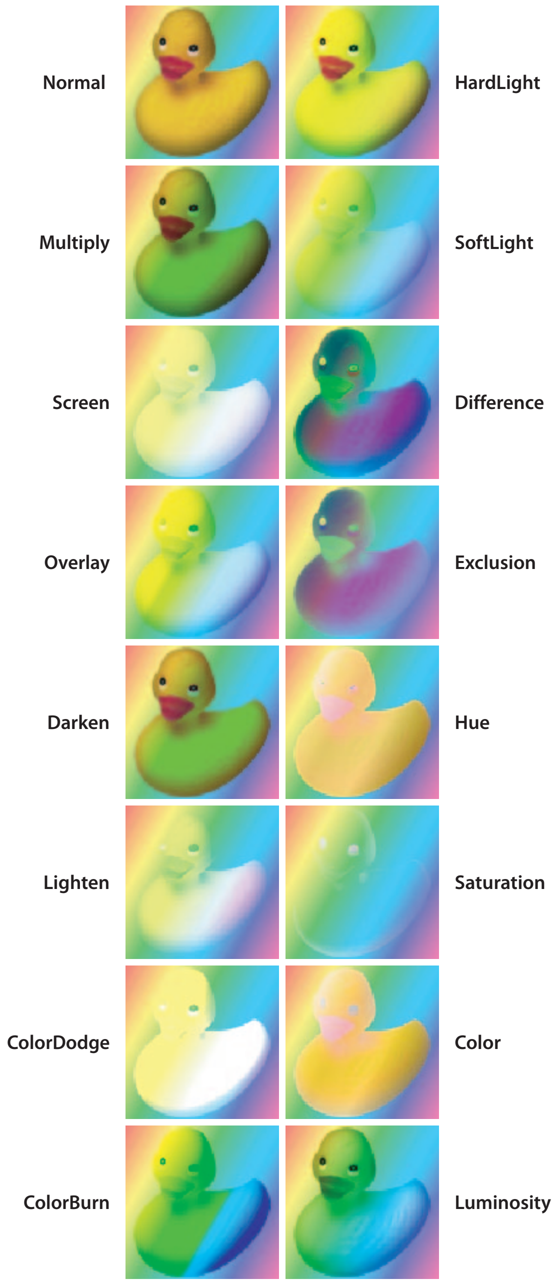
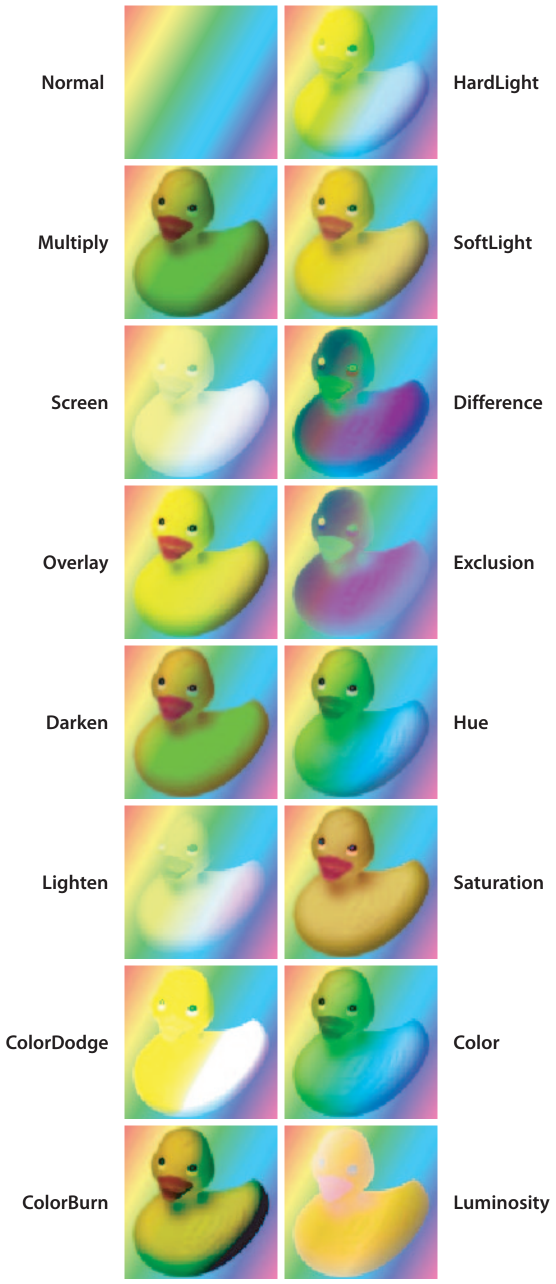


PLATE 17 *Isolated and knockout groups (Sections 7.3.4, "Isolated Groups," page 497 and 7.3.5, "Knockout Groups," page 498)*

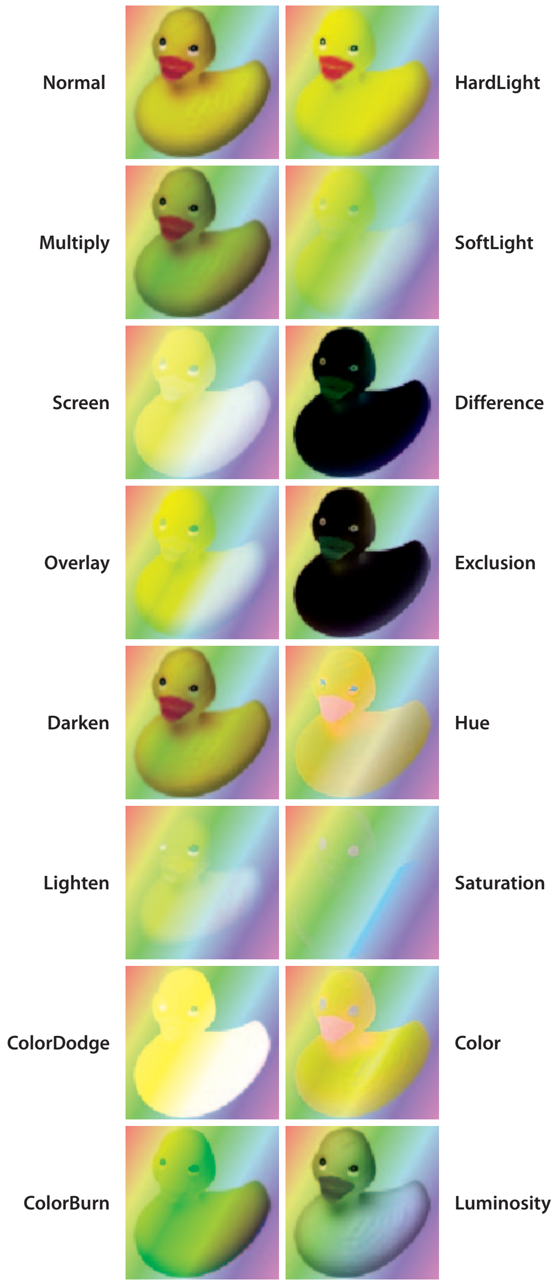


Duck in foreground, rainbow in background

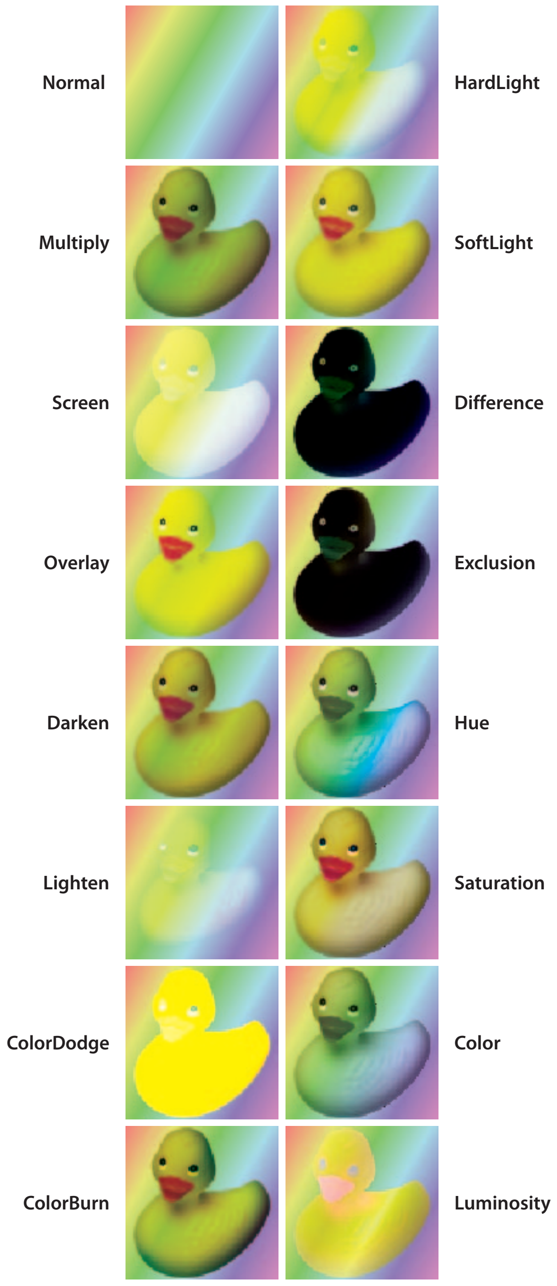


Rainbow in foreground, duck in background

PLATE 18 *RGB blend modes (Section 7.2.4, "Blend Mode," page 480)*



Duck in foreground, rainbow in background



Rainbow in foreground, duck in background

PLATE 19 *CMYK blend modes (Section 7.2.4, "Blend Mode," page 480)*

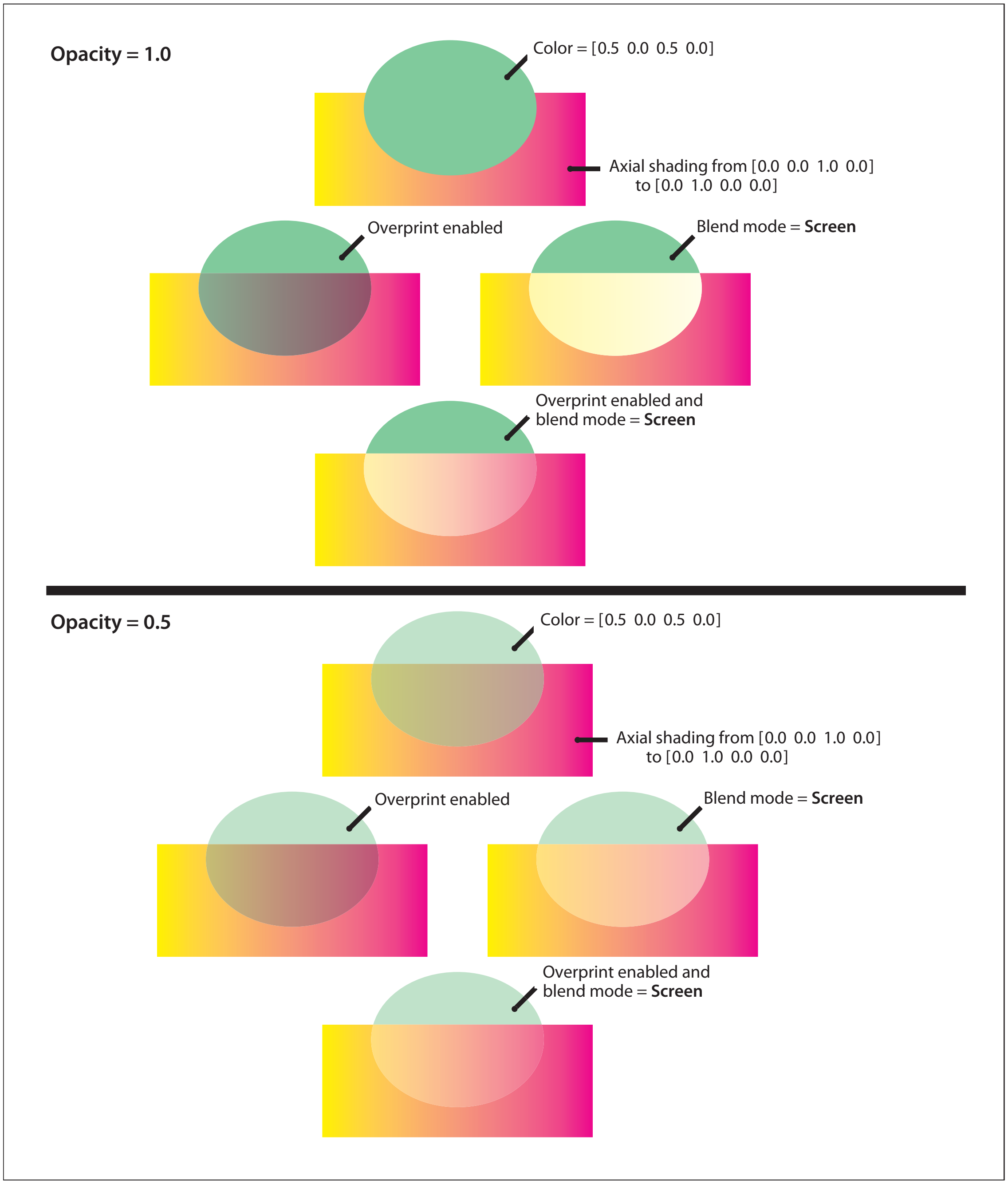

Opacity = 1.0
Color = [0.5 0.0 0.5 0.0]
Axial shading from [0.0 0.0 1.0 0.0] to [0.0 1.0 0.0 0.0]
Overprint enabled
Blend mode = **Screen**
Overprint enabled and blend mode = **Screen**
Opacity = 0.5
Color = [0.5 0.0 0.5 0.0]
Axial shading from [0.0 0.0 1.0 0.0] to [0.0 1.0 0.0 0.0]
Overprint enabled
Blend mode = **Screen**
Overprint enabled and blend mode = **Screen**

PLATE 20 *Blending and overprinting ("Compatibility with Opaque Overprinting," page 527)*

Bibliography

THIS BIBLIOGRAPHY PROVIDES details on books and documents, from both Adobe Systems and other sources, that are referred to in this book.

Resources from Adobe Systems Incorporated

All of these resources from Adobe Systems are available on the Adobe Solutions Network (ASN) site on the World Wide Web, located at

<http://partners.adobe.com/asn/>

The ASN can also be contacted as follows:

Adobe Solutions Network
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704

(800) 685-3510 (from North America)
(206) 675-6145 (from other areas)

Note: *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. Locations given are hints; the organization of the site may change over time.*

Adobe Glyph List, Version 2.0 (currently available through the document *Unicode and Glyph Names* in the Type Technology section)

Adobe Patent Clarification Notice (currently available on the Legal Notices page in the Developer Resources section)

Adobe Type 1 Font Format. Explains the internal organization of a PostScript Type 1 font program. Also see Adobe Technical Note #5015, *Type 1 Font Format Supplement*. (Currently available in the Fonts Technical Notes section of the Type Technology section.)

OPI: Open Prepress Interface Specification 1.3. Also see Adobe Technical Note #5660, *Open Prepress Interface (OPI) Specification, Version 2.0*. (Currently available through the Prepress Technical Notes page in the PostScript Developer Resources section.)

Digital Signature Appearances (currently available on Adobe PDF Technical Notes page).

PDF Signature Build Dictionary Specification for Acrobat 6.0 (currently available on Adobe PDF Technical Notes page)

PostScript Language Reference, Third Edition, Addison-Wesley, Reading, MA, 1999 (currently available on PostScript Technical Notes page)

XML Data Package Specification (currently available from the Adobe XML Architecture Specification page in the Adobe PDF section)

XML Forms Data Format Specification, Version 2.0 (currently available from the Adobe XML Architecture Specification page in the Adobe PDF section)

XMP: Extensible Metadata Platform (currently available through the XMP link in the Product and Technical Resources page)

Numbered Technical Notes:

- Technical Note #5001, *PostScript Language Document Structuring Conventions Specification, Version 3.0*
- Technical Note #5004, *Adobe Font Metrics File Format Specification, Version 4.1*

  Adobe font metrics (AFM) files are available through the Type section of the ASN Web site.
- Technical Note #5014, *Adobe CMap and CID Font Files Specification, Version 1.0*
- Technical Note #5015, *Type 1 Font Format Supplement*
- Technical Note #5044, *Color Separation Conventions for PostScript Language Programs*
- Technical Note #5078, *Adobe-Japan1-4 Character Collection for CID-Keyed Fonts*

- Technical Note #5079, *Adobe-GB1-4 Character Collection for CID-Keyed Fonts*
- Technical Note #5080, *Adobe-CNS1-4 Character Collection for CID-Keyed Fonts*
- Technical Note #5088, *Font Naming Issues*
- Technical Note #5092, *CID-Keyed Font Technology Overview*
- Technical Note #5093, *Adobe-Korea1-2 Character Collection for CID-Keyed Fonts*
- Technical Note #5094, *Adobe CJKV Character Collections and CMaps for CID-Keyed Fonts*
- Technical Note #5097, *Adobe-Japan2-0 Character Collection for CID-Keyed Fonts*
- Technical Note #5116*, Supporting the DCT Filters in PostScript Level 2*
- Technical Note #5146, *Adobe-Japan1-5 Character Collection for CID-Keyed Fonts (Addendum)*
- Technical Note #5176, *The Compact Font Format Specification*
- Technical Note #5177, *The Type 2 Charstring Format*
- Technical Note #5411, *ToUnicode Mapping File Tutorial*
- Technical Note #5432, *Acrobat JavaScript Scripting Reference, Version 6.0*
- Technical Note #5620, *Portable Job Ticket Format, Version 1.1*
- Technical Note #5660, *Open Prepress Interface (OPI) Specification, Version 2.0*

Other Resources

Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983. Includes a discussion of balanced trees.

Apple Computer, Inc., *TrueType Reference Manual*. Available on Apple's Web site at <http://developer.apple.com/fonts/TTRefMan/>.

Arvo, J. (ed.), *Graphics Gems II*, Academic Press, 1994. The section "Geometrically Continuous Cubic Bézier Curves" by Hans-Peter Seidel describes the mathematics used to smoothly join two cubic Bézier curves.

CIP4. See International Cooperation for the Integration of Processes in Prepress, Press and Postpress.

Fairchild, M. D., *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>.

Federal Information Processing Standards Publications, FIPS PUB 186-2, *Digital Signature Standard,* describes DSA signatures. It is available at <http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf>

Foley, J. D. et al., *Computer Graphics: Principles and Practice,* Addison-Wesley, Reading, MA, 1996. (First edition was Foley, J. D. 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.

Glassner, A. S. (ed.), *Graphics Gems*, Academic Press, 1993. The section "An Algorithm for Automatically Fitting Digitized Curves" by Philip J. Schneider describes an algorithm for determining the set of Bézier curves approximating an arbitrary set of user-provided points. Appendix 2 contains an implementation of the algorithm, written in the C programming language. Other sections relevant to the mathematics of Bézier curves include "Solving the Nearest-Point-On-Curve Problem" and "A Bézier Curve-Based Root-Finder," both by Philip J. Schneider, and "Some Properties of Bézier Curves" by Ronald Goldman. The source code appearing in the appendix is available via anonymous FTP, as described in the preface to *Graphics Gems III* (edited by D. Kirk; see its entry below).

Hewlett-Packard Corporation, *PANOSE Classification Metrics Guide*. Available on the Agfa Monotype Web site at <http://www.agfamonotype.com/hardware/pan1.asp>.

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, *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754-1985).

International Color Consortium (ICC). The following are available with related documents at <http://www.color.org>:

- Specification ICC.1:1998-09, *File Format for Color Profiles*, and Document ICC.1A:1999-04, *Addendum 2 to Specification ICC.1:1998-09*
- *ICC Characterization Data Registry*

International Cooperation for the Integration of Processes in Prepress, Press and Postpress (CIP4), *JDF Specification, Version 1.0*. Available through the CIP4 Web site at <http://www.cip4.org>.

International Electrotechnical Commission (IEC), IEC/3WD 61966-2.1, *Colour Measurement and Management in Multimedia Systems and Equipment, Part 2.1: Default RGB Colour Space—sRGB*. Available through Hewlett-Packard's sRGB Web site at <http://www.srgb.com>.

International Organization for Standardization (ISO). The following standards are available through <http://www.iso.ch/>:

- ISO 639, *Codes for the Representation of Names of Languages*
- ISO 3166, *Codes for the Representation of Names of Countries and Their Subdivisions*
- ISO/IEC 8824-1, *Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*
- 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 ISO group that developed the standard)
- ISO/IEC 15444-2, *Information Technology—JPEG 2000 Image Coding System: Extensions*

International Telecommunication Union (ITU). The following can be ordered from ITU at <http://www.itu.int/>

- Recommendations T.4 and T.6. These standards for Group 3 and Group 4 facsimile encoding replace those formerly provided in the CCITT *Blue Book*, Vol. VII.3.
- Recommendation X.509 (1997): *Information Technology—Open Systems Interconnection—The Directory: Authentication Framework.*

Internet Engineering Task Force (IETF) Requests for Comments (RFCs). The following RFCs are available through <http://www.rfc-editor.org>:

- RFC 1321, *The MD5 Message-Digest Algorithm*
- RFC 1738, *Uniform Resource Locators*
- RFC 1808, *Relative Uniform Resource Locators*
- RFC 1950, *ZLIB Compressed Data Format Specification, Version 3.3*
- RFC 1951, *DEFLATE Compressed Data Format Specification, Version 1.3*
- RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*
- RFC 2046, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*
- RFC 2083, *PNG (Portable Network Graphics) Specification, Version 1.0*
- RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*
- RFC 2616, *Hypertext Transfer Protocol—HTTP/1.1*
- RFC 3066, *Tags for the Identification of Languages*
- RFC 3174, *US Secure Hash Algorithm 1 (SHA1)*
- RFC 3280, *Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile*

Internet Engineering Task Force (IETF) Public Key Infrastructure (PKIX) working group: <http://www.ietf.org/html.charters/pkix-charter.html>

Kirk, D. (ed.), *Graphics Gems III*, Academic Press, 1994. The section "Interpolation Using Bézier Curves" by Gershon Elber contains an algorithm for calculating a Bézier curve that passes through a user-specified set of points. The algorithm uses not only cubic Bézier curves, which are supported in PDF, but also higher-order Bézier curves. The appendix contains an implementation of the algorithm, written in the C programming language. The source code appearing in the appendix is available via anonymous FTP, as described in the book's preface.

Lunde, K., *CJKV Information Processing*, O'Reilly & Associates, Sebastopol, CA, 1999. Excellent background material on CMaps, character sets, encodings, and the like.

Microsoft Corporation, *TrueType 1.0 Font Files Technical Specification*. Available at <http://www.microsoft.com/typography/tt/tt.htm>.

Netscape Communications Corporation, *Client-Side JavaScript Reference*. Available through Netscape's developer site at <http://developer.netscape.com>.

Pennebaker, W. B. and Mitchell, J. L., *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1992.

Porter, T. and Duff, T., "Compositing Digital Images," *Computer Graphics*, Vol. 18 No. 3, July 1984. *Computer Graphics* is the newsletter of the ACM's special interest group SIGGRAPH; for more information, see <http://www.acm.org>.

RSA Security, Inc. The following documents, among others related to encryption and digital signatures, are available at <http://www.rsasecurity.com>:

- *PKCS #1 - RSA Cryptography Standard*
  <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html>
- *PKCS #7 - Cryptographic Message Syntax Standard*
  <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html>

Unicode Consortium publications:

- *The Unicode Standard, Version 4.0*, Addison-Wesley, Reading, MA, 2003. The latest information is available at <http://www.unicode.org>.
- Unicode Standard Annex #9, *The Bidirectional Algorithm, Version 4.0.0*, Unicode Standard Annex #14, *Line Breaking Properties, Version 4.0.0*, and Unicode Standard Annex #29, *Text Boundaries, Version 4.0.0*. These technical reports are available at <http://www.unicode.org>.

World Wide Web Consortium (W3C). The following publications are available through the W3C Web site at <http://www.w3.org/>:

- *Cascading Style Sheets, level 2 (CSS2) Specification*
  <http://www.w3.org/TR/REC-CSS2/>
- *Extensible Markup Language (XML) 1.1*
  <http://www.w3.org/TR/xml11/>
- *Extensible Stylesheet Language (XSL) 1.0*
  <http://www.w3.org/TR/xsl/>

- *HTML 4.01 Specification*
  <http://www.w3.org/TR/html401/>
- *Scalable Vector Graphics (SVG) 1.0 Specification*
  <http://www.w3.org/TR/2001/REC-SVG-20010904/>
- *Synchronized Multimedia Integration Language (SMIL 2.0)*
  <http//:www.w3.org/TR/smil20/>
- *Web Content Accessibility Guidelines 1.0*
  <http://www.w3.org/TR/WAI-WEBCONTENT/>
- *XHTML 1.0: The Extensible HyperText Markup Language*
  <http://www.w3.org/TR/xhtml1/>

Index

Page references in **boldface** mark principal or defining occurrences of a topic.

Colophon

THIS DOCUMENT WAS PRODUCED using Adobe® FrameMaker®, Adobe Illustrator®, Adobe Photoshop®, Adobe Acrobat® Distiller®, and other application software packages that support the PostScript® page description language and Type 1 fonts. The type used is from the Adobe Minion® Pro and Myriad® Pro families. Heads are set in Myriad Pro Semibold and the body text is set in 10.5-on-13-point Minion Pro.

Authors—Jim Meehan, Ed Taft, Stephen Chernicoff, Caroline Rose, Ron Karr

Key Contributors—Nabeel Al-Shamma, Steven Kelley Amerige, Bob Ayers, Tim Bienz, Krish Chaudhury, Richard Cohn, Michelle Dalton, Stephen Deach, Jon Ferraiolo, Matt Foley, Martin Fox, Ron Gentile, John Green, Jim King, Bennett Leeds, Pierre Louveaux, Teryk Morris, Carl Orthlieb, Mike Ossesia, Ajay Pande, Roberto Perelman, Scott Petersen, Jim Pravetz, Dan Rabin, Loretta Guarino Reid, Paul Rovner, Ed Rowe, Craig Rublee, Mike Schuster, Steve Schiller, John Warnock, Bob Wulff, Steve Zilles

Reviewers—Parviz Banki, Xintai Chang, L. Peter Deutsch, Mark Donohoe, David Gelphman, Brian Havlin, Raph Levien, Ken Lunde, Eric Muller, John Nash, Terry O'Donnell, Jason Pittenger, Dick Sites, Lydia Stang, Koichi Yoshimura, and numerous others at Adobe Systems and elsewhere

Editing and Book Production—Stephen Chernicoff, Caroline Rose, Ron Karr

Index—Stephen Chernicoff, Ron Karr, Lucie Haskins

Illustrations—Kim Arney, Wendy Bell, Peter Constable, Lisa Ferdinandsen, Carol Keller, Pierre Louveaux, Jim Meehan, Dayna Porterfield, Carl Yoshihara

Book Design—Sharon Anderson

Publication Management—Robert Morrish, Courtney Attwood