Portable Document Format Reference Manual

Version 1.2

Adobe Systems Incorporated

November 27, 1996

Tim Bienz, Richard Cohn, and James R. Meehan
Adobe Systems Incorporated

Library of Congress Cataloging-in-Publication Data

Portable document format reference manual / Adobe Systems Incorporated.
p. cm.
Includes bibliographical references (p. 207) and index.
ISBN 0–201–62628–4
1. File organization (Computer science) 2. PostScript (Computer program language)
3. Text processing (Computer science) I. Adobe Systems.
QA76.9.F5P67 199393–8046
005.74—dc20CIP

1 2 3 4 5 6 7 8 9–MA–9796959493

Contents

Figures

Tables

Examples

CHAPTER 1

Introduction

This book describes the Portable Document Format (PDF), the native file format of the Adobe® Acrobat® family of products. The goal of these products is to enable users to easily and reliably exchange and view electronic documents independent of the environment in which they were created. PDF relies on the imaging model of the PostScript® language to describe text and graphics in a device- 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.

PDF files are built from a sequence of numbered objects similar to those used in the PostScript language. The text, graphics, and images that make up the contents of a page are represented using operators that are based on those in the PostScript language and that closely follow the Adobe Illustrator® 3.0 page description operators.

A PDF file is not a PostScript language program and cannot be directly interpreted by a PostScript interpreter. However, the page descriptions in a PDF file can be converted into a PostScript language program.

1.1 About this book

This book provides a description of the PDF file format, as well as suggestions for producing efficient PDF files. It is intended primarily for application developers who wish to produce PDF files directly. This book also contains enough information to allow developers to write applications that read and modify PDF files. While PDF is independent of any particular application, occasionally PDF features are best explained by the actions a particular application takes when it encounters that feature in a file. Similarly, Appendix D discusses some implementation limits in the Acrobat viewer applications, even though these limits are not part of the file format itself.

This book consists of two sections. The first section describes the file format and the second lists techniques for producing efficient PDF files. In addition, appendices provide example files, detailed descriptions of several predefined font encodings, and a summary of PDF page marking operators.

Readers are assumed to have some knowledge of the PostScript language, as described in the *PostScript Language Reference Manual, Second Edition* [1]. In addition, some understanding of fonts, as described in the *Adobe Type 1 Font Format* [6], is useful.

The first section of this book, Portable Document Format, includes Chapters 2 through 7 and describes the PDF file format.

Chapter 2 describes the motivation for creating the PDF file format and provides an overview of its architecture. PDF is compared to the PostScript language.

Chapter 3 discusses the coordinate systems and transformations used in PDF files. Because the coordinate systems used in PDF are very much like those used in the PostScript language, users with substantial background in the PostScript language may wish to read this chapter only as a review.

Chapter 4 describes the types of objects used to construct documents in PDF files. These types are similar to those used in the PostScript language. Readers familiar with the types of objects present in the PostScript language may wish to read this chapter quickly as a reminder.

Chapter 5 provides a description of the format of PDF files, how they are organized on disk, and the mechanism by which updates can be appended to a PDF file.

Chapter 6 describes the way that a document is represented in a PDF file, using the object types presented in Chapter 4.

Chapter 7 defines the resources used in a PDF file, including fonts, color spaces, images, and others.

Chapter 8 discusses the page marking operators used in PDF files. These are the operators that actually make marks on a page. Many are similar to one or more PostScript language operators. Readers with PostScript language experience will quickly see the similarities.

The second section of this book, Optimizing PDF Files, includes Chapters 10 through 14 and describes techniques for producing efficient PDF files. Many of the techniques presented can also be used in the PostScript language. The techniques are broken down into four areas: text, graphics, images, and general techniques.

Chapter 10 discusses general optimizations that may be used in a wide variety of situations in PDF files.

Chapter 11 discusses optimizations for text.

Chapter 12 discusses graphics optimizations.

Chapter 13 discusses optimizations that may be used on sampled images.

Finally, Chapter 14 contains techniques for using clipping paths to restrict the region in which drawing occurs and a technique using images to make efficient blends.

1.2 Introduction to the Second Edition—PDF 1.1

The second edition of this manual is a revision of the 1993 edition of *Portable Document Format Reference Manual*. It describes version 1.1 of the Portable Document Format.

The PDF specification is independent of any particular implementation of a PDF generator or consumer. To provide guidance to implementors, however, *Implementation Notes* that accompany the specification and Appendix G describe the behavior of Acrobat viewers (versions 1.0, 2.0, and 2.1) when they encounter the changes documented herein.

Implementation note — *PDF 1.1 is the native file format of the Adobe Acrobat 2.0 family of products.*

The PDF 1.1 specification, like the PDF 1.0 specification, defines a minimum interchange level of functionality. The Portable Document Format is an extensible format, which means that PDF files may contain objects not defined by this specification. *Consumers*, applications that read PDF files and interpret their contents, are expected to implement correctly the semantics of objects that are specified by PDF 1.1 and, as gracefully as possible, to ignore any objects that they do not understand. Appendix G provides guidance on how a consumer should handle objects it does not understand.

Implementation note — *Some Acrobat 2.0 and subsequent products provide an interface that supports plug-ins. These plug-ins can use and/or put private data objects within a PDF file. Appendix G indicates the kinds of private data that can be used and Appendix F defines a registry for this data. The registry can be used to avoid conflicts in identifying data from independent plug-ins.*

New features introduced in PDF 1.1 include the following:

- The ability to protect a document with a password and to restrict operations on a document.
- The ability to tie blocks of text together into “articles,” making reading easier.
- The generalization of link and bookmark destinations to “actions,” which include links to other PDF files and foreign files.
- The ability to define new annotation types and to provide additional attributes for existing types.
- The ability to specify default settings and actions when a document is opened.
- Device-independent color.
- An ID included in files to make it easier to verify that a file is the correct file, even under circumstances where the file’s name is incorrect (such as files on some networks).
- A binary option that allows files to be smaller.
- A new date format that allows programmatic comparison of dates.

- The ability to provide additional document information.

Note *In PDF 1.1, dictionary key names are often one or two letters in order to conserve space in files. When these keys are described below, they are followed in parentheses by a more descriptive string. However, only the actual one- or two-letter name may be used in a PDF file.*

1.3 Introduction to the Third Edition—PDF 1.2

This document is a revision of the March 1, 1996, edition of *Portable Document Format Reference Manual*. It describes version 1.2 of the Portable Document Format.

Implementation note *PDF 1.2 is the native file format of the Adobe Acrobat 3.0 family of products.*

New features introduced in PDF 1.2 include the following:

- Interactive elements with state, such as radio buttons and checkboxes.
- Support for playing movies (from external files) and sounds (either embedded in the PDF file or from external files).
- Interactive, fill-in forms, with PDF-based format for data that can be imported, exported, transmitted, and received from the Web.
- Support for Chinese, Korean, and Japanese text.
- Enhanced borders, highlights, and fully general appearances for annotations.
- Support for responding to mouse-events.
- Essentially unlimited number of hypertext links.
- Support for OPI (Open Prepress Interface).
- Advanced color features, such as halftone screens, transfer functions, patterns, and separation color spaces.

Note *PDF is an evolving language, and new editions of this manual will be offered on an ongoing basis to document the changes. The most recent version will be available on the Adobe's Web site.*

While many readers will use a printed copy of this manual, others will refer to it online. With that in mind, many changes have been made to the formatting, such as the choice and size of fonts, to make on-screen reading somewhat easier.

1.4 Conventions used in this book

Text styles are used to identify various operators, keywords, terms, and objects. Five formatting styles are used in this book:

- PostScript language operators, PDF operators, PDF keywords, the names of keys in dictionaries, and other predefined names are written in boldface. Examples are **moveto**, **Tf**, **stream**, **Type**, and **MacRomanEncoding**.

- Operands of PDF operators are written in an italic sans serif font. An example is *linewidth*.

- Object types are written with initial capital letters. An example is FontDescriptor.

- The first occurrence of terms and the boolean values *true* and *false* are written in italics. This style is also used for emphasis.

- Samples of code as it would appear in a PDF file are written in a monospaced font. An example is `/MediaBox [0 0 612 792]`.

Tables containing dictionary keys are normally organized with the **Type** and **Subtype** keys first, followed by any other keys that are required in the dictionary, followed by any optional keys.

Important changes and corrections to the previous edition of this manual are marked with change bars in the left margin. Most of the changes are related to the differences between versions of PDF. Those changes are marked with icons in the right margin:

This marks a section that specifically deals with version 1.0, normally indicating a feature that has been superseded in a later version. **PDF 1.0**

This marks a section that specifically deals with new features in version 1.1. **PDF 1.1**

This marks a section that specifically deals with new features in version 1.2. **PDF 1.2**

Generally, new features and attributes are ignored by older viewers. An attribute marked with a version icon may be new with that version, or may have been substantially redefined in that version.

1.5 A note on syntax

Throughout this book, Backus–Naur form (BNF) notation is used to describe syntax:

```
<xyz> ::=    abc <def> ghi |
             <k> j
```

A token enclosed in angle brackets names a class of document component, while plain text appears verbatim or with some obvious substitution. The grammar rules have two parts. The name of a class of component is on the left of the definition

symbol (::=). In the example above, the class is *xyz*. On the right of the definition symbol is a set of one or more alternative forms that the class component might take in the document. A vertical bar (|) separates alternative forms.

The right side of the definition may be on one or more lines. With only a few exceptions, these lines do not correspond to lines in the file.

The notation { ... } means that the items enclosed in braces are optional. If an asterisk follows the braces, the objects inside the braces may be repeated *zero* or more times. The notation < ... >+ means that the items enclosed within the brackets must be repeated *one* or more times.

When an operator appears in a BNF specification, it is shorthand for the operator plus its operands. For example, when the operator **m** appears in a BNF specification, it means *x y* **m**, where *x* and *y* are numbers.

Note that PDF is case-sensitive. Uppercase and lowercase letters are distinct.

1.6 Copyrights and permissions to use PDF

The general idea of utilizing an interchange format for final-form documents is in the public domain. Anyone is free to devise his or her own set of unique commands and data structures that define an interchange format for final-form documents. Adobe owns the copyright in the data structures, operators, and the written specification for the particular interchange format called the Portable Document Format. These elements 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 as a standard. This enables the public to distinguish between the Portable Document Format and other interchange formats for final-form documents.

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

- Prepare files in which the file 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 the results, prints the results, or otherwise interprets a file represented in the Portable Document Format.
- Copy Adobe's copyrighted list of operators and data structures, as well as the PDF sample code and PostScript language Function definitions in the written specification, to the extent necessary to use the Portable Document Format for the above purposes.

The only condition on such permission is that anyone who uses the copyrighted list of operators and data structures in this way must include an appropriate copyright notice.

This limited right to use the copyrighted list of operators and data structures does not include the right to copy the *Portable Document Format Reference Manual*, other copyrighted material from Adobe, or the software in any of Adobe's products which use the Portable Document Format, in whole or in part.

Section I

Portable Document Format

CHAPTER 2

Overview

Before examining the detailed structure of a PDF file, it is important to understand what PDF is and how it relates to the PostScript language. This chapter discusses PDF and its relationship to the PostScript language.

Chapter 3 discusses the coordinate systems used to describe various components of a PDF file. Chapters 4 and 5 discuss the basic types of objects supported by PDF and the structure of a PDF file. Chapters 6 and 8 describe the structure of a PDF document and the operators used to draw text, graphics, and images.

2.1 What is the Portable Document Format?

PDF is a file format used to represent a document in a manner independent of the application software, hardware, and operating system used to create it. A *PDF file* contains a *PDF document* and other supporting data.

A PDF document contains one or more pages. Each page in the document may contain any combination of text, graphics, and images in a device- and resolution-independent format. This is the *page description*. A PDF document may also contain information possible only in an electronic representation, such as hypertext links.

In addition to a document, a PDF file contains the version of the PDF specification used in the file and information about the location of important structures in the file.

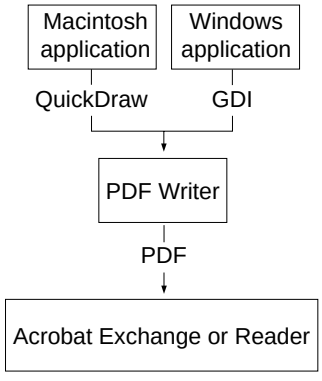
2.2 Using PDF

To understand PDF, it is important to understand how PDF documents will be produced and used. As PDF documents and applications that read PDF files become more prevalent, new ways of creating and using PDF files will be invented. This is one of the goals of this book—to make the file format accessible so that application developers can expand on the ideas behind PDF and the applications that initially support it.

Currently, PDF files may be produced either directly from applications or from files containing PostScript page descriptions.

Many applications can produce PDF files directly. The PDF Writer, available on both Apple® Macintosh® computers and computers running the Microsoft® Windows® environment, acts as a printer driver. A printer driver normally converts operating system graphics and text commands (QuickDraw™ for the Macintosh and GDI for Windows) into commands understood by a printer. The driver embeds these commands in a stream of commands sent to a printer that results in a page being printed. Instead of sending these commands to a printer, the PDF Writer converts them to PDF operators and embeds them in a PDF file, as shown in Figure 2.1.

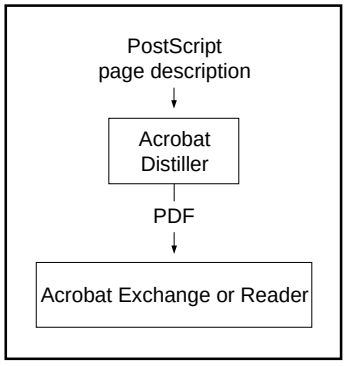
Figure 2.1 *Creating PDF files using PDF Writer*



The resulting PDF files are platform-independent. Regardless of whether they were generated on a Macintosh or Windows computer, they may be viewed by a PDF viewing application on any platform.

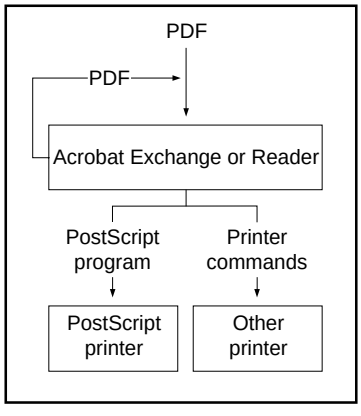
Some applications produce PostScript page descriptions directly because of limitations in the QuickDraw or GDI imaging models or because they run on DOS or UNIX® computers, where there is no system-level printer driver. For these applications, PostScript page descriptions can be converted into PDF files using the Acrobat Distiller® application, as shown in Figure 2.2. The Distiller application accepts any PostScript page description, whether created by a program or hand-coded by a human. The Distiller application produces more efficient PDF files than PDF Writer for some application programs.

Figure 2.2 *Creating PDF files using the Distiller program*



Once a PDF file has been created, Acrobat Exchange™ or Acrobat Reader can be used to view and print the document contained in the file, as shown in Figure 2.3. Users can navigate through the document using thumbnail sketches, hypertext links, and bookmarks. The document's text may be searched and extracted for use in other applications. In addition, an Acrobat Exchange user may modify a PDF document by creating text annotations, hypertext links, thumbnail sketches of each page, and bookmarks that directly access views of specific pages.

Figure 2.3 *Viewing and printing a PDF document*



2.3 General properties

Given the goals and intended use of PDF, its design has several notable properties. This section describes those properties.

2.3.1 PostScript language imaging model

PDF represents text and graphics using the imaging model of the PostScript language. Like a PostScript language program, a PDF page description draws a page by placing "paint" on selected areas.

- The painted figures may be letter shapes, regions defined by combinations of lines and curves, or sampled images such as digitally sampled representations of photographs.
- The paint may be any color.
- Any figure can be clipped to another shape, so that only portions of the figure within the shape appear on the page.
- When a page description begins, the page is completely blank. Various operators in the page description place marks on the page. Each new mark completely obscures any marks it may overlay.

The PDF page marking operators are similar to the marking operators in the PostScript language. The main reason that the PDF marking operators differ from the PostScript language marking operators is that PDF is not a programming language and does not contain procedures, variables, or control constructs. PDF trades reduced flexibility for improved efficiency. A typical PostScript language program defines a set of high-level operators using the PostScript language marking operators. PDF defines its own set of high-level operators that is sufficient for describing most pages. Because these operators are implemented directly in machine code rather than PostScript language code, PDF page descriptions can be drawn more quickly. Because arbitrary programming constructs are not permitted, applications can more efficiently and reliably locate text strings in a PDF document.

2.3.2 Portability

PDF 1.1

A PDF file is either a 7-bit ASCII file or a binary file. If it is a 7-bit ASCII file, only the printable subset of the 7-bit ASCII code plus space, tab, carriage return, and linefeed are used. If it is a binary file, the entire 8-bit range of characters may be used.

ASCII is the most portable form, since it is the only form that will fit through channels that are not 8-bit clean or are subject to end-of-line translation, etc. A binary file simply cannot be transported in such cases.

Unfortunately, some agents, when presented with information labelled as “text,” take unreasonable liberties with the contents. For example, mail transmission systems may not preserve certain 7-bit characters and may change line endings. This can cause damage to PDF files.

Therefore, in situations where it is possible to label PDF files as “binary,” we recommend that this be done. One method for encouraging such treatment is to include a few binary characters (codes greater than 127) in a comment near the beginning of the file, as described in Section 5.1 on page 61, *even if the rest of the file is ASCII*. This ensures that a PDF file will be treated as binary when this is possible, while still allowing it to be transferred through a non-binary channel without damage.

Implementation note *Acrobat applications produce PDF files with a comment that includes binary characters.*

2.3.3 Compression

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

- JPEG compression of color and grayscale images
- CCITT Group 3, CCITT Group 4, LZW (Lempel-Ziv-Welch), and Run Length compression of monochrome images
- LZW and Flate compression of text, graphics, and indexed image data. **PDF 1.2**

Using JPEG compression, color and grayscale images can be compressed by a factor of 10 or more. Effective compression of monochrome images depends upon the compression filter used and the properties of the image, but reductions of 2:1 to 8:1 are common. LZW compression of text and graphics comprising the balance of the document results in compression ratios of approximately 2:1. All of these compression filters produce binary data, which may then be encoded in the ASCII base-85 encoding to maintain portability.

2.3.4 Font independence

Managing fonts is a fundamental challenge in document exchange. Generally, the receiver of a document must have the same fonts the sender used to create the document. Otherwise, a default font is substituted, producing unexpected and undesirable effects because the default font has different character metrics (widths) than the intended font. The sender could include the fonts with the document, but this can easily make even a short document quite large—a typical two-page memo using four fonts might grow from 10K to 250K. Another possibility is that the sender could convert each page of the document to a fixed-resolution image like a facsimile. Even when compressed, however, the image of a single page can be quite large (45–60K when sampled at 200 dpi). In addition, there is no intelligence left in the file, preventing the receiver from searching for or extracting text from the document.

PDF provides a new solution that makes a document independent of the fonts used to create it. A PDF file contains a *font descriptor* for each font used in a document. The font descriptor includes the font name, character metrics, and style information. This is the information needed to simulate missing fonts and is typically only 1–2K per font.

If a font used in a document is available on the computer where the document is viewed, it is used. If it is not available, a multiple master font is used to simulate on a character-by-character basis the weight and width of the original font, to maintain the overall “color” and formatting of the document. This solution applies to both Adobe Type 1 fonts and fonts in the TrueType™ format [20] developed by Apple Computer, Inc.

S*ymbolic fonts* must be handled in a special way. A symbolic font is any font that does not use the standard ISOLatin1 character set. Fonts such as Carta®, Adobe Caslon Swash Italic, Minion™ Ornaments, and Lucida® Math fall into this category. It is not possible to simulate a symbolic font effectively.

For symbolic fonts, a font descriptor (including metrics and style information) is not sufficient; the actual character shapes (or glyphs) are required to accurately display and print the document. For all symbolic fonts other than Symbol and ITC Zapf Dingbats®, a compressed version of the Type 1 font program for the font is included in the PDF file. Symbol and ITC Zapf Dingbats, the most widely used symbolic fonts, ship with Acrobat Exchange and Acrobat Reader and do not need to be included in a PDF file.

2.3.5 Single-pass file generation

Because of system limitations and efficiency considerations, it may be desirable or necessary for an implementation of a program that produces PDF such as the PDF Writer to create a PDF file in a single pass. This may be, for example, because the application has access to limited memory or is unable to open temporary files. For this reason, PDF supports single-pass generation of files. While PDF requires certain objects to contain a number specifying their length in bytes, a mechanism is provided allowing the length to be located in the file after the object. In addition, information such as the number of pages in the document can be written into the file after all pages have been written into the file.

2.3.6 Random access

Tools that extract and display a selected page from a PostScript language program must scan the program from its beginning until the desired page is found. On average, the time needed to view a page depends not only on the complexity of the page but also on the total number of pages in the document. This is problematic for interactive document viewing, where it is important that the time needed to view a page be independent of the total number of pages in the document.

Every PDF file contains a cross-reference table that can be used to locate and directly access pages and other important objects in the file. The location of the cross-reference table is stored at the end of the file, allowing applications that produce PDF files in a single pass to store it easily and allowing applications that read PDF files to locate it easily. Using the cross-reference table, the time needed to view a page in a PDF file can be nearly independent of the total number of pages in the document.

2.3.7 Incremental update

Applications may allow users to modify PDF documents, which can contain hundreds of pages or more. Users should not have to wait for the entire file 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 the objects that were modified or added, and includes an update to the cross-reference table. Support for incremental update allows an application to save modifications to a PDF document in an amount of time proportional to the size of the modification instead of the size of the file. In addition, because the original contents of the file are still present in the file, it is possible to undo saved changes by deleting one or more addenda.

2.3.8 Extensibility

PDF is designed to be extensible. Undoubtedly, developers will want to add features to PDF that have not yet been implemented or thought of.

The design of PDF is such that not only can new features be added, but applications that understand earlier versions of the format will not completely break when they encounter features that they do not implement. Appendix G, "Compatibility and Implementation Notes," specifies how a viewer should behave when it reads a file that does not conform to the specification it was expecting.

2.4 PDF and the PostScript language

The preceding sections mentioned several ways in which PDF differs from the PostScript language. This section summarizes these differences and describes the process of converting a PDF file into a PostScript language program.

While PDF and the PostScript language share the same basic imaging model, there are some important differences between them:

- A PDF file may contain objects such as hypertext links that are useful only for interactive viewing.
- To simplify the processing of page descriptions, PDF provides no programming language constructs.
- PDF enforces a strictly defined file structure that allows an application to access parts of a document randomly.
- PDF files contain information such as font metrics, to ensure viewing fidelity.

Because of these differences, a PDF file cannot be downloaded directly to a PostScript printer for printing. An application that prints a PDF file to a PostScript printer must carry out the following steps:

1. Insert *procsets*, sets of PostScript language procedure definitions that implement the PDF page description operators.
2. Extract the content for each page. Pages are not necessarily stored in sequential order in the PDF file. Each page description is essentially the script portion of a traditional PostScript language program using very specific procedures, such as "m" for **moveto** and "l" for **lineto**.
3. Decode compressed text, graphics, and image data. Except for data encoded with the Flate filter, this is not required for PostScript Level 2 printers, which can accept compressed data in a PostScript language file.
4. Insert any resources, such as fonts, into the PostScript language file. Substitute fonts are defined and inserted as needed, based on the font metrics in the PDF file.

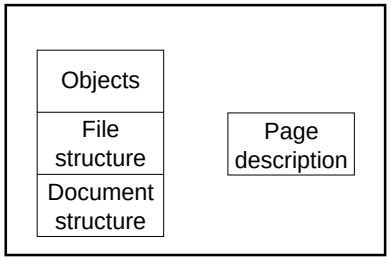
5. Put the information in the correct order. The result is a traditional PostScript language program that fully represents the visual aspects of the document, but no longer contains PDF elements such as hypertext links, annotations, and bookmarks.

6. Send the PostScript language program to the printer.

2.5 Understanding PDF

PDF is best understood by thinking of it in four parts, as shown in Figure 2.4.

Figure 2.4 *PDF components*



The first component is the set of basic object types used by PDF to represent objects. These types, with only a few exceptions, correspond to the data types used in the PostScript language. Chapter 4 discusses these object types.

The second component is the PDF file structure. The 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. Chapter 5 explains the file structure.

The third component is the PDF document structure. The document structure specifies how the basic object types are used to represent components of a PDF document: pages, annotations, hypertext links, fonts, and more. Chapter 6 explains the PDF document structure.

The fourth and final component is the PDF page description. A PDF page description, while part of a PDF page object, can be explained independently of the other components. A PDF page description has only limited interaction with other parts of a PDF document. This simplifies its conversion into a PostScript language program. Chapter 8 discusses PDF page descriptions.

CHAPTER 3

Coordinate Systems

Coordinate systems define the canvas on which all drawing in a PDF document occurs; that is, the position, orientation, and size of the text, graphics, and images that appear on a page are determined by coordinate systems.

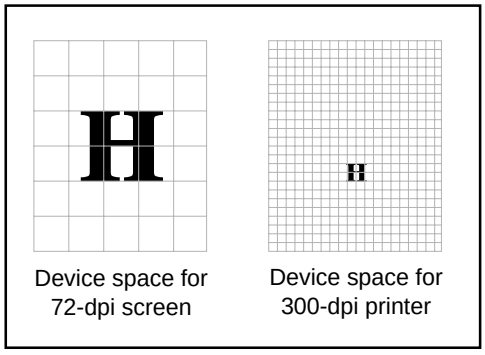
PDF supports a number of coordinate systems, most of them identical to those used in the PostScript language. This chapter describes each of the coordinate systems used in PDF, how they are related, and how transformations among coordinate systems are specified. At the end of the chapter is a description of the mathematics involved in coordinate transformations. It is not necessary to read this section to use coordinate systems and transformations. It is presented for those readers who wish to gain a deeper understanding of the mechanics of coordinate transformations.

3.1 Device space

The contents of a page ultimately appear on a display or a printer. Each type of device on which a PDF page can be drawn has its own built-in coordinate system, and, in general, each type of device has a different coordinate system. Coordinates specified in a device's native coordinate system are said to be in *device space*. On pixel-based devices such as computer screens and laser printers, coordinates in device space generally specify a particular pixel.

If coordinates in PDF files were specified in device space, the files would be device-dependent and would accordingly appear differently on different devices. For example, images drawn in the typical device space of a 72 pixel per inch display and on a 600 dpi printer differ in size by more than a factor of 8; an eight-inch line segment on a display would appear as a one-inch segment on the printer. Different devices also have different orientations of their coordinate systems. On one device, the origin of the coordinate system may be at the upper left corner of the page, with the positive direction of the *y*-axis pointing downward. On another device, the origin may be in the lower left corner of the page with the positive direction of the *y*-axis pointing upward. Figure 3.1 shows an object that is two units high in device space, and illustrates the fact that coordinates specified in device space are device-dependent.

Figure 3.1 *Device space*

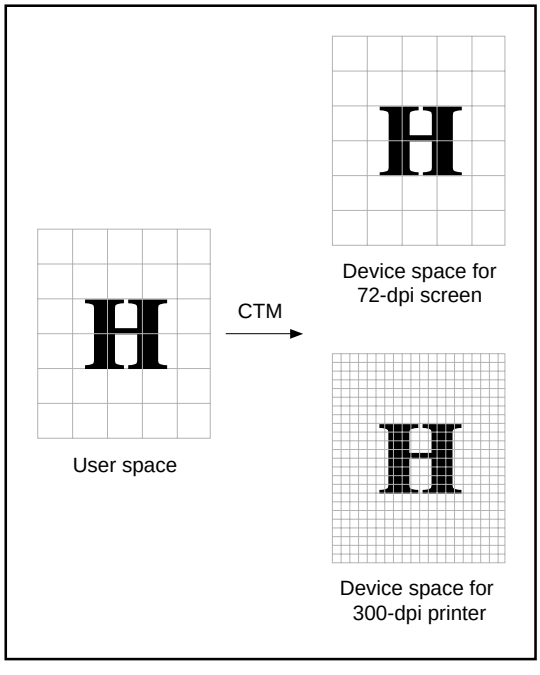


3.2 User space

PDF, like the PostScript language, defines a coordinate system that appears the same, regardless of the device on which output occurs. This allows PDF documents to be independent of the resolution of the output device. This resolution-independent coordinate system is called *user space* and provides the overall coordinate system for a page.

The transformation from user space to device space is specified by the *current transformation matrix* (CTM). Figure 3.2 shows an object that is two units high in user space and indicates that the CTM provides the resolution-independence of the user space coordinate system.

Figure 3.2 *User space*



The user space coordinate system is initialized to a default state for each page of a document. By default, user space coordinates have 72 units per inch, corresponding roughly to the various definitions of the typographic unit of measurement known as the *point*. The positive direction of the *y*-axis points upward, and the positive direction of the *x*-axis to the right. The region of the default coordinate system that is viewed or printed can be different for each page, and is described on page 77.

3.3 Text space

The coordinates of text are specified in *text space*. The transformation from text space to user space is provided by a matrix called the *text matrix*.

3.4 Character space

Characters in a font are defined in *character space*. The transformation from character space to text space is defined by a matrix. For most types of fonts, this matrix is predefined except for an overall scale factor. (For details, see page 227.) This scale factor changes when a user selects the font size for text.

3.5 Image space

All images are defined in *image space*. The transformation from image space to user space is predefined and cannot be changed. All images are one unit by one unit in user space, regardless of the number of samples in the image.

3.6 Form space

PDF provides an object known as a *Form*, discussed on page 118. Forms contain sequences of operations and are the same as forms in the PostScript language. The space in which a form is defined is *form space*. The transformation from form space to user space is specified by a matrix contained in the form.

3.7 Pattern space

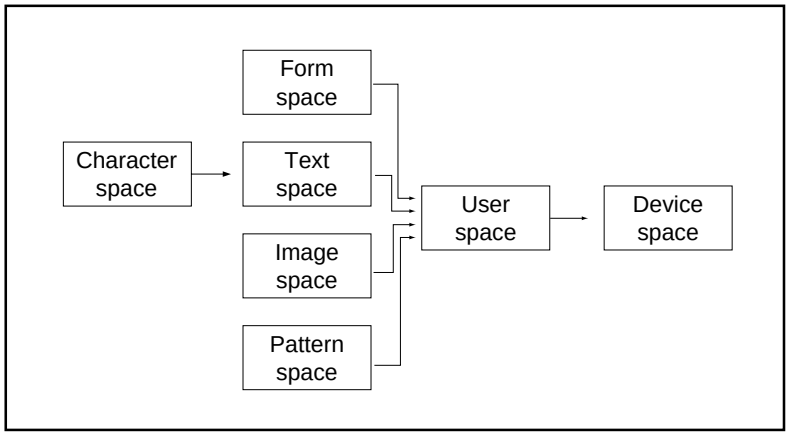
PDF 1.2

PDF defines a type of color known as a *pattern*, discussed on page 201. Like forms, patterns contain sequences of marking operations; they are the same as patterns in the PostScript language. The space in which a pattern is defined is *pattern space*. The transformation from pattern space to user space is specified by a matrix contained in the pattern.

3.8 Relationships among coordinate systems

PDF defines a number of interrelated coordinate systems, described in the previous sections. Figure 3.3 shows the relationships among the coordinate systems. Each line in the figure represents a transformation from one coordinate system to another. PDF allows modifications to many of these transformations.

Figure 3.3 *Relationships among PDF coordinate systems*



Because PDF coordinate systems are defined relative to each other, changes made to one transformation can affect the appearance of objects drawn in several coordinate systems. For example, changes made to the CTM affect the appearance of all objects, not just graphics drawn directly in user space.

3.9 Transformations between coordinate systems

Transformation matrices specify the relationship between two coordinate systems. By modifying a transformation matrix, objects can be scaled, rotated, translated, or transformed in other ways.

A transformation matrix in PDF, as in the PostScript language, is specified by six numbers, usually in the form of an array containing six elements. This section lists the arrays that specify the most common transformations. The following section contains more mathematical details of transformations, including information on specifying transformations that are combinations of those listed in this section.

- 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 *x* and *y*, respectively.

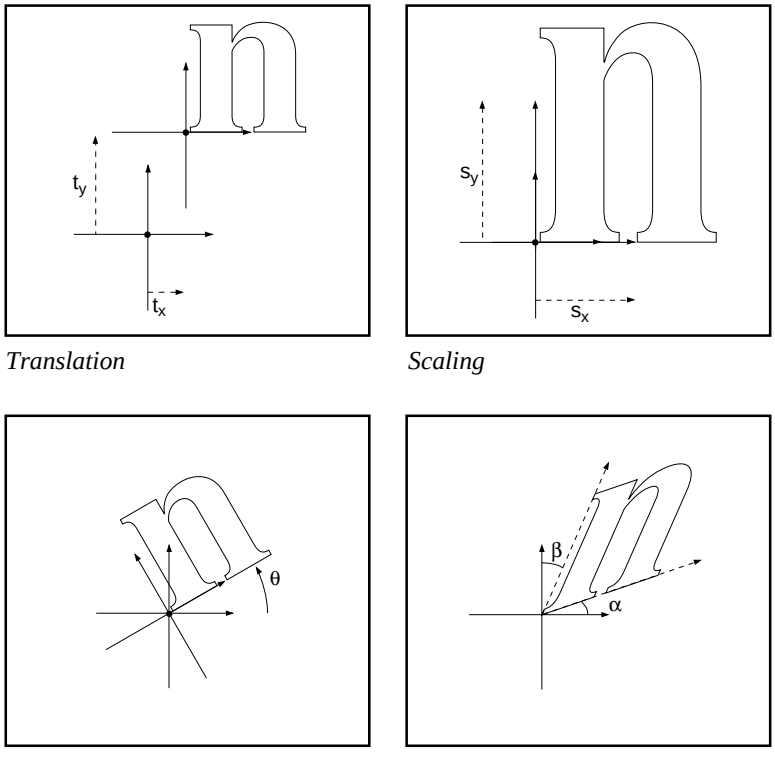
- Scaling is obtained by [s_x 0 0 s_y 0 0]. This scales the coordinates so that one unit in the *x* and *y* directions of the new coordinate system is the same size as s_x and s_y units in the previous coordinate system, respectively.

- Rotations are carried out 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 $\tan\alpha$ $\tan\beta$ 1 0 0], which skews the *x*-axis by an angle α and the *y*-axis by an angle β.

Figure 3.4 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 3.4 *Effects of coordinate transformations*



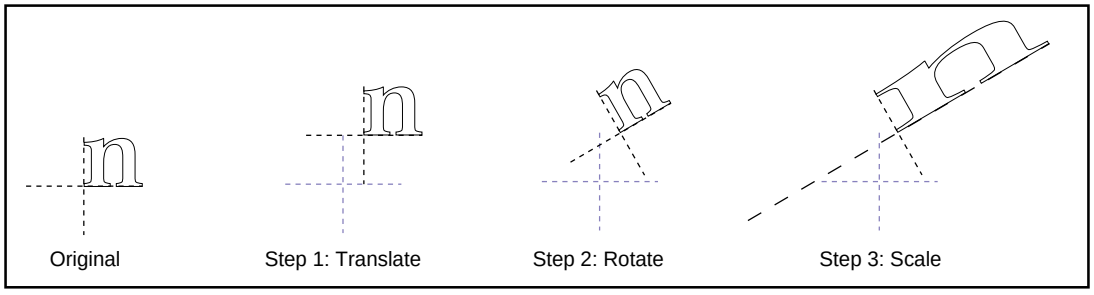
Translation *Scaling*

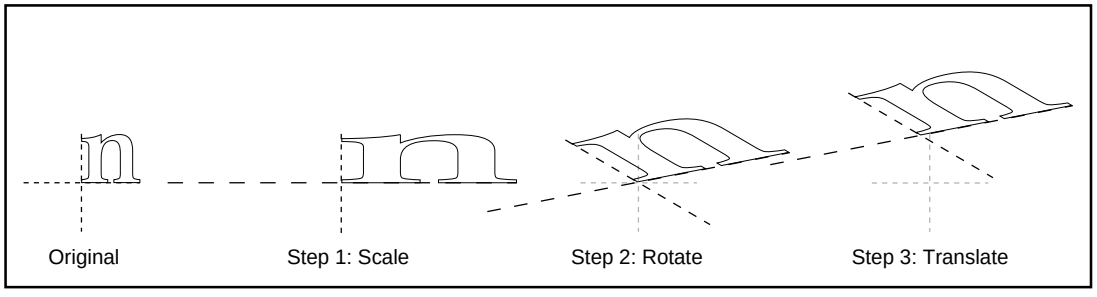
Rotation *Skewing*

If several transformations are applied, the order in which they are applied generally is important. For example, scaling the *x*-axis followed by a translation of the *x*-axis is not the same as first translating the *x*-axis, then performing the scaling. In general, to obtain the expected results, transformations should be done in the order: translate, rotate, scale/skew.

Figure 3.5 shows that the order in which transformations are applied is important. The figure shows two sequences of transformations applied to a coordinate system. After each successive transformation, an outline of the letter "n" is drawn. The transformations in the figure are a translation of 10 units in the *x*-direction and 20 units in the *y*-direction, a rotation of 30 degrees, and a scaling by a factor of 3 in the *x*-direction. In the figure, the axes are drawn with a dash-pattern having two units dash, two units gap. In addition, the untransformed coordinate system is drawn in a lighter color in each section. 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.

Figure 3.5 *Effect of the order of transformations*





3.10 Transformation matrices

This section describes the mathematics of transformation matrices, which is identical to that underlying the PostScript language. It is not necessary to read this section to use the transformations discussed in previous sections.

To understand coordinate system transformations in PDF, it is vital to understand two points:

- Transformations in PDF alter coordinate systems, not objects. All objects drawn before a transformation is specified are unchanged by the transformation. Objects drawn after the transformation is specified will be drawn in the transformed coordinate system.

- Transformation matrices in PDF specify the transformation from the transformed (new) coordinate system to the untransformed (old) coordinate system. All coordinates used after the transformation are specified in the transformed coordinate system. PDF applies the transformation matrix to determine the coordinates in the untransformed coordinate system.

Note *Many computer graphics textbooks consider transformations of objects instead of coordinate systems. Although these are formally equivalent, some results 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]. Although the third element of this vector (1) is not strictly necessary, it provides a convenient way to specify translations of the coordinate system's origin.

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

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

Note *Because a transformation matrix has only six entries that may be changed, for convenience it is often written as the six-element array [a b c d e f].*

Coordinate transformations are expressed as:

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

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:

$$x' = ax + cy + e$$

$$y' = bx + dy + f$$

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

Matrix multiplication is not commutative—the order in which matrices are multiplied is significant. It is not *a priori* obvious in which order the transformation matrices should be multiplied. Matrices representing later transformations could either be multiplied before those representing earlier transformations (premultiplied) or after (postmultiplied).

To determine whether premultiplication or postmultiplication is appropriate, consider a sequence of two transformations. Specifically, apply a scaling transformation to the user space coordinate system, and consider the conversion from this scaled coordinate system to device space. The two transformation matrices in this example are the matrix specifying the scaling (M_S) and the matrix specifying the transformation from user space to device space (the CTM, called M_C here). Recalling that coordinates are always specified in the transformed space, it is clear that the correct order of transformations must first convert the scaled coordinates to those in default user space, and then convert the default user space coordinates to device space coordinates. This can be expressed:

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

where X_D is the coordinate in device space and X_U is the coordinate in default user space. This shows that when a new transformation is added, the matrix representing it must be premultiplied onto 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 transformation being added (M_T) onto the matrix representing any existing transformations (M):

$$M' = M_T M$$

CHAPTER 4

Objects

The object types supported by PDF are similar to the object types supported by the PostScript language. Readers familiar with the PostScript language may wish to skim this chapter, or skip parts of it, particularly Sections 4.2, “Booleans,” through 4.7, “Dictionaries.”

4.1 Introduction

PDF supports seven basic types of objects: Booleans, numbers, strings, names, arrays, dictionaries, and streams. In addition, PDF provides a 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 and the null object. A discussion of creating and referring to indirect objects in PDF files follows.

Note *PDF is case-sensitive. Uppercase and lowercase letters are different.*

4.2 Booleans

The keywords **true** and **false** represent Boolean objects with values *true* and *false*.

4.3 Numbers

PDF provides two types of numbers, integer and real. Integers may be specified by signed or unsigned constants. Reals may only be in decimal format. Throughout this book, *number* means an object whose type is either integer or real.

Note *Exponential format for numbers (such as 1.0E3) is not supported.*

4.4 Strings

A string is a sequence of characters delimited by parentheses. 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. When this occurs, the backslash and end-of-line characters are not considered part of the string. Examples of strings are:

```
( This is string number 1? )
( strangeonium spectroscopy )
(This string is split \
across \
three lines)
```

Within a string, the backslash character is used as an escape to specify unbalanced parentheses, non-printing ASCII characters, and the backslash character itself. This escape mechanism is the same as for PostScript language strings, described in Section 3.2.2 of the *PostScript Language Reference Manual, Second Edition*. Table 4.1 lists the escape sequences for PDF.

Table 4.1 *Escape sequences in strings*

`\n`	linefeed
`\r`	carriage return
`\t`	horizontal tab
`\b`	backspace
`\f`	formfeed
`\\`	backslash
`\(`	left parenthesis
`\)`	right parenthesis
`\`*ddd*	character code *ddd* (octal)

Use of the `\`*ddd* escape sequence is the preferred way to represent characters outside the printable ASCII character set, in order to minimize potential problems transmitting or storing the characters. The number *ddd* may contain one, two, or three octal digits. An example of a string with an octal character in it is:

```
(string with \245two octal characters\307)
```

As in the PostScript language, strings may also be represented in hexadecimal form. A hexadecimal string consists of a sequence of hexadecimal characters (the digits 0–9 and the letters A–F or a–f) enclosed within angle brackets (< and >). Each pair of hexadecimal digits defines one character of the string. If the final digit of a given string is missing—in other words, if there is an odd number of digits—the final digit is assumed to be zero. Whitespace characters (space, tab, carriage return, linefeed, and formfeed) are ignored. For example,

```
<901fa3>
```

is a three-character string consisting of the characters whose hexadecimal codes are 90, 1f, and a3. But:

```
<901fa>
```

is a three-character string containing the characters whose hexadecimal codes are 90, 1f, and a0.

In versions 1.1 and later, it is not necessary to represent strings using only the printable 7-bit ASCII character set. A non-printable ASCII code—in fact, any 8-bit value—may appear in a string. In particular, when a document is encrypted (see page 68), all 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 and the backslash character itself.

PDF 1.1

Informational or content strings can be represented in Unicode. These strings include text annotations, bookmark names, article names, document information, date strings, etc. In PDF 1.1 these strings are stored in **PDFDocEncoding**, which is a superset of ISOLatin1. **PDFDocEncoding** is compatible with Unicode in that all Unicode codes less than 256 match **PDFDocEncoding**.

PDF 1.2

Strings can be stored in either **PDFDocEncoding** or Unicode. If they are in Unicode, the first two bytes of the string must be the Unicode Byte Order marker, <FE FF>. This sequence collides with the character sequence *thorn ydieresis*, which is not likely to be a meaningful beginning of a word or phrase. The high-order byte of a Unicode character appears first in a string.

The string may also contain an escape sequence to indicate the language of the string. This is useful when the language cannot be determined from the character codes used in the string. The escape sequence uses the Unicode hex value U+001B followed by the two ASCII codes for the language identifiers defined by ISO 639 (see Appendix I), optionally followed by the two ASCII codes for country defined by IS0 3166 (see Appendix J), followed by U+001B.

Strings can be used for many purposes and can be formatted in different 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. Such formats are conventions for interpreting strings and are not types themselves. The use of a particular format is indicated with the definition of the string object that uses the format.

4.5 Names

A name, like a string, is a sequence of characters. It must begin with a slash followed by a sequence of ASCII characters in the range ! (<21>) through ~ (<7E>) except %, (,), <, >, [,], {, }, /, and #. Examples of names are:

```
/Name1
/ASomewhatLongerName2
/A;Name_With-various***Characters?.
```

```
/1.2
/.notdef
```

Any character except null (<00>) may be included in a name by writing its two-character hex code, preceded by #. Examples: **PDF 1.2**

```
/Adobe#20Green
/PANTONE#205757#20CV
/paired#28#29parentheses
/TheKeyOfF#23Minor
/A#42
```

Note that `/A#42` is another way of writing the name `/AB`.

Note *PDF 1.1 allowed # as part of a name; this change is incompatible. PDF 1.1 did not exclude the use of blanks in names, although that was not permitted by the Acrobat implementation. There was no "escape character" in PDF 1.1.*

Note *The maximum number of characters in a name is 127. This limit refers to the internal representation of the name. For example, the name* `/A#20B` *has four characters (/, A, space, and B), not six.*

4.6 Arrays

An array is a sequence of PDF objects. An array may contain a mixture of object types. An array is written as a left square bracket (`[`), followed by a sequence of objects, followed by a right square bracket (`]`). An example of an array is:

```
[0 (Higgs) false 3.14 3 549 /SomeName]
```

4.7 Dictionaries

A dictionary is an associative table containing pairs of objects. The first element of each pair is called the *key* and the second element is called the *value*. Unlike dictionaries in the PostScript language, a key must be a name. A value can be any kind of object, including a dictionary. A dictionary is generally used to collect and tie together the attributes of a complex object, with each key–value pair specifying the name and value of an attribute.

A dictionary is represented by two left angle brackets (`<<`), followed by a sequence of key–value pairs, followed by two right angle brackets (`>>`). For example:

Example 4.1 *Dictionary*

```
<< /Type /Example /Key2 12 /Key3 (a string) >>
```

Example 4.2 *Dictionary within a dictionary*

```
<<
/Type /AlsoAnExample
/Subtype /Bad
/Reason (unsure)
/Version 0.01
/MyInfo
     <<
     /Item1 0.4
     /Item2 true
     /LastItem (not!)
     /VeryLastItem (OK)
     >>
>>
```

Dictionary objects are the main building blocks of a PDF document. Many parts of a PDF document, such as pages and fonts, are represented using dictionaries. By convention, the **Type** key of such a dictionary specifies the type of object being described by the dictionary. Its value is always a name. In some cases, the **Subtype** key is used to describe a specialization of a particular type. Its value is always a name. For a font, **Type** is **Font** and several subtypes exist, including **Type1**, **MMType1**, **Type3**, **TrueType**, and others.

4.8 Streams

A stream, like a string, is a sequence of characters. However, an application can read a small portion of a stream at a time, while a string must be read in its entirety. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams.

A stream consists of a dictionary that describes a sequence of characters, followed by the keyword **stream**, followed by zero or more lines of characters, followed by the keyword **endstream**.

<stream> ::= <dictionary>
stream
{ <lines of characters> }*
endstream

All streams must be indirect objects (see page 59). The stream dictionary must be a direct object. The keyword **stream** that follows the stream dictionary should be followed by a carriage return and linefeed or just a linefeed.

PDF 1.1

Note *Without this restriction, it is not possible to differentiate a stream that uses carriage return as end of line and whose first byte of data is a linefeed from a stream that uses carriage return-linefeed pairs as end of line.*

The sequence of characters that make up a stream may be found between the **stream** and **endstream** keywords or, in PDF 1.2, may be contained in an external file. If the data is in an external file, the stream dictionary specifies the file. When a stream's data is external, the characters between **stream** and **endstream** are ignored. **PDF 1.2**

Table 4.2 shows the attributes of a stream.

Table 4.2 *Stream attributes*

Key	*Type*	*Description*
Length	integer	*(Required)* Number of characters from the first line after the line containing the **stream** keyword to the **endstream** keyword.
Filter	name or array of names	*(Optional)* Filters to be applied in processing the stream. The value of the **Filter** key can be either the name of a single decode filter or an array of decode filter names. Specify multiple filters in the order they should be applied to decode the data. For example, data encoded using LZW and ASCII base-85 filters (in that order) can be decoded by providing the following key and value in the stream dictionary: `/Filter [/ASCII85Decode /LZWDecode]`
DecodeParms	*various*	*(Optional)* Parameters used by the decoding filters specified with the **Filter** key. The number and types of the parameters supplied must match those needed by the specified filters. For example, if two filters are used, the decode parameters must be specified by an array of two objects, one corresponding to each filter. Use the null object for a filter's entry in the **DecodeParms** array if that filter does not need any parameters. If none of the filters specified requires any parameters, omit the **DecodeParms** key.
F (File)	File specification	*(Optional)* The file containing the stream data. If this key is present, the characters between **stream** and **endstream** are ignored. However, the **Length** key should still specify the number of those characters. (Usually there are no characters and the **Length** is zero.) **PDF 1.2**
FFilter	name or array of names	*(Optional)* Filters to be applied in processing the data found in the stream's external file. The same rules apply as for the **Filter** key. **PDF 1.2**
FDecodeParms	variable	*(Optional)* Parameters used by the decoding filters specified with the **FFilter** key. **PDF 1.2**

Streams may be filtered to compress them or convert binary streams into ASCII form. These filters and their parameters are listed in Table 4.3 and described in the following sections.

Table 4.3 *Standard filters*

Filter name	*Parameters*	*Semantics*	
ASCIIHexDecode	none	Decodes binary data in an ASCII hexadecimal representation	
ASCII85Decode	none	Decodes binary data in an ASCII base-85 representation	
LZWDecode	dictionary	*(Parameters optional)* Decompresses text or binary data using LZW adaptive compression method	
RunLengthDecode	none	Decompresses binary data using a byte-oriented run-length decoding algorithm	
CCITTFaxDecode	dictionary	*(Parameters optional)* Decompresses binary data using a bit-oriented decoding algorithm, the CCITT facsimile standard	
DCTDecode	dictionary	*(Parameters optional)* Decompresses sampled image data using a discrete cosine transform technique based on the JPEG standard	
FlateDecode	dictionary	*(Parameters optional)* Decompresses text or binary data using the Flate decompression method.	**PDF 1.2**

Example 4.3 shows a stream that has been compressed using LZW and then encoded using ASCII85, while Example 4.4 shows the same stream without any encoding.

Example 4.3 *Stream that has been LZW and ASCII85 encoded*

```
<<
/Length 528
/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%IY!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
```

Example 4.4 *Unencoded stream*

```
<<
/Length 558
>>
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
```

4.8.1 ASCIIHexDecode filter

This filter decodes data that has been encoded as ASCII hexadecimal. ASCII hexadecimal encoding and ASCII base-85 encoding (described in the following section) convert binary data such as images to 7-bit data. In general, ASCII base-85 encoding is preferred because it is more compact.

ASCII hexadecimal encoding produces a 1:2 expansion in the size of the data. Each pair of ASCII hexadecimal digits (0–9 and A–F or a–f) produces one byte of binary data. All whitespace characters are ignored. The right angle bracket (>) indicates the end of data (EOD). Any other character causes an error. If the filter encounters the EOD marker after reading an odd number of hexadecimal digits, it behaves as if a zero followed the last digit.

4.8.2 ASCII85Decode filter

This filter decodes data that has been encoded in the ASCII base-85 encoding and produces binary data.

ASCII base-85 encoding produces five ASCII printing characters from every four bytes of binary data. Each group of four binary bytes ($b_1\ b_2\ b_3\ b_4$) is converted to a group of five encoded characters ($c_1\ c_2\ c_3\ c_4\ c_5$) using the relation:

$$(b_1 \times 256^3) + (b_2 \times 256^2) + (b_3 \times 256) + b_4 =$$

$$(c_1 \times 85^4) + (c_2 \times 85^3) + (c_3 \times 85^2) + (c_4 \times 85) + c_5$$

The five "digits" of the encoded base-85 number are converted to printable ASCII characters by adding 33 (the ASCII code for `!`) to each. The resulting data contains only printable ASCII characters with codes in the range 33 (!) to 117 (u).

Two special cases occur during encoding. First, if all five encoded digits are zero, they are represented by the character code 122 (z), instead of by a series of five exclamation points (!!!!!). In addition, if the length of the binary data to be encoded is not a multiple of four bytes, the last partial 4-tuple is used to produce a last, partial output 5-tuple. Given *n* (1, 2, or 3) bytes of binary data, the encoding first appends 4 – *n* zero bytes to make a complete 4-tuple. This 4-tuple is encoded in the usual way, but without applying the special z case. Finally, only the first *n* + 1 characters of the resulting 5-tuple are written out. Those characters are immediately followed by the EOD marker, which is the two-character sequence `~>`.

The following conditions are errors during decoding:

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

These conditions never occur in the output produced from a correctly encoded byte sequence.

4.8.3 LZWDecode filter

This filter decodes data encoded using the LZW data compression method, which is a variable-length, adaptive compression method. LZW encoding compresses binary and ASCII text data but always produces binary data, even if the original data was ASCII text. If necessary, this binary data may be converted to 7-bit data using either the ASCII hexadecimal or ASCII base-85 encodings described in previous sections.

LZW compression can discover and exploit many patterns in its input data, whether that input is text or image data. The compression obtained using the LZW method varies from file to file; the best case (a file of all zeros) provides a compression approaching 1365:1 for long files, while the worst case (a file in which no pair of adjacent characters appears twice) can produce an expansion of approximately 50%.

Data encoded using LZW consist 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 multi-character sequence that has been encountered previously in the input (258 and greater).

Initially, the code length is 9 bits and the 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 input character sequences. The encoding and decoding filters maintain identical copies of this table.

Whenever both encoder and decoder independently (but synchronously) realize that the current code length is no longer sufficient to represent the number of entries in the table, they increase the number of bits per code by one. The first output code that is 10 bits long is the one following 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. Output 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.

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

The LZW filter can be used to compress text or images. When compressing images, several techniques reduce the size of the resulting compressed data. For example, image data frequently change very little from sample to sample. By subtracting the values of adjacent samples (a process called *differencing*) and LZW-encoding the differences rather than the raw sample values, the size of the output data may be reduced. Further, 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 like components in adjacent samples, rather than between different color components in the same sample, often reduces the output data size. In order to control these and other options, the LZW filter accepts

several optional parameters, shown in Table 4.4. All values supplied to the decode filter by any optional parameters must match those used when the data was encoded.

Table 4.4 *Optional parameters for LZW filter*

Key	*Type*	*Semantics*
Predictor	integer	If **Predictor** is 1, the file is decoded assuming that it was encoded using the normal LZW algorithm. If **Predictor** > 1, decoding is performed assuming that prior to encoding, the data was differenced. The default value is 1. For details on this value, see Section 4.8.5 on page 54.
Columns	integer	Has an effect only if **Predictor** > 1. **Columns** is the number of samples in a sampled row. The first sample in each row is not differenced; all subsequent samples in a row are differenced with the prior sample. Each row begins on a byte boundary. Any extra bits needed to complete a byte at the end of a row (consisting of **Columns** × **Colors** × **BitsPerComponent** bits) are not differenced. The default value is 1.
Colors	integer	Has an effect only if **Predictor** > 1. Number of interleaved color components per sample in a sampled image. Each color component is differenced with the value of the same color component in the previous sample. Allowed values are 1, 2, 3, and 4. The default value is 1.
BitsPerComponent	integer	Has an effect only if **Predictor** > 1. **BitsPerComponent** is the number of bits used to represent each color component in a pixel. Allowed values are 1, 2, 4, and 8. The default value is 8.
EarlyChange	integer	If **EarlyChange** is 0, increases in the length of the code are postponed as long as possible. If it is 1, they occur one code word early. The value of **EarlyChange** used in decoding must match that used during encoding. This parameter is included because LZW sample code distributed by some vendors increases the code word length one word earlier than necessary. The default value is 1.

The LZW compression method is the subject of United States patent number 4,558,302 owned by the Unisys Corporation. Adobe Systems has licensed this patent for use in its products, including Acrobat products. However, independent software vendors may be required to license this patent directly from Unisys to develop software using LZW. Further information can be obtained from Welch Licensing Department, Law Department, M/S C2SW1, Unisys Corporation, Blue Bell, Pennsylvania, 19424.

4.8.4 FlateDecode Filter

PDF 1.2

This filter decodes data that has been encoded with the Flate compression method, which encodes binary or ASCII data, optionally after transformation by a predictor function. It 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. The zlib/deflate compression method is fully defined in Internet Engineering Task Force Requests for Comments (IETF RFCs) 1950 and 1951. The optional Predictor functions are discussed in the following text.

The output produced by Flate encoding is always binary, even if the input is ASCII text.

Table 4.5 *Optional parameters for FlateDecode filter*

Key	*Type*	*Semantics*
Predictor	integer	If **Predictor** is 1, the file is decoded assuming that it was encoded using the normal Flate algorithm. If **Predictor** > 1, decoding is performed assuming that prior to encoding, the data was differenced. The default value is 1. For details on this value, see Section 4.8.5, "LZW and Flate predictor functions".
Columns	integer	Has an effect only if **Predictor** > 1. **Columns** is the number of samples in a sampled row. The first sample in each row is not differenced; all subsequent samples in a row are differenced with the prior sample. Each row begins on a byte boundary. Any extra bits needed to complete a byte at the end of a row (consisting of **Columns** × **Colors** × **BitsPerComponent** bits) are not differenced. The default value is 1.
Colors	integer	Has an effect only if **Predictor** > 1. **Colors** is the number of interleaved color components per sample in a sampled image. Each color component is differenced with the value of the same color component in the previous sample. Allowed values are 1, 2, 3, and 4. The default value is 1.
BitsPerComponent	integer	Has an effect only if **Predictor** > 1. **BitsPerComponent** is the number of bits used to represent each color component in a pixel. Allowed values are 1, 2, 4, and 8. The default value is 8.

4.8.5 LZW and Flate predictor functions

PDF 1.2

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

Two predictor function groups are supported. The first, the TIFF group, consists of the single function that is Predictor 2 in the TIFF standard. (In TIFF 6.0, it applies only to LZW compression, but here it applies to Flate compression as well.) TIFF predictor 2 predicts that each color component of a pixel will be the same as the corresponding color component of the pixel immediately to the left.

The second supported predictor function group, the *PNG* group, consists of the "filters" of the World Wide Web Consortium's Portable Network Graphics (PNG) recommendation. The term *predictors* is used here instead of *filters*, to avoid confusion. There are five basic PNG predictor algorithms, and a sixth one that

invites an optimum hybrid of the first five. The first five are *None*, *Sub* (predicting the same as the pixel to the left), *Up* (predicting the same as the pixel above), *Average* (predicting the average of the pixel to the left and the pixel above), and *Paeth* (a nonlinear function of the pixel above, the pixel to the left, and the pixel to the upper left).

The two predictor function groups have some commonalities. Both assume that data are presented in order, from top row to bottom row, and within a row, from left to right. Both assume that a row occupies a whole number of bytes, rounded upward if necessary. Both assume that pixels and their components are packed into bytes from high- to low-order bits. Both assume that all color components of pixels outside the image (which are necessary for predictions near the boundaries) are 0.

Table 4.6 *Predictor values*

1	No prediction (Default value)
2	TIFF predictor 2
10	PNG prediction (on encode, PNG *None* on all rows)
11	PNG prediction (on encode, PNG *Sub* on all rows)
12	PNG prediction (on encode, PNG *Up* on all rows)
13	PNG prediction (on encode, PNG *Average* on all rows)
14	PNG prediction (on encode, PNG *Paeth* on all rows)
15	PNG prediction (on encode, PNG optimum)

The two predictor function groups also differ in significant ways. First, the postprediction data for each PNG-predicted row begins with an explicit algorithm tag, so different rows can be predicted with different algorithm to improve compression. TIFF 2 prediction has no such identifier; the same algorithm applies to all rows. Second, the TIFF function group predicts each color component from the prior instance of that color component, without regard to the width of the color component or the number of colors. In contrast, the PNG function group predicts each byte from the corresponding byte of the prior pixel (and/or the same pixel on the prior line and/or the prior pixel on the prior line), 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 a cost somewhat worse compression.

4.8.6 Comparison of LZW and Flate encoding

PDF 1.2

Flate encoding, like LZW encoding, discovers and exploits many patterns in its input data, whether text or images. Thanks to its cascaded adaptive Huffman coding, Flate-encoded output is usually substantially more compact than LZW-encoded output for the same input. Flate and LZW decoding speeds are comparable, but Flate encoding speed is considerably slower than LZW encoding speed. Usually, both Flate and LZW compress their inputs substantially. In the worst case, however, Flate encoding expands its input by no more than 11 bytes, plus the effects of algorithm tags added by PNG predictors. LZW encoding has a

worst-case expansion of at least a factor of 1.125, which can increase to a factor of nearly 1.5 in some implementations (plus PNG tags effects, as with Flate encoding).

4.8.7 RunLengthDecode filter

This filter decodes data that has been encoded in a simple byte-oriented, run-length-encoded format. Run-length encoding produces binary data, even if the original data was ASCII text.

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

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 *length* 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* times (2 to 128 times) during decompression. The value 128 is placed at the end of the compressed data, as an EOD marker.

4.8.8 CCITTFaxDecode filter

This filter decodes image data that has been encoded using either Group 3 or Group 4 CCITT facsimile (fax) encoding. This filter is useful only for bitmap image data, not for color images, grayscale images, or text. Group 3 and Group 4 CCITT encoding produce binary data that may be converted, if necessary, to 7-bit data using either the ASCII hexadecimal or ASCII base-85 encodings, described in previous sections.

The compression achieved using CCITT encodings depends on the data, as well as on the value of various optional parameters. For Group 3 one-dimensional encoding, the best case is a file of all zeros. In this case, 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.

CCITT encoding is defined by an international standards organization, the International Coordinating Committee for Telephony and Telegraphy (CCITT). The encoding is designed to achieve efficient compression of monochrome (1 bit per sample) image data at relatively low resolutions. The algorithm is not described in detail here, but can be found in the CCITT standards, [13] and [14], listed in the Bibliography on page 390.

The fax encoding method is bit-oriented, rather than byte-oriented. This means that, in principle, encoded or decoded data may not end on a byte boundary. The filter addresses this in the following ways:

- Encoded data are ordinarily treated as a continuous, unbroken bit stream. However, the **EncodedByteAlign** parameter (described in Table 4.7) 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 don't 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.

Both Group 3 and Group 4 encoding, as well as optional features of the CCITT standard, are supported. The optional parameters that can be used to control the decoding are listed in Table 4.7. Except as noted, all values supplied to the decode filter by the optional parameters must match those used when the data was encoded.

Table 4.7 *Optional parameters for CCITTFaxDecode filter*

Key	*Type*	*Semantics*
K	integer	Selects the encoding scheme used. A negative value indicates pure two-dimensional (Group 4) encoding. Zero indicates pure one-dimensional (Group 3, 1-D) encoding. A positive value indicates 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 decoding filter distinguishes between negative, zero, and positive values of **K**, but does not distinguish between different positive **K** values. The default value is 0.
EndOfLine	Boolean	End-of-line bit patterns are always accepted but required if **EndOfLine** is *true*. The default value is *false*.
EncodedByteAlign	Boolean	If *true*, each encoded line must begin on a byte boundary. The default value is *false*.
Columns	integer	Specifies the width of the image in samples. If **Columns** is not a multiple of 8, the width of the unencoded image is adjusted to the next multiple of 8, so that each line starts on a byte boundary. The default value is 1728.
Rows	integer	Specifies the height of the image in scan lines. If this parameter is zero or is absent, the height of the image 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 source. The default value is 0.
EndOfBlock	Boolean	If *true*, the data is expected to be terminated by an end-of-block, overriding the **Rows** parameter. If *false*, decoding stops when **Rows** lines have been decoded or when the 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. The default value is *true*.
BlackIs1	Boolean	If *true*, causes bits with value 1 to be interpreted as black pixels and bits with value zero to be interpreted as white pixels. The default value is *false*.

DamagedRowsBeforeError

integer — If **DamagedRowsBeforeError** is positive, **EndOfLine** is *true*, and **K** is non-negative, then up to **DamagedRowsBeforeError** rows of data are tolerated before an error is generated. 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 damaged. The default value is 0.

4.8.9 DCTDecode filter

This filter decodes grayscale or color image data that has been encoded in the JPEG baseline format. JPEG encoding produces binary data.

JPEG is a lossy compression method, meaning that some of the information present in the original image is lost when the image is encoded. Because of the information loss, only images (never text) should be encoded in this format. The compression achieved using 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 a perceptible loss of information, and 30:1 compression causes little impairment of the image.

During encoding, several optional parameters control the algorithm and the information loss. The values of these parameters are stored in the encoded data, and the decoding filter generally obtains the parameter values it requires directly from the encoded data. A description of the parameters accepted by the encoding filter can be found in Section 3.13.3 of the *PostScript Language Reference Manual, Second Edition*.

JPEG stands for the ISO/CCITT Joint Photographic Experts Group, an organization responsible for developing an international standard for compression of color image data. The encoding method uses the discrete cosine transform (DCT). Data to be encoded consists of a stream of image samples, each containing 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.

The details of the encoding algorithm are not presented here but can be found in the references [18] and [21] listed in the Bibliography on page 389. Briefly, the JPEG algorithm breaks an image up into blocks of 8×8 samples. 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 the quantization that results in the loss of information in the JPEG algorithm. The quantized coefficients are then compressed.

The amount of loss incurred in JPEG encoding is controlled by the encoding filter, which can reduce the loss by making the step size in the quantization smaller at the expense of reducing the amount of compression achieved by the algorithm. The JPEG filter implementation in the Acrobat products does not support features of the JPEG standard that are not relevant. In addition, certain choices regarding reserved marker codes and other optional features of the standard have been made.

4.9 The null object

The keyword **null** represents the null object.

Note *The value of a dictionary key can be specified as* **null**. *A simpler but equivalent way to express this is to omit the key from the dictionary.*

4.10 Indirect objects

A *direct object* is a Boolean, number, string, name, array, dictionary, stream, or null, as described in the previous sections. An *indirect object* is an object that has been labeled so that it can be referenced by other objects. Any type of object may be labeled as an indirect object. Indirect objects are very useful; for example, if the length of a stream is not known before it is written, the value of the stream's **Length** key may be specified as an indirect object that is stored in the file after the stream.

An indirect object consists of an object identifier, a direct object, and the **endobj** keyword. The *object identifier* consists of an integer *object number*, an integer *generation number*, and the **obj** keyword:

<indirect object> ::=
<object ID>
<direct object>
endobj
<object ID> ::= <object number>
<generation number>
obj

The combination of object number and generation number serves as a unique identifier for an indirect object. Throughout its existence, an indirect object retains the object number and generation number it was initially assigned, even if the object is modified.

Each indirect object has a unique object number, and indirect objects are often but not necessarily numbered sequentially in the file, beginning with 1. Until an object in the file is deleted, all generation numbers are 0.

4.11 Object references

Any object used as an element of an array or as a value in a dictionary may be specified by either a direct object or an indirect reference. An *indirect reference* is a reference to an indirect object, and consists of the indirect object's object number, generation number, and the **R** keyword:

<indirect reference> ::=
<object number>
<generation number>
R

Using an indirect reference to the stream's length, a stream could be written in this way:

Example 4.1 *Indirect reference*

```
7 0 obj
<<
/Length 8 0 R
>>
stream
BT
/F1 12 Tf
72 712 Td (A stream with an indirect Length) Tj
ET
endstream
endobj
8 0 obj
64
endobj
```

Note *An indirect reference to an undefined object is not an error; it is treated as a reference to the null object. For example, if a PDF file contains the indirect reference (12 0 R) but does not contain the definition (12 0 obj ... endobj), then the indirect reference is null.*

CHAPTER 5

File Structure

This chapter describes the overall organization of a PDF file. A PDF file provides a structure that represents a document. This structure provides a way to rapidly access any part of a document and a mechanism for updating it.

The body of a PDF file contains a sequence of PDF objects that are used to construct a document. Chapter 4 describes the types of objects supported by PDF. Chapter 6 explains the way a document is constructed using these object types.

A particular arrangement of the object in a PDF file, optimized for incremental access in network environments, is known as *Linearized PDF*. It is described in Chapter 9. **PDF 1.2**

5.1 PDF files

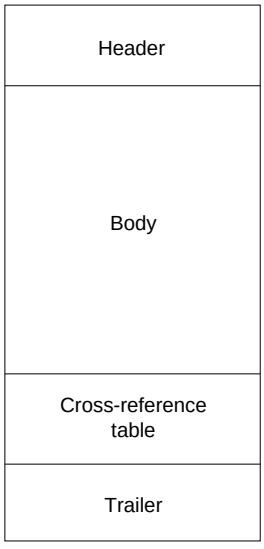
A canonical PDF file consists of four sections: a one-line header, a body, a cross-reference table, and a trailer. Figure 5.1 shows this structure:

```
<PDF file> ::=  <header>
                <body>
                <cross-reference table>
                <trailer>
```

In a PDF 1.0 file, all information is represented in 7-bit ASCII. Binary data must be encoded in ASCII; ASCII hexadecimal and ASCII base-85 are supported. No line in a PDF 1.0 file may be longer than 255 characters. A line in a file is delimited by a carriage return (<0D>), a linefeed (<0A>), or a carriage return followed by a linefeed. Updates may be appended to a PDF file, as described in Section 5.6, "Incremental update." **PDF 1.0**

Because the requirement to use ASCII does not guarantee file transmission transparency, and because it can cause a 20% expansion in the size of objects such as images that are naturally binary data, PDF 1.1 and subsequent versions relax the 7-bit ASCII requirement. PDF 1.1 allows files to contain binary data in strings, streams, and comments. In fact, experiments have shown that PDF files are less likely to be corrupted by system utilities if they *do* contain binary data. It is therefore recommended that the second line of a PDF file be a comment that contains at least four characters with codes 128 or greater. **PDF 1.1**

Figure 5.1 *Structure of a PDF file that has not been updated*



To accommodate binary data, the restriction on line length is also relaxed in PDF 1.1. PDF 1.1 files with binary data may have arbitrarily long lines. However, to increase compatibility with other applications that process PDF files, all lines that are not part of stream object data shall be no longer than 255 characters.

5.2 Header

The first line of a PDF file specifies the version number of the PDF specification to which the file adheres. The current version is 1.2; the first line of a 1.2-conforming PDF file should be **%PDF-1.2**. However, 1.0-conforming files and 1.1-conforming files are also 1.2-conforming files, so an application that understands PDF 1.2 also accepts a file that begins with either **%PDF-1.1** or **%PDF-1.0**.

<header> ::= <PDF version>

5.3 Body

The body of a PDF file consists of a sequence of indirect objects representing a document. The objects, which are of the basic types described in Chapter 4, represent components of the document such as fonts, pages, and sampled images.

Comments can appear anywhere in the body section of a PDF file. Comments have the same syntax as those in the PostScript language; they begin with a % character and may start at any point on a line. All text between the % character and the end of the line is treated as a comment. Occurrences of the % character within strings or streams are not treated as comments.

5.4 Cross-reference table

The cross-reference table contains information that permits random access to indirect objects in the file, so that the entire file need not be read to locate any particular object. For each indirect object in the file, the table contains a one-line entry describing the location of the object in the file.

A PDF file contains one cross-reference table, consisting of one or more *sections*. If no updates have been appended to the file, the cross-reference table contains a single section. One section is added each time updates are appended to the file.

The cross-reference section is the only part of a PDF file with a fixed format. This permits random access to entries in the cross-reference table. The section begins with a line containing the keyword **xref**. Following this line are one or more cross-reference subsections:

<cross-reference section> ::=
xref
<cross-reference subsection>+

Each subsection contains entries for a contiguous range of object numbers. The organization of the cross-reference section into subsections is useful for incremental updates, because 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. Each cross-reference subsection begins with a header line containing two numbers: the first object number in that subsection and the number of entries in the subsection. Following the header are the entries, one per line:

<cross-reference subsection> ::=
<object number of first entry in subsection>
<number of entries in subsection>
<cross-reference entry>+

Each entry is exactly 20 characters long, including the end-of-line marker. There are two formats for cross-reference table entries: one for objects that are in use and another for objects that have been deleted and so are free:

<cross-reference entry> ::=
<in-use entry> |
<free entry>

For an object that is in use, the entry contains a byte offset specifying the number of bytes from the beginning of the file to the beginning of the object, the generation number of the object, and the **n** keyword:

<in-use entry> ::= <byte offset> <generation number> **n**

The byte offset is a ten-digit number, padded with leading zeros if necessary. It is separated from the generation number by a single space. The generation number is a five-digit number, also padded with leading zeros if necessary. Following the

generation number is a single space and the **n** keyword. Following the keyword is the end-of-line sequence. If the end-of-line is a single character (either a carriage return or linefeed), it is preceded by a single space. If the end-of-line sequence is two characters (a carriage return followed by a linefeed), it is not preceded by a space.

For an object that is free, the entry contains the object number of the next free object, a generation number, and the **f** keyword:

<free entry> ::=
<object number of next free object>
<generation number> **f**

The entry has the same format as that for an object that is in use: a ten-digit object number, a space, a five-digit generation number, a space, the **f** keyword, and an end-of-line sequence.

The free objects in the cross-reference table form a linked list, with the entry for each free object containing the object number of the next free object. The first entry in the table (object number 0) is always free and has a generation number of 65535. It is the head of the linked list of free objects. The last free entry in the cross-reference table (the tail of the linked list) uses 0 as the object number of the next free object.

When an indirect object is deleted, its cross-reference entry is marked free, and the generation number in the entry is incremented by one to record the generation number to be used the next time an object with that object number is created. Each time the entry is reused, its generation number is incremented. The maximum generation number is 65535. Once that number is reached, that entry in the cross-reference table will not be reused.

Example 5.1 shows a cross-reference section containing a single subsection with six entries; four that are in use (object numbers 1, 2, 4, and 5) and two that are free (object numbers 0 and 3). Object number 3 has been deleted, and the next object created with an object number of 3 will be given the generation number of 7.

Example 5.1 *Cross-reference section with a single subsection*

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

Example 5.2 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 5.2 *Cross-reference section with multiple subsections*

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

Appendix A contains a more extensive example of the structure of a PDF file after several updates have been made to it.

5.5 Trailer

The trailer enables an application reading a PDF 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 a PDF file contains the end-of-file marker, **%%EOF**. The two preceding lines contain the keyword **startxref** and the byte offset from the beginning of the file to the beginning of the word **xref** in the last cross-reference section in the file. The *trailer dictionary* precedes this line.

The trailer dictionary, shown in Table 5.1, consists of the keyword **trailer** followed by a set of key–value pairs enclosed in double angle brackets:

<trailer> ::= **trailer**
<<
<trailer key–value pair>+
>>
startxref
<cross-reference table start address>
%%EOF

Table 5.1 *Trailer attributes*

Key	*Type*	*Semantics*
Size	integer	*(Required)* Total number of entries in the file's cross-reference table, including the original table and all updates.
Prev	integer	*(Present only if the file has more than one cross-reference section)* Byte offset from the beginning of file to the location of the previous cross-reference section. If the file has never been updated, it will not contain the **Prev** key.

Root	dictionary	*(Required; must be indirect reference)* Catalog object for the document, described on page 73.	
Info	dictionary	*(Optional; must be indirect reference)* Info dictionary for the document, described on page 110.	
ID	array	*(Optional)* An array of two strings, each of which is an ID. The first ID is established when the file is created and the second ID is changed each time the file is updated. IDs are described on page 113.	PDF 1.1
Encrypt	dictionary	*(Required if document is encrypted)* Information used to decrypt a document, described on page 114.	PDF 1.1

An example trailer for a file that has not been updated is shown in Example 5.3. The fact that the file has not been updated is determined from the absence of a **Prev** key in the trailer dictionary.

Example 5.3 *Trailer*

```
trailer
<<
/Size 22
/Root 2 0 R
/Info 1 0 R
>>
startxref
18799
%%EOF
```

5.6 Incremental update

The contents of a PDF file can be updated without rewriting the entire file. Changes can be appended to the end of the file, leaving completely intact the original contents of the file. When a PDF file is updated, any new or changed objects are appended, a cross-reference section is added, and a new trailer is inserted. The resulting file has the structure shown in Figure 5.2:

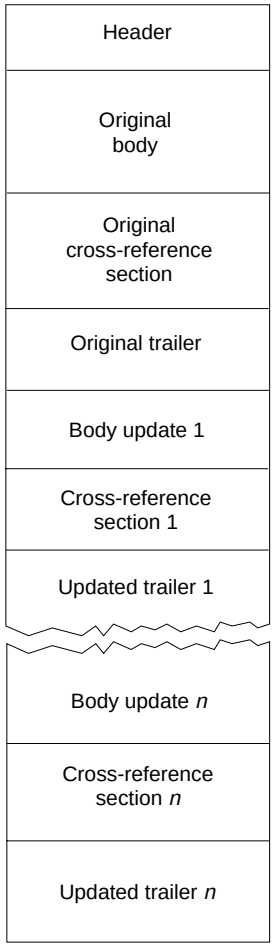
```
<Updated PDF file> ::=
                    <PDF file>
                    {<update>}*

<update> ::=        <body>
                    <cross-reference section>
                    <trailer>
```

A complete example of an updated file is shown in Appendix A.

The cross-reference section added when a PDF file is updated contains entries only for objects that have been changed, replaced, or deleted, plus the entry for object 0. Deleted objects are left unchanged in the file, but are marked as deleted in their cross-reference entries. The trailer that is added contains all the information in the previous trailer, as well as a **Prev** key specifying the location of the previous cross-reference section. As shown in Figure 5.2, after a file has been updated several times it contains several trailers, as well as several **%%EOF** lines.

Because updates are appended to PDF files, it is possible to end up with several copies of an object with the same object ID (object number and generation number) in a file. This occurs, for example, if a text annotation 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. Because it has been changed, however, an updated copy of the object is included in the update section added to the file. The cross-reference section added includes a pointer to this new changed version, overriding the information contained in the original cross-reference section. When a program such as Acrobat reads the file, it must build cross-reference information in such a way that the most recent version of an object is accessed in the file.

Figure 5.2 *Structure of a PDF file after changes have been appended several times*



5.7 Encryption

PDF 1.1

Documents can be encrypted to protect their contents from unauthorized access. Access to a protected document's contents is controlled by the security handler specified in the Encryption dictionary. The Encryption dictionary is the value of the **Encrypt** key in the trailer dictionary. Section 6.14, "Encryption dictionary," describes the Encryption dictionary and security handlers.

Strings and streams in a protected document are encrypted. Other data types (such as integers and Booleans) that are used primarily for structural information in a PDF file are not encrypted. This combination protects a document's contents, while allowing random access to the objects within a PDF file.

All strings and streams in a protected document, except those in the Encryption dictionary, are encrypted using the RC4 encryption algorithm. This prevents unauthorized users from simply removing the password from a PDF file to gain access to it. Strings in the Encryption dictionary are encrypted and decrypted by the security handler itself, using whatever encryption algorithm it chooses.

RC4 is a proprietary algorithm of RSA Data Security, Inc. Adobe Systems has licensed this algorithm for use in its products. Independent software vendors may be required to license this algorithm to develop software that encrypts or decrypts PDF documents. Further information can be obtained from RSA Data Security, Inc., 100 Marine Parkway, Redwood City, California, 95065.

Encryption details

Protection of data in a PDF file consists of two steps: computation of a key to be used to encrypt data, and encryption of the data. The key is simply a string of five bytes. (The key is restricted in length to five bytes (40 bits) to satisfy current U.S. cryptographic export requirements.) The key can be computed in any number of ways, more or less cryptographically secure, while encryption of data based on the key is always performed in the same way. The Encryption dictionary identifies the security handler that computes the key. PDF includes one built-in method for computing the key, called “Standard.” The remainder of this section explains how data is encrypted given a key. Section 6.12, “Encrypt dictionary,” discusses security handlers, including the Standard security handler.

Once the key for a document is computed, strings and streams are encrypted using Algorithm 5.3:

Algorithm 5.3 *Encrypting string and stream data*

1. Extend the key by five bytes using the string or stream’s object identifier. (See Section 4.10, “Indirect objects.”) If the string or stream is a direct object, the identifier of the indirect object containing it is used. An object identifier consists of an object number and a generation number. These are treated as binary integers. The low-order three bytes of the object number and the low-order two bytes of the generation number are concatenated to the key in that order, low-order byte first.

2. The resulting ten-byte string is used as input for the MD5 hash function.

3. The first ten bytes of the output of the MD5 function are used as input to the RC4 function, along with the string or stream data to be encrypted. The output of the RC4 function is the encrypted data that is stored in the PDF file.

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. The PDF encryption algorithm is also symmetric. Given a key, the three steps described above can also be used to decrypt data.

Stream data is encrypted after all stream encoding filters have been applied (and is decrypted before the stream decoding filters are applied). Decryption of strings, other than those in the Encryption dictionary, is done after escape-sequence processing and hex decoding as appropriate to the string representation described in Section 4.4, “Strings.”

CHAPTER 6

Document Structure

PDF provides an electronic representation of a document—a series of pages containing text, graphics, and images, along with other information such as thumbnails (miniature images of the pages), text annotations, hypertext links, and outline entries (also called *bookmarks*). Previous chapters lay the groundwork for understanding the PDF representation of a document, but do not describe the representation itself. Chapter 3 presents the coordinate systems that provide the supports on which the visible part of a PDF document depends. Chapter 4 explains the types of objects supported by PDF. Document components used in PDF are built from those objects. Chapter 5 describes the overall structure of a PDF file, which provides the framework necessary to organize the pieces of a document, move rapidly among the pages of a document, and update a document.

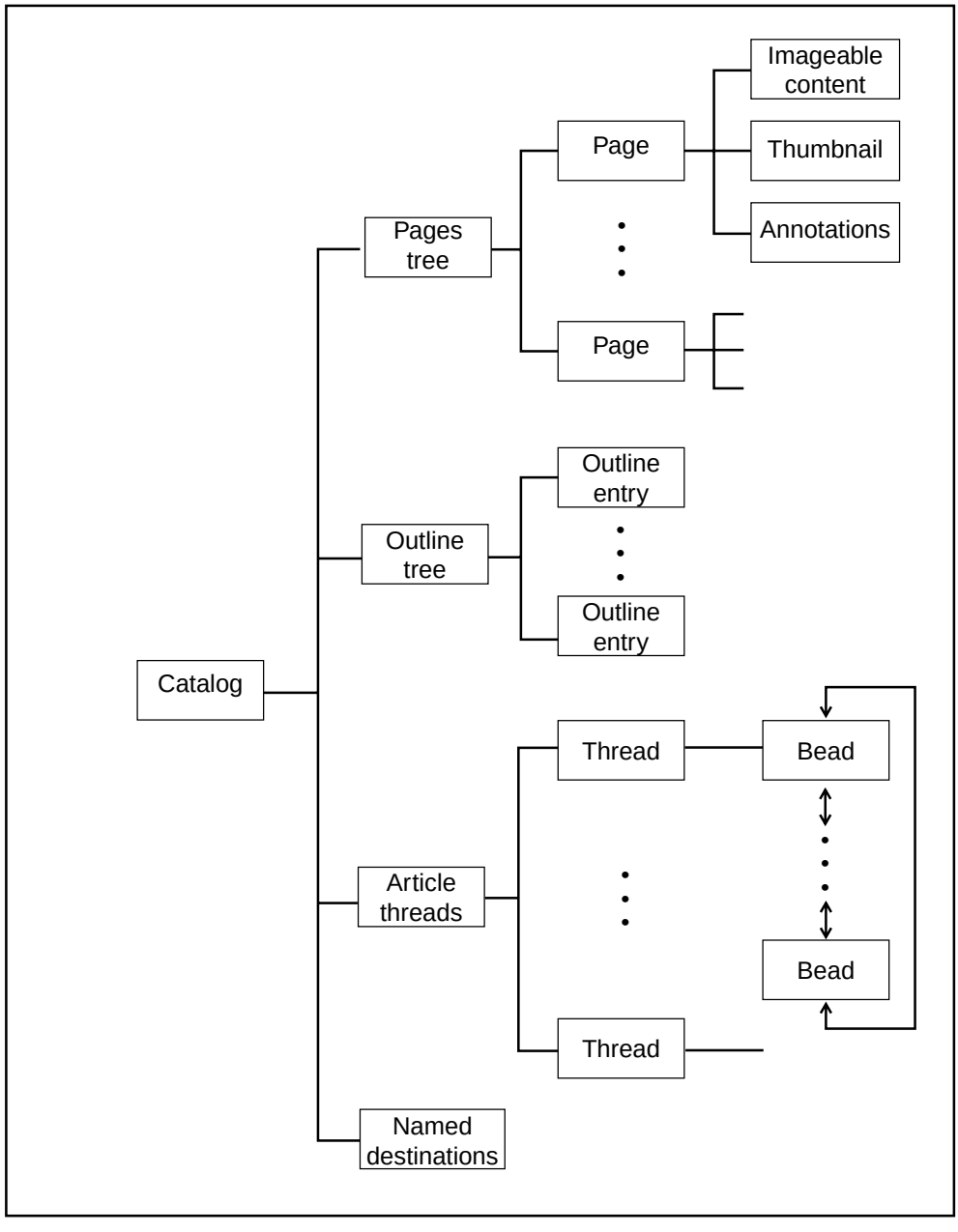
The body of a PDF file consists of a sequence of objects that collectively represent a PDF document. This chapter focuses exclusively on the contents of the body section of a PDF file and contains a description of each type of object that may be contained in a PDF document. Following each description is an example showing the object as it might appear in a PDF file. Complete example PDF files appear in Appendix A.

6.1 Introduction

A PDF document can be described as a hierarchy of objects contained in the body section of a PDF file. Figure 6.1 shows the structure of a PDF document. Most objects in this hierarchy are dictionaries. Parent, child, and sibling relationships are represented by key–value pairs whose values are indirect references to parent, child, or sibling objects. For example, the Catalog object, which is the root of the hierarchy, contains a Pages key whose value is an indirect reference to the object that is the root of the Pages tree.

Each page of the document includes references to its imageable contents, its thumbnail, and any annotations that appear on the page. The PDF file's standard trailer, described in Section 5.5, "Trailer," specifies the location of the Catalog object as the value of the trailer's **Root** key. In addition, the trailer specifies the location of the document's Info dictionary, a structure that contains general information about the document, as the value of the trailer's **Info** key.

Figure 6.1 *Structure of a PDF document*



Note *In many of the tables in this chapter, certain key–value pairs contain the notation "must be an indirect reference" or "indirect reference preferred." Unless one of these is specified in the description of the key–value pair, objects that are the value of a key can either be specified directly or using an indirect reference, as described in Section 4.11, "Object references."*

6.2 Catalog

The Catalog is a dictionary that is the root node of the document. It contains a reference to the tree of pages contained in the document, a reference to the tree of objects representing the document's outline, a reference to the document's article threads, and the list of named destinations. In addition, the Catalog indicates whether the document's outline or thumbnail page images should be displayed automatically when the document is viewed and whether some location other than the first page should be shown when the document is opened. Example 6.1 shows a sample Catalog object.

PDF 1.1

Example 6.1 *Catalog*

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

Table 6.1 shows the attributes for a Catalog.

Table 6.1 *Catalog attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Catalog**.
Pages	dictionary	*(Required, must be an indirect reference)* Pages object that is the root of the document's Pages tree.
Outlines	dictionary	*(Required if the document has an outline; must be an indirect reference)* The Outlines object that is the root of the document's outline tree, described on page 92.
PageMode	name	*(Optional)* How the document should appear when opened. Allowed values: **UseNone** Open document with neither outline nor thumbnails visible. **UseOutlines** Open document with outline visible. **UseThumbs** Open document with thumbnails visible. **FullScreen** Open document in full-screen mode. In full-screen mode, there is no menu bar, window controls, nor any other window present. The default value of **PageMode** is **UseNone**.
OpenAction	array or dictionary	*(Optional)* If the value of this key is an array, it must be a destination; see page 94. If it is a dictionary, it must be an action; see page 96. If this key is omitted, the top of the first page appears at the default zoom level. **PDF 1.1**
Threads	array	*(Required if the document has any threads; must be an indirect reference)* An array of threads as described on page 111. **PDF 1.1**
Dests	dictionary	*(Required in PDF 1.1 if the document has named destinations; must be an indirect reference)* A dictionary of names and corresponding destinations; see page 95. **PDF 1.1**
Names	dictionary	*(Optional)* A dictionary object that contains a list of various types of names and strings to be referenced within the document. See page 107. **PDF 1.2**
URI	dictionary	*(Optional)* Contains document-level information for Uniform Resource Identifier annotations; see page 103. **PDF 1.1**
AA (Additional Actions)	dictionary	*(Optional)* An additional-actions dictionary, providing defaults for the entire document. See page 97. **PDF 1.2**
ViewerPreferences	dictionary	*(Optional)* Specifies a dictionary that contains kiosk options for this document; see Table 6.2. If this key is omitted, viewers behave in accordance with any current user preferences. The name of the key reflects the fact that this dictionary is not part of the document structure itself, but represents a set of viewer-level options for displaying this document. A given viewer implementation may or may not support the options in this dictionary. **PDF 1.2**

The viewer preferences dictionary has the following attributes:

Table 6.2 *Viewer Preferences* **PDF 1.2**

HideToolbar	Boolean	*(Optional)* Specifies that the viewer's toolbar should be hidden whenever the document is active. This attribute defaults to *false*.
HideMenubar	Boolean	*(Optional)* Specifies that the viewer's menubar should be hidden whenever the document is active. This attribute defaults to *false*.
HideWindowUI	Boolean	*(Optional)* Specifies that the user interface elements in the document's window should be hidden. This attribute defaults to *false*.
FitWindow	Boolean	*(Optional)* Specifies that the viewer should resize the window displaying the document to fit the size of the first displayed page of the document. This attribute defaults to *false*.
CenterWindow	Boolean	*(Optional)* Specifies that the viewer should position the window displaying the document in the center of the computer's monitor. This attribute defaults to *false*.
PageLayout	name	*(Optional)* Specifies the layout for the page when the document is opened. If this attribute is not present, viewers behave in accordance with the current user preference. Allowed values: **SinglePage** — Display the pages 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.
NonFullScreenPageMode	name	*(Optional)* Specifies how the document should be displayed after exiting full-screen mode if the value of the **PageMode** key in the Catalog is **FullScreen**. This key is ignored if the value of the **PageMode** key in the Catalog is not **FullScreen**. Allowed values and semantics are the same as for the **PageMode** key in the Catalog, except that a value of **FullScreen** is not allowed.

6.3 Pages tree

The pages of a document are accessible through a tree of nodes known as the Pages tree. This tree defines the ordering of the pages in the document.

To optimize the performance of viewer applications, the Acrobat Distiller program and Acrobat PDF Writer construct balanced trees. (For further information on balanced trees, see reference [9] in the Bibliography on page 389.) The tree structure allows applications to quickly open a document containing thousands of pages using only limited memory. Applications should accept any sort of tree

structure as long as the nodes of the tree contain the keys described in Table 6.3. The simplest structure consists of a single Pages node that references all the page objects directly.

Note *The structure of the Pages tree for a document is unrelated to the content of the document. In a PDF file for a book, for example, there is no guarantee that a chapter is represented by a single node in the Pages tree. Applications that consume or produce PDF files are not required to preserve the existing structure of the Pages tree.*

The root and all interior nodes of the Pages tree are dictionaries, whose minimum contents are shown in Table 6.3.

Table 6.3 *Pages attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Pages**.
Kids	array	*(Required)* List of indirect references to the immediate children of this Pages node.
Count	integer	*(Required)* Specifies the number of leaf nodes (imageable pages) under this node. The leaf nodes do not have to be immediately below this node in the tree, but can be several levels deeper in the tree.
Parent	dictionary	*(Required; must be indirect reference)* Pages object that is the immediate ancestor of this Pages object. The root Pages object has no **Parent**.

Example 6.2 illustrates the Pages object for a document with three pages, while Appendix A contains an example showing the Pages tree for a document containing 62 pages.

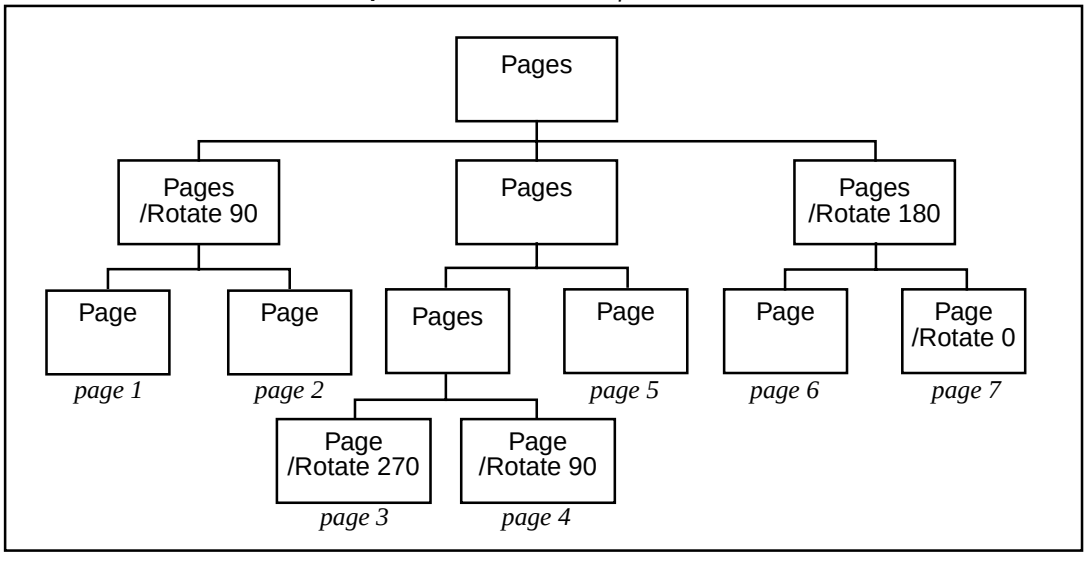
Example 6.2 *Pages tree for a document containing three pages*

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

6.3.1 Inheritance of attributes

A *Pages* object may contain additional keys that provide values for *Page* objects that are its descendants. Such values are said to be “inherited.” For example, a document may specify a **MediaBox** for all pages by defining one in the root Pages object. An individual page in the document could override the **MediaBox** in this example by specifying a **MediaBox** in the Page object for that page.

Example 6.3 *Inheritance of attributes*



Attributes that may be inherited are indicated in Table 6.4. If a *required* key that may be inherited is omitted from a Page object, then a value must be supplied in one of its ancestors. If an *optional* key that may be inherited is omitted, then a value may be supplied in one of its ancestors; barring that, the default value is used.

Example 6.3 demonstrates inheritance by showing a tree of Pages objects and Page objects. Pages 1, 2, and 4 are rotated 90º. Page 3 is rotated 270º. Pages 5 and 7 are not rotated (rotated 0º). Page 6 is rotated 180º.

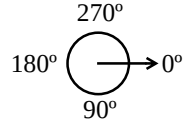
6.4 Page objects

A Page object is a dictionary whose keys describe a single page containing text, graphics, and images. A Page object is a leaf of the Pages tree, and has the attributes shown in Table 6.4.

Table 6.4 *Page attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Page**.
MediaBox	Rectangle	*(Required; may be inherited)* Rectangle specifying the “natural size” of the page, for example the dimensions of an A4 sheet of paper. The coordinates are measured in default user space units.
Parent	dictionary	*(Required; must be indirect reference)* The Pages object that is the immediate ancestor of this page.

Resources	dictionary	*(Required; may be inherited)* Resources required by this page, described in Chapter 7. If the page requires no resources, this value should be an empty dictionary, written as << >>. Omitting this value, or specifying a null value, indicates that the value is to be inherited from an ancestor Pages object.	
Contents	stream or array	*(Optional; must be indirect reference)* The page description (contents) for this page, described in Chapter 8. If **Contents** is an array of streams, they are concatenated to produce the page description. This allows a program that is creating a PDF file to create image objects and other resources as they occur, even though they interrupt the page description. If **Contents** is absent, the page is empty.	
	Note	*The arrangement of streams in the array is unrelated to the content on the page. Applications that consume or produce PDF files are not required to preserve the existing structure of a* **Contents** *array.*	
CropBox	Rectangle	*(Optional; may be inherited)* Rectangle specifying the region of the page displayed and printed.	
Rotate	integer	*(Optional; may be inherited)* Specifies the number of degrees the page should be rotated clockwise when it is displayed. This value must be zero (the default) or a multiple of 90.	
Thumb	stream	*(Optional; must be indirect reference)* Object that contains a thumbnail sketch of the page, described in Section 6.5, "Thumbnails."	
Annots	array	*(Optional)* An array of objects, each representing an annotation on the page, described in Section 6.6, "Annotations." Omit the **Annots** key if the page has no annotations.	
B (Beads)	array	*(Recommended if the page contains article beads)* An array whose elements are indirect references to each article bead on the page, in drawing order (the same order as the **Annots** array). Articles are described on page 111.	PDF 1.1
Dur (Duration)	real	*(Optional; may be inherited)* Specifies the "advance timing" (display duration) of a page. By default, the page does not advance automatically. See page 80.	PDF 1.1
Hid (Hidden)	Boolean	*(Optional; may be inherited)* If *true*, the page should be hidden (not displayed) during a presentation. The default is *false*. See page 80.	PDF 1.1
Trans (Transition)	dictionary	*(Optional; may be inherited)* A Transition dictionary, containing information about transitions between pages. See page 80.	PDF 1.1
AA	dictionary	*(Optional; may be inherited)* An additional-actions dictionary, providing defaults for the entire page. See page 97.	PDF 1.2



Note that some Page attributes may be inherited; see the note, "Inheritance of attributes," on page 76.

Note *The intersection between the page's media box and the crop box is the region of the default user space coordinate system that is viewed or printed. Typically, the crop box is located entirely inside the media box, so that the intersection is the same as the crop box itself.*

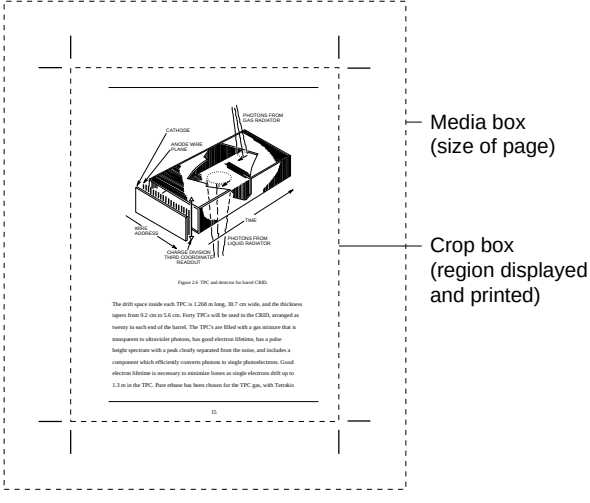
Figure 6.2 on page 79 shows the distinction between the media box and the crop box. In the figure, the crop box has been sized so that the crop marks do not appear when the page is viewed or printed.

Example 6.4 on page 79 shows a Page object with a thumbnail and two annotations. In addition, the Resources dictionary is specified as a direct object, and shows that the page makes use of three fonts, with the names F3, F5, and F7.

Example 6.4 *Page with thumbnail, annotations, and Resources dictionary*

```
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] >>
/Thumb 12 0 R
/Contents 14 0 R
/Annots [23 0 R 24 0 R]
>>
endobj
```

Figure 6.2 *Page object's media box and crop box*



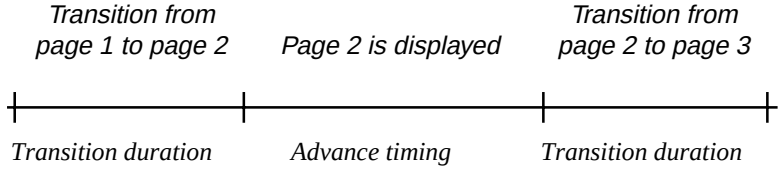
6.4.1 Presentation mode

A Page dictionary may contain three keys, **Dur**, **Hid**, and **Trans**, that provide information that is intended to be used when displaying a PDF document as a "presentation" or "slide show" and are otherwise ignored. A PDF viewer is not required to provide a presentation mode. If such a mode is provided by the viewer or a plug-in, however, then these keys define its behavior.

6.4.1.1 Duration

The **Dur** key in a Page dictionary specifies the *advance timing* of the page. The advance timing is intended to be used only when a presentation is being played in a non-interactive mode. It describes the maximum amount of time the page is displayed before the viewer automatically turns to the next page; the user can advance the page manually before the time is up. If no **Dur** key is specified for a Page object or any of its Pages ancestors, the page does not advance automatically.

The advance timing is defined as the amount of time between the *end* of the last transition and the *beginning* of the next one, as shown in the time-line below:



6.4.1.2 Hidden

The **Hid** (Hidden) key in a Page dictionary specifies that the page is not to be displayed during the presentation. If the user attempts to turn to a hidden page from the previous or following page during a presentation, the page is skipped and the next visible page is displayed. If the page is the destination of a link or thread, the Hidden attribute is ignored and the page *is* displayed.

The Hidden attribute of a page hides the page only during a presentation; other aspects of the user interface ignore the Hidden attribute.

6.4.1.3 Transition

The **Trans** key in a Page dictionary specifies a Transition dictionary, which describes the effect to use when going *to* that page, and the amount of time the transition should take. For example, a transition effect in the Transition dictionary of page two executes whenever the user goes to page two, regardless of the previous page. Table 6.5 defines keys for all Transition dictionaries; they may contain additional keys that control specific transition effects.

Table 6.5 *Transition attributes*

Key	*Type*	*Semantics*
Type	name	*(Optional)* Object type. Always **Trans**.
S (Subtype)	name	*(Optional)* Describes the transition effect. If this key is omitted, there is no transition effect to that page (the page is displayed normally), and the **D** key in the Transition dictionary is ignored. Transition effects are described in the following section.
D (Duration)	real	*(Optional)* The duration (in seconds) of the transition effect. The default duration is 1 second.

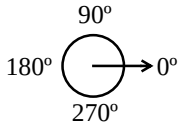
6.4.1.4 Transition effects

All implementations of presentation mode support the transition effects shown in Table 6.6. Some of these effects include optional parameters that control the appearance of the effect. The parameters are described in Table 6.7.

Table 6.6 *Transition Effects*

Effect	*Parameters*	*Description*
Split	**Dm**, **M**	Two lines sweep across the screen revealing the new page image. The lines can be either horizontal or vertical, as determined by the **Dm** key, and can move from the center out or from the edges in as determined by the **M** key.
Blinds	**Dm**	Multiple lines, evenly distributed across the screen, appear and synchronously sweep in the same direction to reveal the new page. The lines are either horizontal or vertical, as determined by the **Dm** key. Horizontal lines move down; vertical lines move to the right.
Box	**M**	A box sweeps from the center out or from the edges inward, as determined by the **M** key, revealing the new page image.
Wipe	**Di**	A single line sweeps across the screen from one edge to the other, revealing the new page image. Possible values for **Di** include 0, 90, 180, and 270.
Dissolve	*(none)*	The old page image “dissolves” in a piecemeal fashion to reveal the new page.
Glitter	**Di**	Similar to **Dissolve**, except the effect sweeps across the image in a wide band moving from one side of the screen to the other. Supported directions are 0, 270, and 315.
R	*(none)*	(Replace) The effect is simply to replace the old page with the new page; i.e., there is no “transition” per se. This is the default effect if the **S** key is omitted from the transition dictionary, but it may be explicitly specified as a way to override any default transition that may be used for full-screen mode.

Table 6.7 *Effect parameters*

Key	*Type*	*Semantics*
Di (Direction)	real	The direction of movement, specified in degrees, increasing in a counterclockwise direction. A value of 0 points to the right, indicating that the effect proceeds from left to right. A value of 90 points upward, indicating that the effect moves from bottom to top. 
	Note	*This is different from the page rotation, where the degrees increase in a clockwise direction.*
Dm (Dimension)	name	For those effects which can be performed either horizontally or vertically, the **Dm** key specifies which dimension to use. Possible values are **H** (horizontal) or **V** (vertical).
M (Motion)	name	For those effects which can be performed either from the center out or the edges in, the **M** key specifies which direction to use. Possible values are **I** (In) or **O** (Out).

Example 6.5 shows a page that, in presentation mode, would be displayed for 5 seconds before advancing to the following page. Before the page is displayed, there is a 3-second transition in which two vertical lines sweep across the screen, from the center outwards.

Example 6.5 *A page with information for presentation mode*

```
<<
/Type /Page
/Parent 4 0 R
/Contents 16 0 R
/Dur 5
/Trans << /S /Split /D 3.0 /M /O /Dm /V >>
>>
```

6.5 Thumbnails

A PDF document may include thumbnail sketches of its pages. They are not required, and even if some pages have them, others may not.

The thumbnail image for a page is the value of the **Thumb** key of the page object. The structure of a thumbnail is very similar to that of an Image resource (see Section 7.11.1 on page 176). The only difference between a thumbnail and an Image resource is that a thumbnail does not include **Type**, **Subtype**, and **Name** keys.

Note *Different pages in a document may have thumbnails with different numbers of bits per color component.*

Example 6.6 *Thumbnail*

```
12 0 obj
<<
/Filter [/ASCII85Decode /DCTDecode]
/Width 76
/Height 99
/BitsPerComponent 8
/ColorSpace /DeviceRGB
/Length 13 0 R
>>
stream
s4IA>!"M;*Ddm8XA,lT0!!3,S!/(=R!<E3%!<N<(!WrK*!WrN,!
… image data omitted …
endstream
endobj
13 0 obj
4298
endobj
```

6.6 Annotations

Annotations are notes or other objects that are associated with a page but are separate from the page description itself. PDF supports several kinds of annotations:

- Text notes (see page 88)
- Hypertext links (page 89)
- Movies (page 89)
- Sounds (page 92)

In the future, PDF may support additional types.

If a page includes annotations, they are stored in an array as the value of the **Annots** key of the Page object. Each annotation is a dictionary. As shown in Table 6.8, all annotations must provide a core set of keys, including **Type**, **Subtype**, and **Rect**. Certain other keys, indicating an annotation's color, title, modification date, border, and other information, are also defined for all annotations but are optional.

Note *All coordinates and measurements in text annotations, link annotations, and outline entries are specified in default user space units.*

As an alternative to the simple border and color characteristics defined in PDF 1.0 and 1.1, Annotations in PDF 1.2 can have one or more *appearances* attached to them. An appearance is a PDF Form XObject that is rendered inside the annotation bounding box.

Table 6.8 *Annotation attributes (common to all annotations)*

Key	*Type*	*Semantics*	
Type	name	(*Required in PDF 1.0, optional otherwise)* Object type. Always **Annot**.	
Subtype	name	*(Required)* Annotation subtype.	
Rect	Rectangle	*(Required)* Rectangle specifying the location of the annotation.	
Border	array	*(Optional)* In PDF 1.0, this is an array of three numbers, specifying the horizontal corner radius, the vertical corner radius, and the width of the border of the annotation. The default values are 0, 0, and 1, respectively. No border is drawn if the width is 0.	**PDF 1.0**
		The array may have a fourth element, a *dash array* that allows specification of solid and dashed borders. The dash array contains "on" and "off" stroke-lengths for drawing dashes, in the same format as the **setdash** marking operator, **d** (see page 215). An example of a border with a dash array is `[0 0 1 [3]]`.	**PDF 1.1**
	Note	*This key has been superseded in PDF 1.2 with the* **BS** *key, described below.*	
C (Color)	array	*(Optional)* The annotation color. For links, this is the border color. For text annotations, it is the background color of a closed annotation's icon, the title bar color of an active open annotation's window, and the window frame color of an inactive open annotation. A color is specified as an array of three numbers in the range 0 to 1, representing a color in **DeviceRGB** space.	**PDF 1.1**
T (Title)	string	*(Optional)* An arbitrary text label associated with the annotation. It is displayed in an active open text annotation's title bar and can be edited from the annotation's properties dialog. The characters in this string are encoded using the predefined encoding **PDFDocEncoding**, described in Appendix C.	**PDF 1.1**
M (ModDate)	Date	*(Optional)* The last time an annotation was modified. A text annotation's modification date is updated each time the text is changed. The ModDate must be a string; the preferred format described on page 133, but viewers should accept and display any string.	**PDF 1.1**
F (Flags)	integer	*(Optional)* Flags. The binary value of the integer is interpreted as a collection of flags that define various characteristics of the annotation. All undefined bits are reserved and must be set to 0. The default value for this key is 0.	**PDF 1.1**
	Note	*Bit-positions are numbered starting at 1, which is the least significant bit.*	
	bit 1	The *Invisible* flag specifies how an annotation is displayed when the corresponding annotation handler is not available. If this flag's value is 1, and the viewer does not provide a handler for the annotation's Subtype, the annotation is not displayed. If this flag's value is 0, and the viewer does not provide a handler for the annotation's Subtype, the annotation appears as an unknown annotation.	
	bit 2	The *Hidden* flag determines whether the annotation is currently shown. If the value of this flag is 1 then the annotation is hidden, in which case, there is no user interaction, display, or printing of the annotation. The ability to hide and show	**PDF 1.2**

annotations selectively, combined with appearances (see the **AP** key), is especially useful in cases where screen real estate is limited. They can then be used to display pop-up auxiliary data similar in function to online help systems.

	bit 3	The *Print* flag indicates whether the annotation should be printed. If the value of this flag is 1 then the annotation is printed. Corresponding to the definition of appearances for annotations, there may be some instances where the author of a document wishes to create an annotation for display purposes only. That is, the annotation should display its appearance while in the viewer but that appearance should not print. This is typical for annotations that act like push buttons whose presence on the printed page would be distracting.	PDF 1.2
H (Highlight)	name	*(Optional)* In PDF 1.1, the visual effect of clicking on a link annotation produced an inversion of the colors inside the bounding box of the annotation. In PDF 1.2, additional highlight modes, specified by this key, have been added for both Link and Widget annotations, as described in Table 6.10. The default for this key is **I** (Invert).	PDF 1.2
BS (Border Style)	dict	*(Optional)* This key overrides the **Border** key. The value of this key is a dictionary, defined on page 85, that specifies several attributes related to the border of the annotation.	PDF 1.2
AP	dictionary	*(Optional)* Appearance dictionary. Specifies that one or more appearances are available for this annotation. See page 87.	PDF 1.2
AS	name	*(Required if more than one appearance is possible)* Appearance state. If the **N**, **R**, or **D** keys in the **AP** dictionary are dictionaries instead of Form XObjects, then this key indicates which entry in the dictionary is to be used in each instance. This allows for the specification of items such as checkboxes and radio buttons whose appearance may change.	PDF 1.2

6.6.1 Annotation borders

PDF 1.2

The border is drawn completely inside the annotation's bounding box. If neither the **BS** key nor the **Border** key is present in the annotation dictionary, the border style is drawn as a solid line with a width of one point.

Table 6.9 *Border Style attributes*

Key	*Type*	*Semantics*
Type	name	*(Optional)* Object type. Always **Border**.
S (Subtype)	name	*(Optional)* One of the following names:
	S	(Solid) The border is drawn as a solid line. This is the default subtype.
	D	(Dashed) The border is drawn with a dashed line. The dash pattern is specified by the **D** attribute.

	B	(Beveled) The border is drawn in a beveled style (faux three-dimensional) such that it looks as if it is pushed out of the page (opposite of Inset).
	I	(Inset) The border is drawn in an inset style (faux three-dimensional) such that it looks as if it is inset into the page (opposite of Beveled).
	U	(Underlined) The border is drawn as a line on the bottom of the annotation rectangle.
		Other subtypes may be defined in the future.
W (Width)	real	*(Optional)* The border width in points. The default is 1 point.
D (Dash array)	array	*(Optional)* If the **Subtype** key is **D**, this array contains numbers representing on and off stroke lengths for drawing dashes, in the same format as the **setdash** marking operator, **d**. The default for this key is [3].

6.6.2 Annotation highlighting

PDF 1.2

An annotation is highlighted when any of the following events is triggered.

1. If the user clicks the mouse button down inside the active area of the annotation, the highlight is shown.
2. If the mouse button is down and the user moves the cursor out of the active area of the annotation, the highlight is removed.
3. If the mouse button is down and the user moves the cursor back into the active area of the annotation, the highlight is shown.
4. If the mouse button is released then the highlight is removed.

A highlight is *removed* by reverting the visual appearance of the document back to what it was before the highlight occurred.

Table 6.10 *Highlight Modes*

Name	*Semantics*
I (Invert)	The rectangle specified by the bounding box of the annotation is inverted.
N (None)	No highlighting is done.
O (Outline)	The border of the annotation is inverted. If no active color is defined, the border is inverted.
P (Push)	If the annotation specifies a Down appearance (i.e., if the Appearance dictionary defines a **D** key), that appearance is drawn. If the annotation does not specify a Down appearance then the region underneath the bounding box of the annotation is drawn inset into the page.

The use of the highlighting works in conjunction with the Down sub-appearance specified in the annotation appearance dictionary (if any) in the following manner: If the highlight value is not **P** (push), then the specified highlight mode is respected regardless of the fact that a down sub-appearance exists.

Note *The active behavior of highlighting is applicable only to annotations whose annotation handler is present. The highlighting behavior is logically the responsibility of the handler. Plug-in designers are encouraged to use the same PDF extensions for specifying highlighting for their annotations.*

6.6.3 Annotation appearances

PDF 1.2

The appearance dictionary defines the following three keys.

Table 6.11 *Appearance dictionary*

Key	*Type*	*Semantics*
N	dictionary or stream	*(Required)* Normal appearance. This is the appearance that is used when the annotation is in a normal state, that is, when the user is not interacting with the annotation. This is also the appearance that is used when printing the annotation.
R	dictionary or stream	*(Optional)* Rollover appearance. This is the appearance that is used when the user moves the mouse cursor inside the annotation. If the cursor leaves the annotation then the Normal appearance is re-displayed. The default is that no alternative appearance is displayed when the rollover occurs.
D	dictionary or stream	*(Optional)* Down appearance. This is the appearance that is used when the user clicks the mouse down inside the annotation. When the user releases the mouse, the viewer displays the Rollover appearance if that is defined, or the Normal appearance otherwise. The default is that no alternative appearance is displayed when the mouse-down event occurs.

If the **N, R,** or **D** values are dictionaries, then they contain one or more sub-appearances. The appearance state (**AS**) key in the annotation dictionary determines which sub-appearance to use as the current appearance.

Consider a checkbox with two states: checked and unchecked. The appearance dictionary, containing only a normal appearance (**N**), would look like this:

```
/AP <<
     /N <<
          /Checked Form XObject
          /Unchecked Form XObject
     >>
>>
```

The appearance state key would determine which appearance is the current normal appearance, For example, if the checkbox were currently checked, the value of the **AS** key would be the name **Checked**.

```
/AS /Checked
```

If the **N, R,** or **D** values are not dictionaries, then they must be Form XObjects (see page 179), which are represented as streams, and the appearance does not depend on the **AS** key.

The Form XObject's graphics state when rendering the appearance should have the origin translated to the lower left corner of the annotation bounding box and scaled to fit inside and clipped to the annotation bounding box. All other graphics state values are set to their defaults.

Some annotation subtypes defined in PDF (e.g., Movie) and all annotation subtypes defined by third parties are implemented through plug-ins. If the plug-in that implements a particular annotation subtype is not available, the viewer displays that annotation's Normal appearance.

Note *The viewer should attempt to provide reasonable behavior (e.g. display nothing) if the sub-appearance named by the* **AS** *key is not present in the* **N**, **R**, *or* **D** *dictionaries.*

6.6.4 Text annotations

A text annotation contains a string of text. When the annotation is open, the text is displayed. A PDF viewer application chooses the size and typeface of the text. Table 6.12 shows the contents of the text annotation dictionary. Example 6.7 shows a text annotation.

Table 6.12 *Text annotation attributes (in addition to those in Table 6.8)*

Key	*Type*	*Semantics*
Subtype	name	*(Required)* Annotation subtype. Always **Text**.
Contents	string	*(Required)* The text to be displayed. Text can be separated into paragraphs using carriage returns. The characters in this string are encoded using the predefined encoding **PDFDocEncoding**, described in Appendix C.
Open	Boolean	*(Optional)* If *true*, specifies that the annotation should initially be displayed opened. The default is *false* (closed).

Example 6.7 *Text annotation*

```
22 0 obj
<<
/Type /Annot
/Subtype /Text
/Rect [266 116 430 204]
/Contents (text for two)
>>
endobj
```

6.6.5 Link annotations

A link annotation, when activated, displays a *destination* or performs an *action*. A destination is a view of another location, possibly on a different page, with a different zoom factor, or in a different file. Table 6.13 shows the contents of the link annotation dictionary.

Table 6.13 *Link annotation attributes (in addition to those in Table 6.8)*

Key	*Type*	*Semantics*
Subtype	name	*(Required)* Annotation subtype. Always **Link**.
Dest	array or name	*(Required unless the* **A** *key is present)* The view to go to, represented as a destination. See page 94.
A (Action)	dictionary	*(Required unless the* **Dest** *key is present)* The action to be performed on activating this link annotation. See page 96. **PDF 1.1**

Example 6.8 *Link annotation*

```
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
```

6.6.6 Movie Player annotations **PDF 1.2**

A Movie Player annotation describes the static display and playing of movies and sounds within PDF documents. These annotations appear to be embedded in the document, like links. The activation area may be invisible, bordered in the manner of a link button, and it may have the movie's Poster frame displayed. There are several authoring options that control the way a movie is to be displayed and played.

Table 6.14 *Movie Player annotation attributes (in addition to those in Table 6.8)*

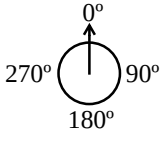
Key	*Type*	*Semantics*
Subtype	name	*(Required)* Annotation subtype. Always **Movie**.
Movie	dictionary	*(Required)* A description of the static characteristics of the Movie; see Table 6.15.
A (Activation)	Boolean	*(Optional)* A flag that indicates whether the movie should be shown by clicking in the annotation rectangle. Possible values are:

	false	Do not play the movie when clicked.
	true	Play the movie with the default activation values. (This is the default value for the **A** key.)
or		
dictionary	*(Optional)* Directions for playing the Movie; see Table 6.16.	

The Movie dictionary contains information needed to locate the movie data and to display the poster (if requested) in the annotation rectangle:

Table 6.15 *Movie dictionary attributes*

Key	*Type*	*Semantics*
F (File)	File specification	*(Required)* 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)* If the movie is visible, the horizontal and vertical sizes of the movie's bounding box in pixels: [*horiz vert*]. An "invisible movie" is one with no video: it has only sound.
Rotate	integer	*(Optional)* Specifies the number of degrees clockwise the movie must be rotated, relative to the rotation of the page. This value must be a multiple of 90. The default is 0.
Poster	Boolean	*(Optional)* A flag indicating whether the poster is to be retrieved from the movie file for display. Possible values are: **false** Do not show a poster image. (This is the default if the **Poster** key is omitted.) **true** Show the poster image from the movie file.
or		
	stream	*(Optional)* An Image resource (see Section 7.11.1 on page 176) that is to be displayed as the poster.



The Movie Activation dictionary contains information needed to control the dynamics of playing the movie:

Table 6.16 *Movie Activation attributes*

Key	*Type*	*Semantics*
ShowControls	Boolean	*(Optional)* If this key is *true*, a Movie Controller bar is shown when the movie is being played. The default is *false*.
Mode	name	*(Optional)* The playing mode for the movie. Currently defined values are: **Once** Show the movie once and stop.

Open	Show the movie and leave the controller open.
Repeat	Repeat the movie from the beginning until stopped.
Palindrome	Play the movie back and forth until stopped.

If Mode is omitted or unrecognized, the mode is **Once**.

Synchronous Boolean — *(Optional)* If this key is *true*, the player does not return to Acrobat until the movie is completed or dismissed by the user. The default is *false*.

Start

number, string, or array — *(Optional)* The starting time of the movie segment to play. If omitted, the movie is played from the beginning.

Movie time values are based on 64-bit integers. If the start time is representable in 32 bits, the key value should be an integer. If not, the key value should be an 8-byte string, with the most significant byte of the value first, that is treated as a 64-bit integer. If the time scale for the starting time is not the same as the Movie time scale, the start time is represented as an array of two values: the first element is the time value (integer or string), and the second element is the time scale (integer). The time scale is measured in events per second.

Duration

number, string, or array — *(Optional)* The duration of the movie segment to play. If omitted, the movie is played to the end.

Movie time values are based on 64-bit integers. If the duration is representable in 32 bits, the key value should be an integer. If not, the key value should be an 8-byte string, with the most significant byte of the value first, that is treated as a 64-bit integer. If the time scale for the duration is not the same as the Movie time scale, the duration is represented as an array of two values: the first element is the time value (integer or string), and the second element is the time scale (integer). The time scale is measured in events per second. A negative duration means the movie is to be played backwards.

Rate number — *(Optional)* The initial speed at which the movie is played. The default speed is 1.0. A negative speed means the movie is to be played backwards with respect to **Start** and **Duration**.

Volume number — *(Optional)* The initial volume setting for the movie. This number must be between -1 and 1. Negative settings are muted. The default volume is 1.

FWScale array — *(Optional)* For floating play windows, the magnification at which to play the movie. **FWScale** is an array of two integers: [*numerator denominator*] representing a rational magnification factor for the movie. The final window size for the movie is (*numerator* ÷ *denominator*) × **Aspect** *pixels*. The presence of the **FWScale** key implies that the movie is to be played in a floating window. The absence of the **FWScale** key implies that the movie is to be played in the annotation rectangle.

FWPosition	array	*(Optional)* For floating play windows, the position within the screen at which to play the movie. **FWPosition** is an array of two numbers: [*horizontal vertical*] representing a relative position of the movie window with respect to the left and top of the screen. Each number must be in the range 0 to 1, with [0.5 0.5] meaning "center the movie window on the screen."

6.6.7 Sound annotations

PDF 1.2

A Sound annotation is analogous to a Text annotation, except that it contains sound, recorded from the computer's microphone or imported from a file. The annotation behaves like a Text annotation in most ways, with a different icon (a speaker) to indicate that it is a Sound annotation.

Table 6.17 *Sound annotation attributes (in addition to those in Table 6.8)*

Key	*Type*	*Semantics*
Subtype	name	*(Required)* Annotation subtype. Always **Sound**.
Sound	stream or File specification	*(Required)* A Sound object (stream) or an external sound file (file specification). See Section 6.16 on page 131.

6.7 Outline tree

An outline allows a user to access views of a document by name. As with a link annotation, activation of an outline entry (also called a *bookmark*) brings up a new view based on the destination description. Outline entries form a hierarchy of elements. An entry may be one of several at the same level in the outline, it may be a sub-entry of another entry, and it may have its own set of child entries. An outline entry may be *open* or *closed*. If it is open, its immediate children are visible when the outline is displayed. If it is closed, they are not.

If a document includes an outline, it is accessed from the **Outlines** key in the Catalog object. The value of this key is the Outlines object, which is the root of the outline tree. The contents of the Outlines dictionary appear in Table 6.18 and Example 6.9. The top-level outline entries are contained in a linked list, with **First** pointing to the head of the list and **Last** pointing to the tail of the list. When displayed, outline entries appear in the order in which they occur in the linked list.

Table 6.18 *Outlines attributes*

Key	*Type*	*Semantics*
Count	integer	(*Required if document has any open outline entries, otherwise optional*) Total number of open entries in the outline. This includes the total number of items open at all outline levels, not just top-level outline entries. If the count is zero, this key should be omitted.

First	dictionary	(*Required if document has any outline entries; must be indirect reference*) Reference to the outline entry that is the head of the linked list of top-level outline entries.
Last	dictionary	(*Required if document has any outline entries; must be indirect reference*) Reference to the outline entry that is the tail of the linked list of top-level outline entries.

Example 6.9 *Outlines object with six open entries*

```
21 0 obj
<<
/Count 6
/First 22 0 R
/Last 29 0 R
>>
endobj
```

Each outline entry is a dictionary, whose contents are shown in Table 6.19.

Table 6.19 *Outline entry attributes*

Key	*Type*	*Semantics*
Title	string	*(Required)* The text that appears in the outline for this entry. The characters in this string are encoded using the predefined encoding **PDFDocEncoding**, described in Appendix C.
Dest	array or name	*(Required unless the* **A** *key is present)* A destination, as described on page 94.
A (Action)	dictionary	*(Required unless the* **Dest** *key is present)* The action to be performed when this bookmark is activated; see page 96. **PDF 1.1**
AA (Additional Actions)	dictionary	*(Optional; may be inherited)* See page 97.
Parent	dictionary	*(Required; must be indirect reference)* Specifies the entry for which the current entry is a sub-entry. The parent of the top-level entries is the Outlines object.
Prev	dictionary	*(Required if the entry is not the first of several entries at the same outline level; must be indirect reference)* Specifies the previous entry in the linked list of outline entries at this level.
Next	dictionary	*(Required if the entry is not the last of several entries at the same outline level; must be indirect reference)* Specifies the next entry in the linked list of outline entries at this level.
First	dictionary	*(Required if an entry has sub-entries; must be indirect reference)* Specifies the outline entry that is the head of the linked list of sub-entries of this outline item.

Last	dictionary	*(Required if an entry has sub-entries; must be indirect reference)* Specifies the outline entry that is the tail of the linked list of sub-entries of this outline item.
Count	integer	*(Required if an entry has sub-entries)* If positive, specifies the number of open descendants the entry has. This includes not just immediate sub-entries, but sub-entries of those entries, and so on. If the value is negative, the entry is closed, and the absolute value of **Count** specifies how many entries would appear if the entry were reopened. If an entry has no descendants, the **Count** key is omitted.

Example 6.10 shows an outline entry. An example of a complete outline tree can be found in Appendix A.

Example 6.10 *Outline entry*

```
22 0 obj
<<
/Parent 21 0 R
/Dest [3 0 R /Top 0 792 0]
/Title (Document)
/Next 29 0 R
/First 25 0 R
/Last 28 0 R
/Count 4
>>
endobj
```

6.8 Destinations

An annotation or Outline entry may specify a *destination*, which consists of a page, the location of the display window on that page, and the zoom factor to use when displaying that page.

A destination may be represented explicitly as an array, or implicitly through a name; see page 95.

6.8.1 Explicit destinations

Table 6.20 shows the allowed forms of the destination. In the table, *top*, *left*, *right*, and *bottom* are numbers specified in the default user space coordinate system. *page* is an indirect reference to the destination Page object, except in the case of the GoToR action, where it is a page number. The page's *bounding box* is the smallest rectangle enclosing all objects on the page. No side of the bounding box is permitted to be outside the page's crop box. If it is, that side of the bounding box is defined by the corresponding side of the crop box.

Table 6.20 *Destination specification*

Value of **Dest** *key*	*Semantics*
[*page* /XYZ *left top zoom*]	If *left*, *top*, or *zoom* is null, the current value of that parameter is retained. For example, a destination of `[4 0 R null null null]` specifies the page object with an object ID of 4, retaining the same *top*, *left*, and *zoom* as the current page. A zoom of 0 has the same meaning as a zoom of null.
[*page* /Fit]	Fit the page to the window.
[*page* /FitH *top*]	Fit the width of the page to the window. *top* specifies the *y*-coordinate of the top edge of the window.
[*page* /FitV *left*]	Fit the height of the page to the window. *left* specifies the *x*-coordinate of the left edge of the window.
[*page* /FitR *left bottom right top*]	Fit the rectangle specified by *left bottom right top* in the window. If the height (*top* – *bottom*) and width (*right* – *left*) imply different zoom factors, the numerically smaller zoom factor is used, to ensure that the specified rectangle fits in the window.
[*page* /FitB]	Fit the page's bounding box to the window. **PDF 1.1**
[*page* /FitBH *top*]	Fit the width of the page's bounding box to the window. *top* specifies the *y*-coordinate of the top edge of the window. **PDF 1.1**
[*page* /FitBV *left*]	Fit the height of the page's bounding box to the window. *left* specifies the *x*-coordinate of the left edge of the window. **PDF 1.1**

6.8.2 Named destinations **PDF 1.1**

A destination may be represented implicitly, using a *string* or a *name*. Both of these cases are referred to as "named destinations." These are especially useful when the destination is in another file. For example, one file may contain a link to the first page of Chapter 6 in another file. If the link uses a name such as

```
/Chap6.begin
```

rather than an explicit location, such as a certain rectangle on page 42, then the place where Chapter 6 starts can change without invalidating the link.

The mapping from *strings* to destinations is a feature of PDF 1.2. The Catalog of any document may contain a **Names** key whose value is a dictionary. Each value in this dictionary is a tree, similar to the Pages tree. The leaf-nodes in each such tree contain pairs of strings and indirect objects. The **Dests** entry in this dictionary is a tree where the indirect objects are destinations. The structure of this tree is described in Section 6.10, "Name trees". **PDF 1.2**

The mapping from *names* to destinations is a feature of PDF 1.1, and it is supported in PDF 1.2 for backwards compatibility. The Catalog of any document may contain a **Dests** key. Each key in this dictionary is a name, and the corresponding value is either a destination or a dictionary. If it is a dictionary, it must have a **D** key whose value is a destination. (The dictionary enables named destinations to have additional attributes.)

Note *There are several performance advantages to using strings instead of names for named destinations. See page 275.*

If an annotation or Outline entry that contains a named destination also includes a file specification (**F** key), then the destination is in the specified file. Otherwise, if there is no **F** key, the destination is in the current file.

6.9 Actions

PDF 1.1

Beginning in PDF 1.1, it is possible to specify an *action* to be performed when a Link annotation or Outline entry is activated, or when a document is opened. PDF defines several subtypes of actions:

Table 6.21 *Types of actions*

Action type	*Description*	
GoTo	Change the current page view to a specified page and zoom factor. See page 99.	
GoToR	("GoTo Remote") Open another PDF file at a specified page and zoom factor. See page 100	
Launch	Launch an application, usually to open a file. See page 100.	
Thread	Begin reading an article thread. See page 111.	
URI	Resolve the specified Uniform Resource Identifier (URI). See page 102.	
Sound	Play a sound. See page 104.	PDF 1.2
Movie	Play a movie. See page 104	PDF 1.2
SetState	Store a value in the appearance-state (**AS** key) of an annotation's appearance dictionary. See page 87.	PDF 1.2
Hide	Set or clear the Hidden flag for an annotation. See page 106.	PDF 1.2
Named actions	Execute an action predefined by the viewer. See page 106.	PDF 1.2
SubmitForm	Send data to a URL. See page 129.	PDF 1.2
ResetForm	Set field values to their defaults. See page 130.	PDF 1.2
ImportData	Import field values from a file. See page 130	PDF 1.2

Note *It is intended that plug-in extensions may add new actions, as described in Appendix G.*

An action is represented as a dictionary. Every action must contain an **S** (Subtype) key. Other keys may be present, depending on the action type.

Table 6.22 *Action attributes (common to all actions)*

Key	*Type*	*Semantics*	
Type	name	*(Optional)* Object type. Always **Action**.	
S (Subtype)	name	*(Required)* Action subtype.	
Next	dictionary or array	*(Optional)* The action, or sequence of actions, to be performed after the current action.	**PDF 1.2**

In PDF 1.1, only single actions can be triggered as the result of a user action. PDF 1.2 allows actions to be chained together to provide a richer action model. For example, if a user clicks on a particular link, it might play a sound, then go to a new page, and then start up a movie.

The value of the **Next** key is either an action or an array of actions, each of which may in turn contain a **Next** key. Note that this data structure allows the expression of the actions as a tree. Actions are executed in the following order:

1. The current action is performed.
2. If the current action has a **Next** key whose value is an action, that action is executed.
3. If the current action has a **Next** key whose value is an array of actions, then each action in the array is executed, in order.

The application should attempt to provide reasonable behavior in extreme situations: actions that are self-referential should not be subsequently executed; actions that close the document or otherwise render the execution of the next action impossible should terminate the execution of the action tree. The application should ensure that a sequence of actions is interruptible by the user.

Note *During its execution, an action should not modify its dictionary or the action tree in which it resides. The effect of such modification on subsequent execution of actions in the tree is undefined.*

6.9.1 Action Trigger Points

PDF 1.2

In PDF 1.1, the presence of an **A** key or **Dest** key in an annotation or Outline entry denotes an action that is to be performed when the mouse button is released after clicking inside the annotation or Outline entry. PDF 1.2 provides a more general mechanism by defining other “trigger points” (events) and associating actions with

each one by means of an “additional actions” dictionary, which is included in an annotation or Outline entry as the value of the **AA** key. **AA** keys are also permitted in Page objects and in a document’s Catalog.

Note *The term “mouse” is used to represent a generic pointing device available for use inside a viewing application. It is assumed that the pointing device has the following characteristics:*

- *A selection button that may be activated, held, and released. This is often the (left) mouse button.*
- *A notion of location, that is, an indication of where the device is pointing. In many systems, this is usually denoted by the cursor.*
- *A notion of focus, that is, which element in the document is currently interacting with the user. In many systems, this is usually denoted by a blinking caret, a focus rectangle, or a color change.*

The viewing application and any accessibility programs must ensure that a mapping exists into this environment in order for these actions to be executed correctly.

Table 6.23 *Additional Actions attributes*

Key	*Type*	*Semantics*
E (Enter)	dictionary	*(Optional)* The action that is executed when the cursor enters the activation area.
X (Exit)	dictionary	*(Optional)* The action that is executed when the cursor exits the activation area.
D (Down)	dictionary	*(Optional)* The action that is executed when the user depresses the mouse button inside the activation area.
U (Up)	dictionary	*(Optional)* The action that is executed when the user releases the mouse button inside the activation area. For backwards compatibility, the **A** key in an annotation dictionary takes precedence over the **U** key in this dictionary.
O (Open)	dictionary	*(Optional. Defined only for Page objects.)* The action that is executed after the page has finished being drawn. This is independent of any **OpenAction** that may be defined in the Catalog (see Table 6.1 on page 74), and would be executed after such an action.
C (Close)	dictionary	*(Optional. Defined only for Page objects.)* The action that is executed when the page is no longer being displayed (e.g., when the user goes to the next page or follows a link). This action applies to the page being closed, and it executed before any other page is opened.
FP (first page)	dictionary	*(Optional)* The action that is executed when the user goes to the first page of the document.
PP (prev. page)	dictionary	*(Optional)* The action that is executed when the user goes to the previous page of the document.

NP (next page)	dictionary	*(Optional)* The action that is executed when the user goes to the next page of the document.
LP (last page)	dictionary	*(Optional)* The action that is executed when the user goes to the last page of the document.

Trigger points for mouse-events are defined with the following constraints:

- An Enter event cannot occur unless the mouse button is already up.
- An Up event cannot occur without an Enter event and a Down event occurring.
- An Exit cannot occur without an Enter occurring.
- In the case of overlapping or nested annotations, entering a second annotation's activation area causes an Exit of the first annotation's area to occur.

Annotations inherit trigger points (**AA** dictionaries) from the pages on which they appear and from the document. That is, if an annotation does not define actions for the **E**, **X**, **D**, or **U** keys, the definitions for the Page object, if any, are used; failing that, the definitions in the Catalog are used. An exception is made for the **U** key if the annotation has an **A** key, which takes precedence.

Likewise, pages and outline entries inherit trigger points from the document. That is, if a Page or Outline object does not define actions for the **O** or **C** keys, the definitions in the Catalog, if any, are used.

Inheritance of trigger points enables an author to define actions that apply to large groups of annotations.

Note *The Enter and Exit triggers are determined by the annotation's handler and may not correspond to the annotation's bounding box. Annotation handlers can implement non-rectangular activation regions.*

6.9.2 GoTo action

A GoTo action has the same effect as specifying a destination (with a **Dest** key) in a Link annotation, but it is less compact and is not compatible with PDF 1.0. Destinations are preferred over GoTo actions.

Table 6.24 *GoTo action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **GoTo**.
D (Dest)	array, string, or name	*(Required)* The destination, as described in Table 6.20 on page 95.

Example 6.11 *GoTo action*

```
42 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
```

Note *This example has the same effect as the Link annotation shown in Example 6.8 on page 89, which uses a destination (a* **Dest** *key).*

6.9.3 GoToR action

The GoToR action is similar to the GoTo action. However, it includes an additional parameter, the **F** key, that specifies the PDF file that contains the action's destination.

Table 6.25 *GoToR action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **GoToR**.
D (Dest)	array	*(Required)* An explicit destination, represented by an array, as described in Table 6.20 on page 95, except that the destination page (the first element of the array) must be specified by a *page number*, not by an indirect reference to a Page object. The first page is 0.
	or	
	string or name	*(Required)* The name of a destination. See Section 6.8.2 on page 95.
F (File)	File specification	*(Required)* The file containing the destination.
NewWindow	Boolean	*(Optional)* If the value is *true*, then the destination document will be opened in a new window; if it is *false*, then the destination document will be opened in the same window as the source document. If this attribute is omitted, the viewer will behave in accordance with the current user preference. **PDF 1.2**

6.9.4 Launch action

The Launch action specifies an application to launch or a document to open. The action must specify the application or document as a file, using the **F** key.

PDF also allows platform-specific information to be included in the Launch dictionary where that information is needed for specific platform. The key **Win** is used for information related to Microsoft Windows launches; the key **Unix** is used for information related to UNIX system launches. If there is no platform specific key, then the **F** key is used.

Table 6.26 *Launch action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **Launch**.
F (File)	File specification	*(Required if there is no alternative key)* The file to use in performing the specified action. A viewer that encounters an action with no **F** key and for which it does not understand any of the alternative keys does nothing.
Win	dictionary	*(Optional)* Windows-specific launch parameters as described in Table 6.27.
Unix	*undefined*	*(Optional)* Not yet defined.
NewWindow	Boolean	*(Optional)* If the destination is a PDF document, then this switch applies. If the value is *true*, the document is opened in a new window; if it is *false*, then the document is opened in the same window as the source document. If this attribute is omitted, the viewer behaves in accordance with the current user preference. **PDF 1.2**

Table 6.27 *Windows-specific launch attributes*

Key	*Type*	*Semantics*
F (File)	File specification	*(Required)* The document or application to launch, specified as a DOS file name using standard DOS syntax. If the string includes a backslash (\ ), the backslash must itself be preceded by a backslash.
O (Operation)	string	*(Optional)* The operation to perform: `(open)` or `(print)`. `(open)` is the default. If the **F** key specifies an application, this key is ignored and the application is launched.
P (Parameters)	string	*(Optional)* The parameters passed to the application specified by the **F** key. If the **F** key specifies a document, this key should not be provided.
D (Directory)	string	*(Optional)* The default directory, specified using standard DOS syntax.

6.9.5 Thread action

When a viewer performs a Thread action, it goes to the specified thread and enters thread mode. The thread need not be in the current PDF file.

Table 6.28 *Thread action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **Thread**.
F (File)	File specification	*(Optional)* If this key is omitted, then the thread is in the current file. Otherwise this key specifies an external file containing the thread.
D (Dest)		*(Required)* The desired thread destination. One of the following forms must be provided:
	dictionary	An indirect reference to a thread. This form of the destination requires that the thread be in the current file. (See Section 6.12, "Articles.")
	number	The index of a thread. (The index of the first thread in a document is 0.)
	string	The title of the thread. If more than one thread has the same title, the first thread in the document's list of threads with that title is chosen.
B (Bead)		*(Optional)* The desired bead in the destination thread. One of the following forms may be provided:
	dictionary	An indirect reference to a Bead dictionary. This form of the bead requires that the thread be in the current file. See Table 6.44 on page 112.
	number	A number that specifies the bead's index in the thread. (The index of the first bead in a thread is 0.)

6.9.6 URI action

PDF 1.1

A Uniform Resource Identifier (URI) is a string that identifies 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. In PDF, activating a URI action causes the URI to be resolved.

Note *The URI action is resolved by the Acrobat WebLink plug-in.*

Table 6.29 *URI action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **URI**.
URI	string	*(Required)* The Uniform Resource Identifier to resolve, encoded in 7-bit ASCII.
IsMap	Boolean	*(Optional)* If this key is *true*, the mouse position should be tracked when link is activated.

In a URI string, any characters following a # define a *fragment identifier.*

<fragmentID> ::= <handler>{=<data>}

handler indicates a service, typically provided by a plug-in, that makes use of the resolved URI, and *data* (the characters following an equal sign) provides additional information for the handler. The only handler defined by PDF is **nameddest**. The data is interpreted as the name of a destination; such a URI action is similar to a GoToR action that uses a named destination. If there is no equal sign, then the entire fragment identifier is interpreted as data for the **nameddest** handler; i.e., it is a named destination. For example, the following two examples are equivalent; the first one is preferred.

```
<< /Type /Action /S /URI
/URI (http://www.adobe.com/techdocs/PDF1.2Man\
ual.pdf#nameddest=Chap2.Section3)
>>

<< /Type /Action /S /URI
/URI (http://www.adobe.com/techdocs/PDF1.2Man\
ual.pdf#Chap2.Section3)
>>
```

The syntax for *handler* is similar to a PDF name object (see page 45) except that '=' is not permitted (because it separates the handler from the data). Furthermore, PDF names allow characters that are not allowed in URI strings. To use such a character in a fragment identifier, in either the handler or the data, write its two hex-digit character code, preceded by a percent sign. The name **X&Y**, for example, would be written as `X%26Y`.

A URI action's **IsMap** attribute indicates that when the action is performed, the (*x*, *y*) position of the mouse within the parent link annotation (relative to the upper left hand corner of the link rectangle) should be concatenated to the end of the URI, preceded by a question mark. Here is an example:

```
http://www.adobe.com/intro?100,200
```

Suppose the bounding rectangle in user space of the Link annotation (the value of the **Rect** key) is $[\,ll_x\ ll_y\ ur_x\ ur_y\,]$. Given the coordinates of the mouse position in device space, (x_d, y_d), transform the mouse coordinates to user space, (x_u, y_u). The final coordinates, (x, y), are obtained in this way:

$$x = x_u - ll_x$$
$$y = y_u - ur_y$$

Because these coordinates can be fractional and the **IsMap** attribute requires integers, the final coordinates should be rounded to the nearest integer.

6.9.6.1 URI dictionary in the Catalog

In order to support URI action types, the Catalog of the PDF file may include a URI dictionary.

Table 6.30 *URI attributes*

Key	*Type*	*Semantics*
Base	string	*(Optional)* Base URI to resolve relative references. This element allows the URI of the document itself to be recorded in situations in which the document may be accessed out of context. URI actions within the document may be in a "partial" form relative to this base address. When the base address is not specified, the URI is assumed to be the one originally used to locate the document. For example, if a document has been moved but the documents pointed to by relative links within the document have not, the **Base** key could be used to override the true URI of the document to fix the relative links. This concept is parallel to the description of the body element <BASE> as described in Section 2.7.2 of the HTML specification [11].

6.9.7 Sound actions

PDF 1.2

A Sound action can be used to play a sound from with a PDF document.

Table 6.31 *Sound action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Object subtype. Always **Sound**.
Sound	stream or File specification	*(Required)* A Sound object. See Section 6.16 on page 131. An external sound file.
Synchronous	Boolean	*(Optional. The default is false.)* If this key is *true*, the viewer does not permit further user interaction, other than cancelling the action, until the sound has been completed played.
Repeat	Boolean	*(Optional. The default is false.)* If this key is *true*, the sound is repeated indefinitely. If this key is present, the **Synchronous** key is ignored.
Mix	Boolean	*(Optional. The default is false.)* If this key is *true*, the viewer attempts to mix this sound with any other sound playing. If this key is false, any playing sound is stopped before starting this sound. This may be used to stop a Repeating sound.
Volume	number	*(Optional. The default is 1.)* The volume setting for the sound. This number must be between 0 and 1.

6.9.8 Movie Player actions

PDF 1.2

A Movie Player action can be used to play a movie in a "floating window" or within the rectangle of a Movie Player annotation. A Movie Player action dictionary is identical to the Movie Activation dictionary described in Table 6.16, with the following additional elements.

Table 6.32 *Movie Player attributes (in addition to those in Table 6.16)*

Key	*Type*	*Semantics*
Type	name	*(Optional)* Object type. Always **Action**.
S (Subtype)	name	*(Required)* Object subtype. Always **Movie**.
Operation	name	*(Optional)* The action command for the movie. Currently defined values are: **Play** Start playing the movie. The **Mode** key determines the type of play. This is the default value. **Stop** Stop playing the movie and exit the action. **Pause** Pause a playing movie. **Resume** Resume a paused movie. If **Play** is selected on a paused movie, the movie is repositioned to the **Start** position (if specified) before playing.
T (Title)	string	The **Title** of the Movie annotation to be played. The movie annotation must be contained in the destination page.
Annot	dictionary	An indirect reference to a Movie annotation.

Either **T** or **Annot** must be specified, but not both.

6.9.9 SetState action

PDF 1.2

Corresponding to the use of appearances for an annotation is the **SetState** action. This action allows for the setting of the value of the **AS** key in the appearance dictionary. This allows another annotation, page, or document level event to change the state of one or more target annotations. Note that the effect of this action is temporary and does not permanently affect the document.

Table 6.33 *SetState action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S	name	*(Required)* Subtype. Must be **SetState**.
T	dictionary or array	*(Required)* Target. This is an indirect reference to an annotation or the fully qualified field name of a form field or an array of such objects, each of which is to change state as a result of this action.
AS	name	*(Optional)* Appearance State, indicating which appearance in the sub-dictionary of appearances for the annotation should be shown. This has the effect of setting the appearance state (**AS**) value in the annotation's dictionary.

6.9.10 Hide action

PDF 1.2

Corresponding to the *Hidden* flag, the Hide action allows the author to determine when a particular annotation is hidden or shown. For example, combined with the action triggers (see page 97), when the user rolls the mouse over an area of the page, an annotation can appear that describes the action that would occur if the user clicked on that spot on the page. The Hide action is equivalent to setting or clearing the Hidden flag for the annotation. Note that the effect of this action is temporary and does not affect the document.

Table 6.34 *Hide action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action subtype. Must be **Hide**.
T	dictionary or array	*(Required)* Target. This is an indirect reference to an annotation or the fully qualified field name of a form field or an array of such objects, each of which is to be hidden or shown.
H	Boolean	*(Optional)* Hide. If this Boolean is *true*, the action hides the target annotation. If *false*, the action shows the target annotation. The default for this key is *true*.

6.9.11 Named actions

PDF 1.2

PDF 1.2 defines several actions, shown in the following table, that a viewer is expected to support. Additional names may be defined in the future. Viewers may support additional named actions, but a document that uses such names is not portable. If a named action is inappropriate for a viewing platform, or if the viewer does not recognize the name, it should take no action.

Table 6.35 *Named Action Attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **Named**.
N (Name)	name	*(Required)* One of the names listed in Table 6.36.

Table 6.36 *Named Action List*

Name	*Action*
FirstPage	Go to the first page of the document.
PrevPage	Go to the previous page in the document.
NextPage	Go to the next page in the document.
LastPage	Go to the last page in the document.

6.9.12 NOP action

PDF 1.2

The **NOP** action does nothing when it is executed. It exists primarily as a means of overriding a trigger point that would inherit behavior either from some other part of the document or from the viewer itself. For example, a page could specify the **NOP** action for the **NP** key of its Additional Actions dictionary to prevent anything from happening when the user attempted to go to the next page of the document.

Table 6.37 *NOP Action Attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	*(Required)* Action type. Always **NOP**.

6.10 Name trees

PDF 1.2

In PDF 1.2, the Catalog of a document may contain a **Names** key. The value of this key is a dictionary. Each value in this dictionary is a "name tree," which is a tree, similar to the Pages tree, where the leaf-nodes contain pairs of strings (the "names") and objects (the "values"). A name tree has the same purpose as a dictionary, mapping keys to values, but it does so in a different manner, and the keys in a name tree are strings, not PDF name objects.

The only entry in the **Names** dictionary that is defined by PDF 1.2 is **Dests**, which is used for named destinations. (See Section 6.8.2 on page 95.) Other entries may be added in the future.

Table 6.38 *Names dictionary in the Catalog*

Key	*Type*	*Semantics*
Dests	dictionary	The root of a name tree for named destinations.

A name tree contains three kinds of nodes, a *root* node (one per tree), *intermediate* nodes, and *leaf* nodes. The root node contains only a **Kids** array. An intermediate node contains a **Kids** array and a **Limits** array. A leaf node contains a **Limits** array and a **Names** array. The names in the tree are stored in the leaf nodes, sorted by binary value, where each string is interpreted as a stream of unsigned octets (8-bit bytes). The names are sorted in ascending order. Shorter names appear before longer names that begin with the same byte sequence. Names are encoded with **PDFDocEncoding**.

Table 6.39 *The root node in a name tree*

Key	*Type*	*Semantics*
Kids	array	An array of indirect references to the immediate children of the root node. They may be intermediate nodes or leaf-nodes.

Table 6.40 *An intermediate node in a name tree*

Key	*Type*	*Semantics*
Kids	array	An array of indirect references to the immediate children of this node. They may be intermediate nodes or leaf nodes.
Limits	array	An array of two strings, representing the (alphabetically) least and greatest names included in any leaf nodes that are descendants of this node.

Table 6.41 *A leaf node in a name tree*

Key	*Type*	*Semantics*
Limits	array	An array of two strings, representing the (alphabetically) least and greatest names in the **Names** array.
Names	array	An array of the form [*name value name value …*], where each name is a string, and the value is an indirect reference to the object associated with that name. The names are sorted in alphabetical order.

As an example of a name tree, consider a document that has named destinations for all the chemical elements, where the name of the destination is the name of the element itself, from actinium to zirconium. An outline of the tree, showing object numbers and nodes, might look like this:

#3: root
    #4: intermediate: actinium to phosphorus
        #6: leaf: actinium = …
        #7: leaf: …
        #8: intermediate: … to gadolinium
        #9: leaf: gallium = #14, germanium = #15, gold = #16
            #14: destination: …
            #15: destination: `[/FitR 0 0 100 100]`
            #16: destination: …
        #10: intermediate: hafnium to phosphorus
    #5: intermediate: platinum to zirconium
        #11: leaf: platinum = #17, plutonium = #18, polonium = #19
            #17: destination: …
            #18: destination: …
            #19: destination: `[/FitXYZ 50 50 800]`
        #12: intermediate: potassium to …
        #13: intermediate: … to zirconium

The representation of this tree in the PDF file would look like this:

```
1 0 obj <<
/Type /Catalog
/Names 2 0 R
>> endobj

2 0 obj <<          % Dictionary of name-trees
/Dests 3 0 R
>> endobj

3 0 obj <<                          % Root node
/Kids [4 0 R 5 0 R]
>> endobj

4 0 obj <<                 % Intermediate node
/Limits [(actinium) (phosphorus)]
/Kids [6 0 R 7 0 R 8 0 R 9 0 R 10 0 R]
>> endobj

5 0 obj <<                 % Intermediate node
/Limits [(platinum) (zirconium)]
/Kids [11 0 R 12 0 R 13 0 R]
>> endobj

9 0 obj <<                          % Leaf node
/Limits [(gallium) (gold)]
/Names [(gallium) 14 0 R (germanium) 15 0 R (gold) 16
0 R]
>> endobj

11 0 obj <<                         % Leaf node
```

```
/Limits [(platinum) (polonium)]
/Names [(platinum) 17 0 R (plutonium) 18 0 R
(polonium) 19 0 R]
>> endobj

15 0 obj <<                          % Destination
/D [/FitR 0 0 100 100]
>> endobj

19 0 obj <<                          % Destination
/D [/FitXYZ 50 50 800]
>> endobj
```

6.11 Info dictionary

A document's trailer may contain a reference to an Info dictionary that provides information about the document. This optional dictionary may contain one or more keys, whose values should be strings. These strings may be displayed in an Acrobat viewer's Document Info dialog. The characters in these strings are encoded using the predefined encoding **PDFDocEncoding**, described in Appendix C.

Note *Omit any key in the Info dictionary for which a value is not known, rather than including it with an empty string as its value.*

Table 6.42 *PDF Info dictionary attributes*

Key	*Type*	*Semantics*	
Author	string	*(Optional*) The name of the person who created the document.	
CreationDate	Date	*(Optional)* The date the document was created.	
ModDate	Date	*(Optional)* The date the document was last modified.	PDF 1.1
Creator	string	*(Optional)* If the document was converted into a PDF document from another form, this is the name of the application that created the original document.	
Producer	string	*(Optional)* The name of the application that converted the document from its native format to PDF.	
Title	string	*(Optional)* The document's title.	PDF 1.1
Subject	string	*(Optional)* The subject of the document.	PDF 1.1
Keywords	string	*(Optional)* Keywords associated with the document.	PDF 1.1

Info strings that are to be interpreted as dates must include the `D:` prefix (see on Section 7.2 on page 133). In particular, the 1.0 key **CreationDate** and the 1.1 key **ModDate** should use this format. All Info strings that represent dates should be displayed as a human-readable date. Other Info strings are uninterpreted.

Info keys and strings may be added to or changed by users or extensions, and some extensions may choose to permit searches on these keys. PDF 1.1 does not define short names for the keys in Table 6.42, to make it easier to browse and edit Info dictionary entries. New names should be chosen with care so that they make sense to users.

Although private data can be stored in the Info dictionary, it is more appropriate to store it in the Catalog. This allows a user or program to alter entries in the Info dictionary with less chance of unforeseen side effects.

Example 6.1 shows an example of an Info dictionary.

Example 6.1 *Info dictionary*

```
1 0 obj
<<
/Creator (Adobe Illustrator)
/CreationDate (D:19930204080603-08'00')
/Author (Werner Heisenberg)
/Producer
(Acrobat Network Distiller 1.0 for Macintosh)
>>
endobj
```

6.12 Articles

PDF 1.1

An *article thread* identifies related elements in a document, enabling a user to follow a flow of information that may span multiple columns or pages.

A PDF document may include one or more article threads. Each thread has a title and a list of thread elements, which are referred to as *beads*. A viewer may allow the user to select a particular thread and then navigate through it; the viewer automatically maintains a comfortable zoom level for reading and moves from one bead to the next, rather than from one page to the next.

If a document includes any threads, they are stored in an array as the value of the **Threads** key in the Catalog object. Each thread and its beads are dictionaries. Table 6.43 lists the attributes of a Thread dictionary, and Table 6.44 lists the attributes of a Bead dictionary.

Table 6.43 *Thread attributes*

Key	*Type*	*Semantics*
F (First)	dict	*(Required; must be an indirect reference)* Specifies the first bead in this thread.

I (Info)	dict	*(Optional)* Information about the thread. This dictionary should contain information similar to the document's Info dictionary and should use the same key names and data formats for entries that correspond to Info dictionary entries. Entries in this dictionary should be strings encoded using the predefined encoding **PDFDocEncoding**, described in Appendix C.

Table 6.44 *Bead attributes*

Key	*Type*	*Semantics*
T (Thread)	dict	*(Required for the first bead of a thread; must be an indirect reference)* The thread in which this bead is an element. **PDF 1.2**
V (Prev)	dict	*(Required; must be indirect)* The previous bead of this thread; for the first bead in a thread, **V** specifies the last bead in the thread.
N (Next)	dict	*(Required; must be indirect)* The next bead of this thread; for the last bead in a thread, **N** specifies the first bead in the thread.
P (Page)	dict	*(Required; must be indirect)* The Page on which this bead appears.
R (Rect)	Rectangle	*(Required)* Rectangle specifying the location of this bead.

Example 6.2 shows a thread with three beads:

Example 6.2 *Thread*

```
22 0 obj <<
/F 23 0 R
/I << /Title (Man Bites Dog) >> >>
endobj

23 0 obj <<
/T 22 0 R
/V 25 0 R
/N 24 0 R
/P 8 0 R
/R [158 247 318 905] >>
endobj

24 0 obj <<
/T 22 0 R
/V 23 0 R
/N 25 0 R
/P 8 0 R
/R [322 246 486 904] >>
endobj
```

```
25 0 obj <<
/T 22 0 R
/V 25 0 R
/N 23 0 R
/P 10 0 R
/R [157 254 319 903] >>
endobj
```

The Page object for each page on which beads appear should contain a **B** key, as described in Section 6.4, "Page objects." The value of this key is an array of indirect references to each bead on the page, in drawing order.

6.13 File ID

PDF 1.1

A PDF file may contain a reference to another PDF file. Storing a file name, even in a platform-independent format, does not guarantee that the file can be found, even if it exists and its name has not been changed. Different server software applications often present different names for the same file. For example, servers running on DOS platforms must convert all file names to eight letters and a three-letter extension. Different servers use different strategies for converting long names to this format.

References to PDF files can be made more reliable by making the PDF file reference consist of two parts: (1) a normal operating system-based file reference and (2) a file ID. The file ID characterizes the file and is stored with the file. Placing a file ID with the file reference and in the file itself increases the chances that a file reference can be resolved correctly. Matching the ID in the reference with the ID in the file indicates whether the desired file was found.

A PDF file may have an **ID** key in its trailer. The value of this key is an array of two strings. The first element is a permanent ID, based on the contents of the file at the time the file was created. This ID does not change when the file is incrementally updated. The second element is a changing ID, based on the contents of the file at the time the file is incrementally updated. When a file is first written, the IDs are set to the same value. When resolving a file reference, if both IDs match, it is very likely that the correct file has been found. If only the first ID matches, then a different version of the correct file has been found.

To help insure the uniqueness of the file ID, it is recommend that file ID be computed using a message digest algorithm such as MD5, as described in *RFC 1321: The MD5 Message-Digest Algorithm* [22]. It is recommend that the following information be passed to the message digest algorithm:

- the current time
- a string representation of the location of the file, usually a path name
- the document size in bytes
- the value of each entry in the document's Info dictionary.

Implementation note *Adobe applications pass this information to the MD5 message digest algorithm to calculate file IDs. Note that the calculation of the file IDs need not be reproducible. All that matters is that the file IDs are 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 IDs for the same file created at the same time, but this does not affect the uniqueness of the ID.*

6.14 Encryption dictionary

PDF 1.1

Documents can be protected via encryption, as described in Section 5.7, "Encryption." Every protected document must have an Encryption dictionary, which specifies the security handler to be used to authorize access to the document. The Encryption dictionary also contains whatever additional information the security handler chooses to store in it.

The only entry required by PDF in the Encryption dictionary is the **Filter** key, whose value is the name of the security handler that encrypted the document. A security handler may require additional keys in the Encryption dictionary. The keys required by the built-in standard security handler are described below.

Standard encryption

The built-in encryption method provided by PDF allows for the following functionality. A document has two passwords: an owner password and a user password. The document also specifies operations that should be restricted even when the document is decrypted: printing, copying text and graphics out of the document, modifying the document, and adding or modifying text notes. When the correct user password is supplied, the document is opened and decrypted but these operations are restricted; when the owner password is supplied, all operations are allowed. The owner password is required to change these passwords and restrictions. A document is encrypted whenever a user or owner password or restrictions are supplied for the document. However, a user is prompted for a password on opening a document only if the document has a user password. It is possible to determine if a document has a user password by testing the empty string as the user password, as described in Algorithm 6.6 below.

Table 6.45 lists the Encryption dictionary attributes of a document that is encrypted using the standard security handler. The data in this dictionary is used to determine if a candidate password string is the user password, the owner password, or neither. If it is the user password, the dictionary data is used to determine the operations that should be restricted. The calculation and use of data in this dictionary is described below.

Table 6.45 *Standard security handler attributes*

Key	*Type*	*Semantics*
Filter	name	*(Required in all encrypted documents)* Name of security handler. The name of the built-in handler is **Standard**.

R (Revision)	number	*(Required)* Revision number of algorithm used to encode data in this dictionary. The current revision number for the **Standard** security handler is 2.
O (Owner)	string	*(Required)* Data describing password needed to gain full access to file.
U (User)	string	*(Required)* Data describing password needed to open file.
P (Permissions)	integer	*(Required)* Collection of flags describing permissions granted to user who opens a file with the user password. See Table 6.46.

The value of the **P** (Permissions) key is an unsigned 32-bit integer that contains a collection of access rights. These rights are enabled on opening the document with the user password if the corresponding bit is set in the integer. Table 6.46 specifies the meanings of the bits, with bit 1 being the least significant.

Table 6.46 *Permission flags*

Bit position	*Semantics*
1–2	Reserved. Must be set to 0.
3	Enable printing of the document.
4	Enable changing the document other than by adding or changing text notes.
5	Enable copying of text and graphics from the document.
6	Enable adding and changing text notes.
7–32	Reserved. Each bit must be set to 1.

As described in Section 5.7, “Encryption,” one function of a security handler is to produce a five-byte key that is provided as input to the PDF encryption algorithm. Given a password string, the standard security handler computes an encryption key using Algorithm 6.3:

Algorithm 6.3*Computing an encryption key*

1. Truncate the password string to 32 bytes. (Treat an omitted password as an empty string.) If the string is less than 32 bytes long, pad it with bytes from the following string to make it exactly 32 bytes long:

   ```
   <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>
   ```

   Pad the password string from the beginning of the pad string. That is, if the password string is *n* bytes long, concatenate bytes 1 to 32 – *n* of the pad string to the password string.

2. Provide the string as input to the MD5 hash function.

3. Input the value of the Encryption dictionary's **O** (Owner) key (a 32-byte string) to the MD5 hash function. (Algorithm 6.4 explains how the **O** string is computed.)

4. Treat the value of the **P** (Permissions) key as an unsigned four-byte integer, and provide these bytes to the MD5 hash function, low-order byte first.

5. Input the first element of the file's ID to the MD5 hash function. (See Section 6.13, "File ID.")

6. The first five bytes of the MD5 output make up the encryption key.

The encryption key is used to encrypt strings and stream data, as described in Section 5.7, "Encryption." The same key decrypts data as well.

To create an encrypted document, the standard security handler must compute the entries in the Encryption dictionary as well as the encryption key. While calculation of the value of the **Filter**, **R** (Revision), and **P** (Permissions) keys are straightforward, calculation of the value of the **O** (Owner) and **U** (User) keys requires further explanation.

Algorithm 6.4 shows how to calculate the value of the **O** key in the Encryption dictionary:

Algorithm 6.4*Computing the* **O** *(Owner) key in the Encryption dictionary*

1. Truncate or pad the owner password string as needed, following the same process as in step 1 of computing an encryption key (Algorithm 6.3). If there is no owner password, the user password is used instead.

2. Provide the string as input to the MD5 hash function.

3. Create an RC4 key using the first five bytes of the MD5 output.

4. Truncate or pad the user password string as needed, following the same process as in step 1 of Algorithm 6.3.

5. Encrypt the padded user password string using the RC4 algorithm with the key from step 3.

6. Store the encrypted string as the value of the **O** key in the Encryption dictionary.

Algorithm 6.5 shows how to calculate the value of the **U** key in the Encryption dictionary:

Algorithm 6.5*Computing the* **U** *(User) key in the Encryption dictionary*

1. Create an encryption key (Algorithm 6.3) based on the user password string.

2. Encrypt the 32-byte string specified in step 1 of Algorithm 6.3 using the RC4 algorithm with the encryption key from the previous step.

3. Store the encrypted string as the value of the **U** key in the Encryption dictionary.

The standard security handler performs several other functions beyond calculating the key needed to encrypt or decrypt data, as described above. It determines if the candidate password string is the user password, the owner password, or neither. If it is the user password, it determines the operations that should be restricted.

Given a password, the standard security handler uses the contents of the Encryption dictionary to determine if a document should be opened and what permissions should be granted. If the password is the user password, the document is opened only with the permissions specified in the Encryption dictionary. If the password is the owner password (but not the same as the user password), all permissions are enabled.

To determine if a password is the correct user password, the security handler uses Algorithm 6.6:

Algorithm 6.6*Checking for the correct user password*

1. Compute an encryption key from the password string, using Algorithm 6.3.
2. Decrypt the value of the **U** key, using RC4 with the encryption key.
3. If the resulting string is the same as the string described in step 1 of Algorithm 6.3, the password is the user password.

To determine if a password is the correct owner password, the security handler uses Algorithm 6.7:

Algorithm 6.7*Checking for the correct owner password*

1. Compute an encryption key from the password string, using steps 1 through 3 of Algorithm 6.4.
2. Decrypt the value of the **O** key, using RC4 with the encryption key.
3. Use the resulting string as an encryption key to decrypt the value of the **U** key, again using RC4.
4. If the resulting string is the same as the string described in step 1 of Algorithm 6.3, the password is the owner password.

Note *Despite the specification of document permissions in a PDF file, PDF cannot enforce the restrictions specified. It is up to the implementors of PDF viewers to respect the intent of the document creator by limiting access to an encrypted PDF file according to the permissions and passwords contained in the file.*

Document creators have two choices if the standard encryption provided by PDF is not sufficient. They can use an alternative, more secure, security handler, or they can encrypt whole PDF documents, bypassing PDF security entirely.

6.15 Acrobat Forms

PDF 1.2

6.15.1 Introduction

Beginning with version 1.2, PDF defines a number of interactive document features. Prominent among these is the feature that allows a PDF document to represent a *form*. By form we mean the PDF equivalent of the familiar paper instrument and not the Form XObject defined on page 179. Any unqualified use of the term *form* refers to the feature defined here; the other is referred to explicitly as a Form XObject.

A form consists of a collection of *fields***.** A field should be thought of as a set of properties. The three most important properties of a field are its type, its name, and its value. Other properties are used to specify the appearance of a field. Fields can be organized into a hierarchy, and there are still other properties of the field that associate it with its parent and children.

6.15.2 Form

A PDF file may contain at most one form, although that form may have an arbitrary number of fields that appear on any page of the PDF file. The properties of the form itself are encoded in the AcroForm dictionary, which must be referenced in the Catalog dictionary of the PDF file using the key **AcroForm**.

Note *Although each PDF file contains at most a single form, arbitrary sets of fields may be imported and exported from the PDF file; see Section 6.15.13 on page 129.*

The AcroForm dictionary has the following attributes:

Table 6.47 *AcroForm dictionary attributes*

Key	*Type*	*Semantics*
Fields	array	*(Required)* This array contains a reference to each root field in the PDF file. A root field is a field with no parent.
NeedAppearances	Boolean	*(Optional)* Acrobat forms use annotations; in particular, forms use the **AP** dictionary of an annotation to represent the appearance of a field's value. Form-authoring applications can omit the **AP** dictionary for fields that contain text (see Section 6.15.6 on page 121). When the PDF file is opened, if **NeedAppearances** is *true*, the viewer creates an **AP** dictionary and all of its subparts including the **N** dictionary and the appearance stream it refers to, for any field containing variable text that does not have one. The default value for this attribute is *false*.

In addition to the attributes in Table 6.47, the AcroForm dictionary can contain document-wide defaults for the **Q**, **DR**, **OH** and **DA** field dictionary attributes, described below.

6.15.3 Fields

The properties that define each distinct field are stored in a dictionary. All such dictionaries must be indirect objects. A particular property of a field is found in one of three ways. In the simplest case, the property is found immediately in the dictionary, referenced by a predefined key. For example, the simple way to represent the type of a checkbox field is to place the key-value pair

```
<< /FT /Btn >>
```

in the field's dictionary, indicating that is a button.

Fields are arranged in a hierarchy, and many properties that can be stored directly in the dictionary can also be inherited from the field's parent. In general, an inherited property might be found in the grandparent or some arbitrary ancestor of the field.

One field property is neither stored directly in the dictionary nor inherited from an ancestor. This property, the *fully qualified field name*, must be derived from the **T** attribute of the field and its ancestors. A fully qualified field name is an ordered collection of the partial field names of a field and all of its ancestors. The partial field name is stored in the dictionary as the key-value pair

```
<< /T string >>
```

One way of producing the ordered collection is to store the partial field names in an array. Using this technique, two fully qualified field names are identical if the arrays are of equal size and the components of the array are pairwise identical.

It is possible for different fields to have the same fully qualified field name, albeit under a restricted set of circumstances. All fields with the same fully qualified field name must be descendents of a common ancestor with that same name. In practice, this means that none of these descendants can have a **T** attribute. If different fields have the same fully qualified field name, the fields should differ only in properties that specify appearance. In particular, whenever different fields have the same fully qualified field name, they must have the same type, value, and default value, as described below. Given a fully qualified field name, there is an effective way to find the common ancestor of all fields with that name.

6.15.4 Field dictionaries

All types of fields share the following attributes. Since the notion of required and optional fields is somewhat clouded by inheritance, in addition to the normal categories of *Required* and *Optional*, we use the terms *Required, inheritable* for attributes that *must* exist somewhere in the inheritance hierarchy, and *Optional, inheritable* for attributes that *may* exist somewhere in the inheritance hierarchy.

Table 6.48 *Attributes common to all types of fields*

Key	*Type*	*Semantics*
FT	name	*(Required, inheritable)* The type of field. Valid field types include:

Btn Button field. This field type is used for a *checkbox*, a field that toggles between two states, on and off. It is also used for a *radio button*, a field made up of an arbitrary number of toggles, only one of which can be on at any given time. It is also used for a *push button*, a simple interactive element that retains no value.

Tx Text field. A field whose value is text. Text may be single- or multi-line.

Ch Choice field. Like a radio button, this field type takes on one value from an arbitrary set. It is presented to the user as a pop-up list or a scrollable list, rather than as a set of mutually exclusive toggles. See Section 6.15.11 on page 126.

T string *(Optional)* The partial field name. The fully qualified field name is derived from the partial field name, as described above. The string is encoded in **PDFDocEncoding**.

V *various* *(Optional, inheritable)* The value. The format of the value is dependent on the type of the field as specified by the **FT** attribute. See the following sections for details on the field values for particular types. If the value is a string, it is encoded in **PDFDocEncoding**.

DV *various* *(Optional, inheritable)* The default value. The value of the field reverts to this when a **ResetForm** action (see page 130). The format of **DV** is the same as **V**. If the default value is a string, it is encoded in **PDFDocEncoding**.

Ff integer *(Optional, inheritable)* Flags. Collection of flags defining various characteristics of the field. The default value is 0. The following flags apply to any field, regardless of field type:

bit 1 *Read-only* flag, indicating that the user is not allowed to change the field value in the viewer. This is intended for use with computed fields or fields that are imported from a database.

bit 2 *Required* flag, indicating that the this field must have a value at the time the field is exported by a **Submit** action; see page 129.

bit 3 *No-export* flag, indicating that this field must not be exported by a **Submit** action; see page 129.

Kids array *(Optional)* The **Kids** and **Parent** attributes define the child/parent relationship between field dictionaries. This array contains references to child field dictionaries of this field. A child may be referenced from only one **Kids** array and thus has a unique parent.

A field dictionary may use a **Kids** array to reference sub-fields; i.e. fields that share a common prefix with their fully qualified field. That shared prefix is the fully qualified field name of this field. For instance, you might create a field dictionary whose **T** key is `(Address)` with sub-field dictionaries whose **T** keys are `(Street)` and `(City)`.

A field dictionary may use a **Kids** array to reference child fields that differ from this field only in their appearance. Such child fields inherit **FT**, **V**, and **DV** attributes and may not specify a **T** attribute.

Parent	dictionary	*(Required if this field is referenced from another field's* **Kids** *array)* The parent field.

6.15.5 Widget annotations

Acrobat forms use annotations to represent the appearance of fields and to manage user interactions. These annotations have a subtype of **Widget**.

If a field has an appearance, that appearance must be represented with an **AP** dictionary (seepage 87). Conversely, if an **AP** dictionary exists, the appearance it represents must be consistent with the value of the field.

As an optimization, the attributes of a field are merged into a single dictionary with the attributes of the Widget annotation that represents the appearance of the field and manages user interactions. Hence, a field may also be a bona fide annotation that is referenced in the **Annots** array of some Page dictionary. The keys for a field dictionary and an annotation dictionary are mutually exclusive, so there is no ambiguity in doing this. A field is also a Widget annotation if and only if it contains an **S** (Subtype) attribute whose value is **Widget**. A Widget annotation is also a field if and only if it contains a **T** attribute, or if it contains a **Parent** attribute one of whose ancestors, as found by following the **Parent** references, contains a **T** attribute.

A Widget annotation that is not also a field is a "purely interactive document element"; see "*Push button* flag," page 123.

6.15.6 Fields comprising variable text

Both text fields and choice fields can contain variable text, i.e., text stored in field dictionary attributes that are used to generate the appearance of the field. The sections on text and choice fields, below, contain more information on exactly which attributes are used in each case. When there is a change in the attributes of a text or choice field that affect the appearance, the viewer must generate an appearance stream for the field, contained in the **AP** dictionary of the Widget annotation, conforming to these new attributes. A number of additional attributes are defined to specify the initial appearance of this text.

Table 6.49 *Attributes common to all types of fields containing variable text*

Key	*Type*	*Semantics*
DR	dict	*(Required, inheritable)* Default resources. This is a Resources dictionary (see page 138). At a minimum, this dictionary must contain a resource of type **Font**. The Font resource is, in turn, a dictionary which must contain at least one key-value pair specifying the resource name and resource dictionary of the default font for this field. All resources from the **DR** dictionary are copied to the appearance stream when it is created or updated.
DA	string	*(Required, inheritable)* Default appearance string. This string contains a sequence of valid page-content graphics or text-state operators that determine such appearance properties as text color, text size, etc.

Q	integer	*(Optional, inheritable)* Quadding. This determines whether the input text is left-justified (0), centered (1) or right-justified (2). The default value is 0.
OH	integer	*(Optional)* Original height. This determines the height of the **BBox** attribute of the appearance stream. The default for this attribute is the height of the **Rect** attribute of the Widget annotation that contains the appearance stream.

A viewer must generate the appearance stream according to the following procedures. There are two cases to consider. In the first case, the appearance stream is created where none existed previously. In the second, an existing appearance stream is updated to reflect a new value.

In the first case, when the Widget annotation used to represent the appearance of the field contains no appearance stream, a viewer performs the following actions:

- Create the appearance dictionary, **AP**, in the Widget annotation. Create an empty stream. Set the normal face, **N**, of the appearance dictionary to the newly created stream. Set the Resources dictionary of the newly created stream to **DR**. Set the **BBox** attribute of the stream. The lower and left coordinates of the **BBox** are set to 0. The top coordinate is set to **OH**. The right coordinate is set to the width of the **Rect** of the Widget annotation divided by the ratio of **OH** to the height of the **Rect** of the annotation. All other stream attributes are set to their defaults.

- Insert "`/Tx BMC q`" into the stream data. The **BMC** operator labels the stream data as marked content with the tag **Tx** (see page 240). The **q** operator saves the current graphics state.

- If any graphics-state changes are required, such as clipping, emit them here.

- Insert "`BT`" to begin the text object.

- Insert the **DA** string into the stream data, subject to the following constraints:

  Graphics state operators, if specified, must be operators that are legal within a text object (see Figure 8.1 on page 210). At a minimum, a **Tf** operator along with its two parameters, *fontname* and *size*, must be present. The *fontname* must match a resource name in the **Font** resource of the default resource dictionary, **DR**. If *size* is 0, then the size of the font is computed as a function of **OH** (or the height of the annotation's rect when **OH** is not defined), and that value is inserted instead of 0. (This is referred to as "auto-sizing.") If a **Tm** operator is present, the *x* and *y* translation components are replaced with positioning information determined by the viewer to be appropriate, based on the field value, the **Q** attribute, and any layout rules it employs.

Note *The* **DA** *string should contain at most one* **Tm** *operator.*

- If the **DA** string contains no **Tm** operator, emit `Tm` with appropriate *x* and *y* translation components, as described above.

- Emit any text string operators necessary to show the variable text along with any additional, necessary text positioning operators.

- Emit `ET Q EMC`. The **ET** operator ends the text object, the **Q** operator restores the previous graphics state, and the **EMC** operator indicates the end of the content that was marked with tag **Tx**.

In the second case, when an appearance stream already exists, the viewer performs the following actions:

- Copy the resources from **DR** to the Resources dictionary of the appearance stream. Conflicts between resource names in **DR** and the existing appearance stream are resolved in favor of the appearance stream, i.e., no copy takes place.

- Search the appearance stream for the sequence `/Tx BMC` and the matching `EMC`, accounting for any nested **BMC**/**EMC** pairs in this matching process. Valid appearance streams should have exactly one such **BMC** operator with tag **Tx**.

- Replace the stream contents identified in the previous search by the stream contents generated in the first case beginning at the second step; i.e. emit `/Tx BMC q`, etc. If the original appearance stream contains no content marked with tag **Tx**, then append the new stream contents to the end of the original stream.

6.15.7 Button field

The **Btn** field type is used for checkboxes, radio buttons, and simple push buttons that retain no value. Simple push buttons are distinguished from checkboxes or radio buttons by the *push-button* flag. Radio buttons and checkboxes are distinguished with the *radio* flag.

Table 6.50 *Field flags (***Ff***) for Btn fields*

Bit Position	*Semantics*
16	*Radio* flag. If this flag is *true* (i.e., if this bit is 1), then the **Btn** field is a radio button; otherwise it is a checkbox. This flag is meaningful only if the push-button flag is *false*.
17	*Push-button* flag. This flag is used to indicate a button field that is “purely interactive”; i.e., one that responds to user input in some way but does not change state as a result of that input.

6.15.8 Push button

A push button is the simplest type of field. It has no value. It can assume any of the attributes common to all fields except the **V** and **DV** attributes. The **FT** attribute must be set to **Btn**, and the *push-button* flag must be set to 1.

6.15.9 Checkbox

A checkbox is a field that can assume two states, *on* and *off*. Each state can have a separate appearance created from arbitrary marking operations. These appearances are found in the **AP** dictionary of the field annotation; see page 87. The appearance for the *off* state is optional but, if present, must be stored in the **AP** dictionary under the name **Off**. The recommended name for the *on* state is **On**, but this is not required. The **Ff** attribute of a checkbox field should have both the *push-button* and *radio* flags set to 0. In addition to the attributes common to all fields, a checkbox field can contain the following attributes:

Table 6.51 *Checkbox attributes*

Key	*Type*	*Semantics*
FT	name	*(Required, inheritable)* Field type. Must be **Btn**.
Ff	integer	*(Optional, inheritable)* Flags. The *push-button* and *radio* flags must both be set to 0.
V	name	*(Optional, inheritable)* Field value. This indicates which of the two states the checkbox is in. The value of a checkbox is the name that is used to identify its appearance in the **AP** dictionary. So, if the checkbox is in the *off* state, then the value is **Off**. If it is in the *on* state, then the value contains whatever name is used to identify this state in the **AP** dictionary. As mentioned above, it is recommended but not required that this name be **On**. The default value for this attribute is **Off**.

Example 6.3 *Simple checkbox field with Zapf Dingbats check character*

```
1 0 obj
<<
/FT /Btn
/T (Urgent)
/V /On
/AS /On
/AP << /N << /On 2 0 R >> >>
>>
endobj
2 0 obj
<< /Resources 20 0 R /Length 50 >>
stream
q 0 0 1 rg BT /ZaDb 12 Tf 0 0 Td (8) Tj ET Q
endstream
endobj
```

6.15.10 Radio button

A radio button is a field that can assume at most one of *n*+1 predefined states. The radio button is conceptually formed from *n* checkboxes, with the added semantics that selecting any one of the checkboxes deselects all the others. The PDF

representation of a radio button is *n*+1 fields, one for the radio button itself and one for each of the underlying checkboxes. The **Ff** attribute of a radio button has the *push-button* flag set to 0 and the *radio* flag set to 1.

Each child of the radio button is a checkbox that inherits its type and value from the parent radio button.

In addition to the attributes common to all fields, the radio button contains the following attributes:

Table 6.52 *Radio button attributes*

Key	*Type*	*Semantics*
FT	name	*(Required, inheritable)* Field type. Must be **Btn**.
Ff	integer	*(Required, inheritable)* Flags. The *push-button* flag should be 0. The *radio* flag should be 1. A radio button contains one additional flag:
	bit 15	"*No toggle to Off*" flag. If this bit is 1, then selecting the checkbox that is in the *on* state leaves it in that state, rather than toggling it to *off*; in effect, the checkbox is "re-selected." If this bit is 0, the checkbox that is currently in the *on* state can be set to *off* even if no other checkbox enters the *on* state.
Kids	array	*(Required)* An array of *n* checkboxes.
V	name	*(Optional)* The value. **V** is the name of the key in the **AP** dictionary of whichever child checkbox is in the *on* state. The default value for this attribute is **Off**.

Example 6.4 *Radio button field with two buttons*

```
10 0 obj                    % The radio button
<<
/FT /Btn
/Ff … radio flag = 1, push-button = 0 …
/T (Credit Card)
/V /Master-Card
/Kids [11 0 R 12 0 R]
>>
endobj

11 0 obj                          % checkbox 1
<<
/Parent 10 0 R
/AS /Master-Card
/AP
<</N << /Master-Card 8 0 R /Off 9 0 R >> >>
>>
endobj
```

```
12 0 obj                              % checkbox 2
<<
/Parent 10 0 R
/AS /Off
/AP
<< /N << /Visa 8 0 R /Off 9 0 R >> >>
>>
endobj

8 0 obj                          % An “On” stream
<< /Resources 20 0 R /Length 50 >>
stream
q 0 0 1 rg BT /ZaDb 12 Tf 0 0 Td (8) Tj ET Q
endstream
endobj

9 0 obj                         % An “Off” stream
<< /Resources 20 0 R /Length 50 >>
stream
q 0 0 1 rg BT /ZaDb 12 Tf 0 0 Td (4) Tj ET Q
endstream
endobj
```

6.15.11 Choice

A choice is a field that contains *n* items, at most one of which may be selected as the field value. A choice may be presented to the user as a scrollable list within a rectangle on the page, or as a pop-up list triggered by user input. A pop-up list box may allow user input in addition to the predefined choice options, in which case it is referred to as a *combo box*. A choice can be used to represent the behavior of list boxes, combo boxes, or pop-up list boxes, depending on the setting of certain flags.

In addition to the attributes common to all fields, the field dictionary that represents a choice contains the following attributes:

Table 6.53 *Choice attributes*

Key	*Type*	*Semantics*
FT	name	*(Required, inheritable)* Field type. Must be **Ch**
Ff	integer	*(Optional, inheritable)* Flags. Collection of flags defining various characteristics of the list box field. The following flags apply specifically to choices:
	bit 18	*Pop-up* flag. Distinguishes between a list box (0) and a pop-up (1).
	bit 19	*Edit* flag. Indicates that this list box is a combo box and hence, the user may input a string value rather than selecting it from the pre-defined list.

	bit 20	*Sort* flag. This flag is included for use by form-authoring tools that need to keep track of the user's preference regarding sort order of list items. If set, the authoring tool should sort the items in the **Opt** array in alphabetical order. This flag is *not* used by viewers. It is the responsibility of the authoring tool to put the items in the **Opt** array in the order desired. The viewer simply presents them in that order.
Opt	array	*(Required, inheritable)* An array of *n* elements. Each element is either a string representing one of the *n* potential values of the field, or an array containing two strings: a potential value and a string that is used to produce the appearance. All strings in the array are encoded in **PDFDocEncoding**.
V	string or name	*(Optional, inheritable)* Field value. Refers to the **Opt** array element that is currently selected. If that **Opt** array element is a string, then the value is that string. If it is an array, then the value is the string found in the first element of that array. The default value of this attribute is **null**.
TopIndex	integer	*(Optional, inheritable)* List boxes are scrollable. This integer indicates the index in the **Opt** array of the first visible element in the list box.

If the choice is presented to the user as a scrollable list, then the appearance contains the strings found in the **Opt** array, one per line, beginning with the string at the index indicated by **TopIndex**.

If the choice is presented to the user as a pop-up, then the appearance contains the string found in the **V** attribute.

Note *When a pop-up choice is activated in the viewer by user input, the choices appear on the screen, typically immediately under the current value. The viewer may choose to present to the user this transient appearance of the field by generating an appropriate appearance stream, based on the* **V** *and* **Opt** *attributes.*

Example 6.5 *List box field*

```
<<
/FT /Ch
/Ff ...
/T (Body Color)
/V (Blue)
/Opt [(Red) (My favorite color) (Blue)]
>>
```

6.15.12 Text field

A text field is a field whose value is arbitrary, user-specified text. The text may be restricted to a single line or allowed to span multiple lines. The text is presented on the page in a single style; i.e. font, size, color, etc. The value of a text field is a string. The variable text used to generate the appearance of the field as described on page 121 is found in the value of the text field.

In addition to the attributes common to all fields, the field dictionary that represents a text field contains the following attributes:

Table 6.54 *Text field attributes*

Key	*Type*	*Semantics*
FT	name	*(Required, inheritable)* Field type. Must be **Tx**.
Ff	integer	*(Optional, inheritable)* Flags. Collection of flags defining various characteristics of the text field. Flags that apply to text fields include:
	bit 13	*Multi-line*. Distinguishes between a multi-line (1) and single-line (0) text field.
	bit 14	*Password*. Keyboard input is not displayed on the screen when this flag is 1. Rather, some benign feedback is given, such as displaying an asterisk for each input character. In order to preserve the confidentiality of password fields, the value of a text field with the password flag set should never be stored in the PDF file by a viewer.
V	string	*(Optional, inheritable)* Field value. The value is a string in **PDFDocEncoding**.
MaxLen	integer	*(Optional, inheritable)* Maximum number of characters allowed in this text field.

Example 6.6 *Text field*

```
6 0 obj
<<
/FT /Tx
/Ff … set multi-line flag …
/T (Silly prose)
/DR 21 0 R
/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 65 >>
stream
/Tx BMC 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 EMC
endstream
endobj
```

6.15.13 AcroForm actions

Forms support three additional types of actions: **SubmitForm**, **ResetForm**, and **ImportData**.

6.15.13.1 SubmitForm Action

The **SubmitForm** action is used to send name-value pairs from the selected fields to the indicated URL, presumably that of a Web server where they will be processed, and from which a response sent back. The name that is sent is the fully qualified name of the field, and the value that is sent is the value of the **V** entry in the field dictionary.

These name-value pairs can be sent using Forms Data Format (FDF) (see Appendix H) or HTML Form format, as specified in RFC 1866, *Hypertext Markup Language - 2.0* (i.e. application/x-www-form-urlencoded).

Table 6.55 *SubmitForm action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	(*Required*) Action type. Always **SubmitForm**.
F	File specification	(*Required*) A URL file specification; see Section 7.3.3 on page 137. This is the URL of the script at the Web server that will process the request.
Fields	array	(*Optional*) The name-field value pairs to send, or to exclude from sending. Which of these two interpretations to use is decided based on bit 1 of the **Flags** key below. Each entry in the **Fields** array is an indirect object reference to the corresponding field dictionary in the AcroForm. If the **Fields** key is not present, then *all* name-field value pairs in the document are sent (except for those that have the "no-export" flag set), and bit 1 of **Flags** is ignored. The actual selection of name-value pairs to send can be further refined through bit 2 of the **Flags** key. If the **Fields** array represents fields to include, and an entry in this array represents a non-terminal field, that is, a field that has descendants (i.e. **Kids)** and those descendants can have values, then all name-value pairs from terminal fields in the subtree underneath it get sent (except for those that have the "no-export" flag set).
Flags	integer	(*Optional*) The binary value of the integer is interpreted as a collection of flags that define various characteristics of the action. The default value is 0.
	bit 1	*Include/exclude* flag. This determines how the **Fields** key is interpreted. If this bit is 0, then **Fields** represents the individual fields to send. If the bit is 1, then all name-value pairs in the AcroForm are sent, except for those in the **Fields** array (and those that are flagged as "no-export").
	bit 2	"*Include no-value fields*" flag. This can further restrict which name-value pairs get sent. If this bit is 0, then those fields (previously selected by the combination of the **Fields** array and bit 1 of **Flags**) that don't have a value (i.e. they don't have a **V** key) is *not* sent. If, on the other hand, this bit is 1, then for any fields that don't have a **V** key only the field name is sent.

	bit 3	*Export format* flag. If this bit is 0, then the data is sent using the Forms Data Format (FDF). Otherwise it is sent using the HTML Form format.

6.15.13.2 ResetForm Action

This action is used to revert the indicated fields to their default values (if defined), that is, the value of the **V** key is set to the value of the **DV** key in the field dictionary. If no default value is defined for a field, then the **V** key is removed. Resetting a field that can have no **V** key (i.e., a push button) has no effect.

Table 6.56 *ResetForm action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	(*Required*) Action type. Always **ResetForm**.
Fields	array	(*Optional*) The fields to reset, or to exclude from resetting. Which of these two interpretations to use is decided based on bit 1 of the **Flags** key below. Each entry is an indirect object reference to the corresponding field dictionary in the form. If the **Fields** key is not present, then *all* fields in the document are reset, and bit 1 of **Flags** is ignored. If the **Fields** array represents fields to include, and an entry in this array represents a non-terminal field, that is, a field that has descendants (i.e. **Kids)** and those descendants can have values, then all fields in the subtree underneath it get reset.
Flags	integer	(*Optional*) The binary value of the integer is interpreted as a collection of flags that define various characteristics of the action. The default is 0.
	bit 1	*Include/exclud*e flag. This determines how the **Fields** key is interpreted. If this bit is 0, then **Fields** represents the individual fields to reset. If the bit is 1, then all fields in the form are reset *except* for those in the **Fields** array.

6.15.13.3 ImportData Action

Import data in Forms Data Format from a specified file into the Form.

Table 6.57 *ImportData action attributes (in addition to those in Table 6.22)*

Key	*Type*	*Semantics*
S (Subtype)	name	(*Required*) Action type. Always **ImportData**.
F (File)	File specification	(*Required*) File specification of the FDF file to import.

6.16 Sounds

PDF 1.2

A PDF document may contain Sound objects, and it may refer to external sound files.

6.16.1 Sound objects

A Sound is represented as a stream, whose dictionary contains the following keys, in addition to the standard keys in any stream dictionary:

Table 6.58 *Sound attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Stream type. Always **Sound**
R (Rate)	number	*(Required)* Sample rate, in samples per second.
C (Channels)	integer	*(Optional)* Number of sound channels in the data. The default is 1.
B (Bits)	integer	*(Optional)* Number of bits per sample value (per channel). The default is 8.
E (Encoding)	name	*(Optional)* The format of the sample data. Allowable values are: **Raw** unspecified or unsigned values, 0 to $2^{\mathbf{B}\,(\text{Bits})} - 1$. **Signed** two's-complement values. **muLaw** μ-law encoded samples. **ALaw** A-law encoded samples. The default is **Raw**.
CO (Compression)	name	*(Optional)* The type of sound compression (not file compression) used on the data, if any. If this key is omitted, then no sound compression has been used, and the data contains sampled waveforms, to be played at **R** (Rate) samples per second, per channel.
CP (Compression Parameters)	*various*	*(Optional)* Optional parameters, specific to the Compression used.

Multi-channel uncompressed sound is saved in an interleaved format, in channel order (1, 2). For 2-channel stereophonic sounds, channel 1 is the left channel, and channel 2 is the right channel.

Samples 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 to be packed into the byte stream, starting at the most significant end of the byte and proceeding to the least 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 the subsequent bytes.

To maximize portability of PDF documents containing embedded sounds, Adobe recommends that PDF applications and plug-ins support at least the following formats (assuming the platform has sufficient hardware and OS support to play sounds at all):

B	8 or 16 bits per channel
C	1 or 2 channels
E	**Raw**, **Signed**, or **muLaw** encoding
R	8000, 11025, or 22050 samples per second

If the encoding is **muLaw**, **B** must be 8, **C** must be 1, and **R** must be 8000. If the encoding is **Raw** or **Signed**, **R** must be 11025 or 22050. Sound players should be prepared to convert between formats, downsample rates, and combine channels as necessary to render sound on the target platform.

6.16.2 External sounds

If a Sound annotation or Sound Action includes an **F** key (file specification), the file must be a *self-describing* sound file, containing all the information necessary to render the sound. No additional information need be present in the PDF file.

Note *AIFF, AIFF-C (Macintosh), RIFF (.wav) and snd (.au) files all contain sufficient information for playing.*

CHAPTER 7

Common Data Structures

Chapter 4 describes all the basic object types in PDF. In this chapter, we describe data structures that are built from these basic types but which occur so often in PDF files that it is useful to regard them as types in their own right.

7.1 Rectangle

Rectangles are used to describe locations on the page and bounding boxes for several objects in PDF, such as fonts. A rectangle is represented as an array of four numbers, [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.

7.2 Date

PDF 1.1

PDF defines a standard date format. The PDF date format closely follows the format defined by the international standard ASN.1 (Abstract Syntax Notation One, defined in CCITT X.208 or ISO/IEC 8824). A date is a string of the form:

(D:*YYYYMMDDHHmmSSOHH'mm'*)

where

- *YYYY* is the year.
- *MM* is the month (01–12).
- *DD* is the day (01–31).
- *HH* is the hour (00–23).
- *mm* are the minutes (00-59).
- *SS* are the seconds (00-59).
- *O* is the relation of local time to GMT, where **+** indicates that local time is later than GMT, **-** indicates that local time is earlier than GMT, and **Z** indicates that local time is GMT.

- *HH* is the absolute value of the offset from GMT in hours. The quote (') is part of the syntax.

- *mm* is the absolute value of the offset from GMT in minutes. The quote (') is part of the syntax.

Example:

```
D:199512231952-08'00'
```

The `D:` prefix permits arbitrary keys to be recognized as dates. However, it is not required. Trailing fields other than the year are also optional. The default value for day and month is 1; all other numerical fields default to 0. If no GMT information is specified, the relationship of the specified time to GMT is considered unknown. Whether the time zone is known or not, the rest of the date should be specified in local time.

7.3 File specification

PDF 1.2

A file specification, together with a file system, describes the location of a file. A *simple* file specification is one that does not specify the file system to be used, and a *full* file specification includes information that selects one or more file systems. Simple file specifications are strings with a standard format for representing the name of the referenced file; the format is independent of operating system naming conventions.

7.3.1 File specification strings

The standard format for a simple file specification divides the string into component strings separated by the slash character (`/`). The slash is used as a generic component-separator that is mapped to the appropriate separator when generating a system-dependent file name. The component string may be empty, and if the component string contains one or more slashes (e.g., “in/out”) then each such literal slash must be preceded by a backslash:

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

Note that the backslash must itself be preceded by a backslash to indicate it is being used as a character in the string and not as the escape character. The backlashes are removed in defining the components; they are needed only to distinguish the component values from the component separators. The component strings are stored as octets and are passed to the operating system without interpretation or conversion of any sort.

A simple file specification that begins with a slash is an *absolute* file specification. Within an absolute file specification, the last component is the file name, and the preceding components are the context. The file name may be empty in some file specifications; for example, URL specifications can specify directories instead of

files. A file specification that begins with a component (i.e., one that does not begin with a slash) is a *relative* file specification. A relative file specification is relative to the file specification of the document that contains the relative file specification.

In the case of a URL file system, the rules of RFC 1808, *Relative Uniform Resource Locators* [15], are used to compute an absolute URL from the document's file specification and a relative file specification. Prior to this process, the relative file reference is converted into a relative URL by using the escape mechanism of RFC 1738, *Uniform Resource Locators* [12], to represent any octets that would be either "unsafe" according to RFC 1738 or not representable in 7-bit US ASCII. In addition, such URL-based relative file references are limited to being paths as defined in RFC 1808; the scheme, network location/login, fragment identifier, query information, and parameters are not allowed.

In the case of other file systems, an absolute file specification is created from a relative file specification and the file specification of the document that contains the relative file specification by removing the file name component of the document's file specification and appending the relative file specification.

The special component "`..`" allows condensing of a file specification. Proceeding from left to right, whenever a component that is not "`..`" is followed by "`..`", that component and the "`..`" are eliminated from the file specification, and the process is begun again. This allows relative file specifications that are relative to an initial segment of an absolute file specification.

The conversion of a file specification into a system-dependent file name is specified for each file system. For the Macintosh, the components are separated by colons (`:`). For UNIX, the components are separated by slashes, and an initial slash, if present, is preserved. 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 of the file specification; the first component is the server name and the second component is the share name (volume name). All the 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.)

Table 7.1 provides examples of file specifications on various platforms.

Table 7.1 *Examples of file specifications*

System	*System-dependent path*	*Written as…*
Macintosh	`Macintosh HD:PDFDocs:spec.pdf`	`(/Macintosh HD/PDFDocs/spec.pdf)`
DOS	`\pdfdocs\spec.pdf` *(no drive)*	`(//pdfdocs/spec.pdf)`
DOS	`r:\pdfdocs\spec.pdf`	`(/r/pdfdocs/spec.pdf)`

Table 7.1 *Examples of file specifications*

System	*System-dependent path*	*Written as…*
DOS	`pcadobe/eng:\pdfdocs\spec.pdf`	`(/pcadobe/eng/pdfdocs/spec.pdf)`
UNIX	`/user/fred/pdfdocs/spec.pdf`	`(/user/fred/pdfdocs/spec.pdf)`
UNIX	`pdfdocs/spec.pdf` *(relative)*	`(pdfdocs/spec.pdf)`

Multi-byte strings in file specifications

PDF 1.2

A string used to specify a file name may contain multi-byte character codes. Since the slash character <2F> is used as a delimiter for components of a file name, and the backslash character <5C> is used as an escape character, any occurrence of either of these bytes in a multi-byte character must be preceded by the ASCII byte for the backslash character.

For example, a file name that contains the double-byte character code <89 5C> must be written as <89 5C 5C>. When the viewer processes a file name with this sequence of bytes, it replaces the sequence with the original double-byte code.

The strings used to specify a file name are interpreted in the platform encoding where the document is being viewed. Where system portability is required, it is recommended that file names consisting only of ASCII characters be used; see Section 7.3.4, "Safe path names".

7.3.2 File specification dictionaries

A file specification can be either a string, formatted as described above, or a dictionary. The dictionary form of the file specification provides for platform-specific file specifications and allows extension of the form of file specifications. A dictionary that contains a platform-specific file system key or a file system key (**FS**) is a *full* file specification. This provides alternate ways to locate a file.

A viewer should use the appropriate platform-specific key (**Mac**, **DOS**, or **Unix**). If it does not find the appropriate platform-specific key and there is no file system value (**FS**), it should treat the value of the file specification key (**F**) as a simple file specification. The keys need not specify the same file, allowing a single file specification to describe appropriate but different files for different platforms.

Table 7.2 describes the file specification dictionary attributes.

Table 7.2 *File specification attributes*

Key	*Type*	*Semantics*
FS (FileSystem)	name	*(Optional)* The name of the "file system" to be used to interpret this file specification. A viewer or plug-in can register a file system. A *file system* interprets file specifications, opens files, and provides the usual input and output operations. If a file specification includes a file system, all other keys are interpreted by this file system. Note that this key is independent of the **F**, **Mac**, **DOS**, and **Unix** keys.
F (File)	string	*(Required if no other keys are present)* A file specification using the string format described earlier in this section, or (if the file system is **URL**) a URL, described in the next section. A viewer that encounters an action with no **F** key and that does not understand any of the alternative keys need not do anything.
Mac	string	*(Optional)* A string that specifies a Macintosh file name using the string format described above.
DOS	string	*(Optional)* A string that specifies a DOS file name using the string format described above.
Unix	string	*(Optional)* A string that specifies a UNIX file name using the string format described above.
ID	array	*(Optional)* An array of two strings. The ID is a file ID as described in Section 6.13. This allows a viewer to find the intended file more often, and it allows viewers to warn a user if the file has changed since the link was made.
V (Volatile)	Boolean	If **V** is *true*, this indicates that the document referenced by the file specification changes frequently with time. An implementation can use this value to determine whether it is safe to use a cached copy of a document. For example, a Movie annotation could reference a URL to a live camera; if **V** is *true*, then the implementation could determine that it should reacquire the Movie each time it is played. The default value is *false*. **PDF 1.2**

The string values of the **DOS**, **Mac** and **Unix** keys should not be modified by the implementation.

7.3.3 URL

When the **FS** key in a file-specification dictionary has the value **URL**, the value of the **F** key in that dictionary is not a file specification string: instead, it is a URL formatted as specified in RFC 1738 and must follow the character-encoding requirements of that RFC. Because 7-bit US ASCII is a strict subset of the **PDFDocEncoding**, this value may also be considered to be in the **PDFDocEncoding**.

Note *Protocols most expected to be seen in PDF are "http" and "ftp".*

Example 7.1 *URLs*

```
/Movie                          % relative URL
<</F (AbbeyRoad.mov) >>

/Movie                          % absolute URL
<</F (/Movies/Beatles/AbbeyRoad.mov) >>

/Movie                          % relative URL
<</F <</FS /URL /F (AbbeyRoad.mov) >> >>

/Movie                          % absolute URL
<</F
<<
/FS /URL
/F (ftp://oranda/ftp/Movies/AbbeyRoad.mov)
>>
>>
```

7.3.4 Safe path names

Care must be taken to use *safe* path names when creating collections of documents that are to be used on various file systems. A safe path name is one that can be used to locate files on the most common file systems. For maximum compatibility, only a subset of the US ASCII character set should be used: the uppercase alphabetic (A-Z) characters, the numeric characters (0-9), and the underscore (_). The period (.) has special meaning as part of a relative path specifier in DOS and Windows file names, and as the first character in a Macintosh pathname. When used in file names, the period should be used only to separate a base file name from a file extension. Some systems are case-insensitive, so names within a directory should be 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 eight characters plus a three-character extension. File system software typically converts long names to short names by retaining the first six or seven characters and the first three characters after the last period, if any. The seventh or eighth characters are converted to other values unrelated to the original value. Therefore, safe file names are distinguishable from the first six characters.

7.4 Resources Dictionaries

The marking operations for drawing a page are stored in a stream that is the value of the **Contents** key in the Page object's dictionary. In addition to pages, however, there are other objects in PDF that also include streams of marking operations: forms, patterns, and the procedures that draw characters in Type 3 fonts. These are all referred to as *marking contexts*. Chapter 8 describes the operations used in all marking contexts, including pages.

The marking operations use various types of basic objects such as numbers and strings; these are represented as direct objects in the contents stream. Other objects, such as fonts, that are represented by streams or dictionaries may also be needed by the marking operations, but no indirect objects of any sort, including streams, may appear in a contents stream. Neither may dictionaries, with the exception of in-line property lists (see Section 7.16 on page 206). Instead, such objects are represented by names, and they are called *named resources*.

Each marking context includes a list of the named resources it uses. This resource list is stored as a dictionary that is the value of the context's **Resources** key, and it serves two functions: it enumerates the named resources in the contents stream, and it establishes the mapping from the names to the objects used by the marking operations.

For example, if the Times-Roman font were needed by a text operator, it might be referred to in the contents stream by the name `/F42`. The Resources dictionary would include the mapping from the name `/F42` to the actual font object.

PDF defines several types of named resources. These include:

- ProcSet (page 140)
- Font (page 141)
- Color space (page 169)
- XObject (page 175)
- Extended graphics state (page 189)
- Pattern (page 201)
- Property list (page 206)

Other types of objects may be used during marking operations, but they are not referred to directly and are therefore not named. These include:

- Encoding (page 156)
- Font descriptor (page 161)
- Halftone (page 191)
- Function (page 185)
- CMap (page 157)

Each key in the Resources dictionary is the name of a *resource type*; each value is a dictionary or an array. If it is a dictionary, its keys are the *resource names*, and its values are indirect references to the PDF objects specifying those resources. If it is an array, it contains a list of names. Only ProcSet resources are represented as arrays in the Resources dictionary; all other resource lists are represented as dictionaries.

Example 7.2 shows a Resources dictionary containing ProcSets, Fonts, and XObjects. The ProcSets are specified by an array, as described in the following section. The Fonts are specified with a dictionary; it contains four names, /F5, /F6, /F7, and /F8, and these are associated with objects 6, 8, 10, and 12, respectively, which are fonts. Likewise, the XObject dictionary contains the names /Im1 and /Im2, associated with objects 13 and 15, respectively, which are XObjects.

Example 7.2 *Resources dictionary*

```
<<
/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 >>
>>
```

Note *The phrase "current Resources dictionary" refers to the Resources dictionary of the current marking context. Since Page objects can inherit their attributes (see page 76), the current Resources dictionary for a Page may be stored in some ancestor of the Page object.*

7.5 ProcSets

The types of instructions that may be used in a PDF marking context are grouped into independent sets of related instructions. Each of these sets, called a ProcSet, may or may not be used in a particular context. ProcSets contain implementations of the PDF operators and are used only when a page or other context is printed. The Resources dictionary for each context must contain a **ProcSet** key whose value is an array consisting of the names of the ProcSets used in that context. Each of the entries in the array must be one of the predefined names shown in Table 7.3. For an example of a **ProcSet** key, see the Resources dictionary in Example 7.2.

Table 7.3 *Predefined ProcSets*

ProcSet Name	*Required if the page has any…*
/PDF	marks in the context whatsoever
/Text	text
/ImageB	grayscale images or image masks
/ImageC	color images
/ImageI	indexed images (also called color-table images)

7.6 Fonts

A font is represented in PDF as a dictionary specifying the type of font, its real name, its encoding, and information that can be used to provide a substitute when the font is not available. PDF defines the following types of fonts:

- Type 1
- an instance of a multiple master Type 1 font
- a subset of a Type 1 font **PDF 1.1**
- Type 3
- TrueType
- a subset of a TrueType font
- Type 0 **PDF 1.2**
- CIDFont Type 0 **PDF 1.2**
- CIDFont Type 2 **PDF 1.2**

Any font may include a **Name** attribute; this is required only in PDF 1.0. The name is used as the operand of the **Tf** operator when selecting the font. If **Name** is supplied, it should match the name used in the Font dictionary within the current Resources dictionary.

Most fonts specify the *width* of the characters in the font. The width of a character refers to the horizontal offset between the origin of the character and the origin of the next character when writing in horizontal mode, as shown in Figure 7.1. In horizontal mode, the vertical offset is always 0.

Figure 7.1 *Character metrics*

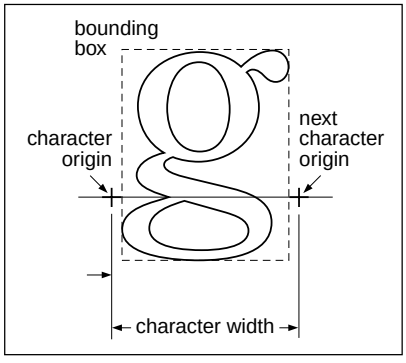


Figure 7.2 shows the relationship between fonts, encodings, CMaps, and descriptors. Three types of fonts are shown: Type 1, TrueType, and Type 0. Note that the descriptors use different keys to refer to an embedded font file. (For illustration purposes only, the figure shows nested direct objects instead of indirect objects.)

The term *glyph* is used to refer to an element of a font, particularly in a multi-byte font, where the term *character*, which is the more common term, might be confused with *byte*.

7.6.1 Type 1 fonts

Type 1 fonts, described in detail in *Adobe Type 1 Font Format* [6], are special-purpose PostScript language programs used for defining fonts. As compared to Type 3 fonts, Type 1 fonts can be defined more compactly, make use of a special procedure for drawing the characters that results in higher quality output at small sizes and low resolution, and have a built-in mechanism for specifying hints, which are data that indicate basic features of the character shapes not directly expressible by the basic PostScript language operators. In addition, a Type 1 font that contains a UniqueID in the font itself can be cached across jobs, potentially resulting in enhanced performance. See Section 2.5 of the *Adobe Type 1 Font Format* for further information on UniqueIDs for Type 1 fonts.

Table 7.4 shows the attributes of Type 1 fonts.

Note *Character widths in Type 1 fonts are measured in units in which 1000 units correspond to 1 unit in text space.*

Table 7.4 *Type 1 font attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **Type1**.
BaseFont	name	*(Required)* A PostScript language name specifying the base font, described below.

Figure 7.2 *Fonts, encodings, CMaps, and descriptors*

```
<< /Type /Font
   /Subtype /Type1
   /Encoding << /Type /Encoding
                /Differences [ … ]
                /BaseEncoding … >>
   /FirstChar …
   /LastChar …
   /FontDescriptor  << /Type /FontDescriptor
                       /Flags …
                       …
                       /FontFile stream >>
   … >>

<< /Type /Font
   /Subtype /TrueType
   /Encoding << /Type /Encoding
                /Differences [ … ]
                /BaseEncoding … >>
   /FirstChar …
   /LastChar …
   /FontDescriptor << /Type /FontDescriptor
                      /Flags …
                      …
                        /FontFile2 stream >>
   … >>

<< /Type /Font
   /Subtype /Type0
   /Encoding << /Type /CMap
                /CIDSystemInfo …
                … >>
   /DescendantFonts [ << /Type /Font
                         /Subtype /CIDFontType0
                         /CIDSystemInfo …
                         /FontDescriptor << /Type /FontDescriptor
                                            /Flags …
                                            …
                                            /FontFile3 stream >>
                         /BaseFont … >>
                      …  ]
   … >>
```

FirstChar	integer	*(Required except for base 14 Type 1 fonts listed in Table 7.5)* Specifies the first character code defined in the font's **Widths** array.
LastChar	integer	*(Required except for base 14 Type 1 fonts)* Specifies the last character code defined in the font's **Widths** array.
Widths	array	*(Required except for base 14 Type 1 fonts; indirect reference preferred)* An array of (**LastChar** – **FirstChar** + 1) widths. For character codes outside the range **FirstChar** to **LastChar**, the value of **MissingWidth** from the font's descriptor is used (see page 161). The units in which character widths are measured depend on the type of font.
Encoding	dictionary	*(Optional)* An Encoding dictionary that specifies the font's character encoding. If this key is not present, the font's built-in encoding is used. Appendix C describes the predefined encodings (**MacRomanEncoding**, **MacExpertEncoding**, and **WinAnsiEncoding**).
FontDescriptor	dictionary	*(Required except for base 14 fonts; must be indirect reference)* A font descriptor describing the font's metrics other than its character widths.

The PostScript language name of a font is the name which, in a PostScript language program, is used as an operand of the **findfont** operator. It is the name associated with the font by a **definefont** operation. This is usually the value of the **FontName** key in the PostScript language font dictionary of the font. For more information, see Section 5.2 of the *PostScript Language Reference Manual, Second Edition*.

Example 7.1 shows the font dictionary for the Adobe Garamond Semibold font. In this example, the font is given the name `/F1`, by which it would be referred to in a PDF page description. The font has an encoding (object 25), although neither the encoding nor the font descriptor (object 7) is shown in the example.

Example 7.1 *Type 1 font and character widths array*

```
14 0 obj
<<
/Type /Font
/Subtype /Type1
/Name /F1
/BaseFont /AGaramond-Semibold
/Encoding 25 0 R
/FontDescriptor 7 0 R
/FirstChar 0
/LastChar 255
/Widths 21 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
```

7.6.2 The base 14 Type 1 fonts

Some font attributes can be omitted for the fourteen Type 1 fonts guaranteed to be present with Acrobat Exchange and Acrobat Reader. These fonts are called the base 14 fonts and include members of the Courier, Helvetica, and Times families, along with Symbol and ITC Zapf Dingbats. Table 7.5 lists the PostScript language names of these fonts.

Table 7.5 *Base 14 fonts*

Courier	Symbol
Courier-Bold	Times-Roman
Courier-Oblique	Times-Bold
Courier-BoldOblique	Times-Italic
Helvetica	Times-BoldItalic
Helvetica-Bold	ZapfDingbats
Helvetica-Oblique	
Helvetica-BoldOblique	

7.6.3 TrueType fonts

The TrueType font format was developed by Apple Computer. A TrueType font, shown in Table 7.6, has the same keys as a Type 1 font. The **Subtype** and **BaseFont** keys have different values than those in a Type 1 font.

Note *Character widths in TrueType fonts are measured in units in which 1000 units correspond to 1 unit in text space.*

Table 7.6 *TrueType font attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **TrueType**.
BaseFont	name	*(Required)* Style string specifying the base TrueType font, described below.
FirstChar	integer	*(Required)* The same as for Type 1.
LastChar	integer	*(Required)* The same as for Type 1.
Widths	array	*(Required; indirect reference preferred)* The same as for Type 1.
Encoding	dictionary	*(Optional)* The same as for Type 1.
FontDescriptor	dictionary	*(Required; must be indirect reference)* The same as for Type 1.

For TrueType fonts, the **BaseFont** key in the font dictionary may contain a *style string*. If the font is a bold, italic, or bold italic font for which no PostScript language name is available, the **BaseFont** key contains the base name of the font with any spaces removed, followed by a comma, followed by a style string. The style string is one of the strings “Italic”, “Bold”, or “BoldItalic”. For example, the italic variant of the New York font has a **BaseFont** that is written as

```
/NewYork,Italic
```

Example 7.1 *TrueType font*

```
17 0 obj
<<
/Type /Font
/Subtype /TrueType
/Name /F1
/BaseFont /NewYork,Bold
/FirstChar 0
/LastChar 255
/Widths 23 0 R
/Encoding /MacRomanEncoding
```

```
/FontDescriptor 7 0 R
>>
endobj
23 0 obj
[0 333 333 333 333 333 333 333 0 333 333 333 333 333
333 333 333 333 333 333 333 333 333 333 333 333 333
333 333 0 333 333 333 303 500 666 666 882 848 303 446
446 507 666 303 378 303
```
… omitted data …
```
303 530 1280 757 605 757 605 605 355 355 355 355 803
803 790 803 780 780 780 340 636 636 636 636 636 636
636 636 636 636]
endobj
```

7.6.4 Font Subsets

PDF 1.1

PDF permits documents to include *subsets* of Type 1 and TrueType fonts. The font and the 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.

The value of the font's **BaseFont** key and the font descriptor's **FontName** key use the following format:

pseudoUniqueTag+PostScriptName

pseudoUniqueTag consists of exactly six uppercase alphabetic characters. *PostScriptName* must be the name of the complete Type 1 font. A plus sign separates *pseudoUniqueTag* and *PostScriptName*. For example, `EOODIA+Poetica` is the name of a subset of the Poetica font. The purpose of the tag is to identify the subset. Different subsets should have different tags.

Note *Any font whose* **BaseFont** *or* **FontName** *uses this format is assumed to be a font subset.*

Implementation note *These restrictions make font subsets compatible with 1.0 viewers, enable the Distiller application to recognize font subsets in its input stream, and enable Acrobat 2.0 viewers to merge documents containing subsets.*

7.6.5 Multiple master Type 1 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. 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.

To specify the appearance of the font, numeric values must be supplied for each design dimension of the multiple master font. A completely specified multiple master font is referred to as an *instance* of the multiple master font.

The note *Adobe Type 1 Font Format: Multiple Master Extensions* [7] describes multiple master fonts. An instance of a multiple master font, shown in Table 7.7, has the same keys as an ordinary Type 1 font.

Note *Character widths in multiple master Type 1 fonts are measured in units in which 1000 units correspond to 1 unit in text space.*

Table 7.7 *Multiple master Type 1 font additional attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **MMType1**.
BaseFont	name	*(Required)* Specifies the PostScript language name of the instance. If the name contains spaces (such as “MinionMM 366 465 11”), these spaces are replaced with underscores.
FirstChar	integer	*(Required)* The same as for Type 1.
LastChar	integer	*(Required)* The same as for Type 1.
Widths	array	*(Required; indirect reference preferred)* The same as for Type 1.
FontDescriptor	dictionary	*(Required; must be indirect reference)* The same as for Type 1.
Encoding	dictionary	*(Optional)* The same as for Type 1.

Example 7.2 *Multiple master font and character widths array*

```
7 0 obj
<<
/Type /Font
/Subtype /MMType1
/Name /F4
/BaseFont /MinionMM_366_465_11_
/FirstChar 32
/LastChar 255
/Widths 19 0 R
/Encoding 5 0 R
/FontDescriptor 6 0 R
>>
endobj
19 0 obj
[187 235 317 430 427 717 607 168 326 326 421
619 219 317 219 282 427 427 427 427 427 427
427 427 427 427 219 219 619 619
```

```
… omitted data …
301 301 301 569 569 0 569 607 607 607 239 400
400 400 400 253 400 400 400 400 400]
endobj
```

7.6.6 Type 3 fonts

PostScript Type 3 fonts, also known as user-defined fonts, are described in Section 5.7 of the *PostScript Language Reference Manual, Second Edition*. PDF provides a variant of Type 3 fonts in which characters are defined by streams of PDF page-marking operators. These streams, known as CharProcs, are associated with the character names. As with any font, the character names are accessed via an encoding vector.

PDF Type 3 fonts differ from the other fonts provided by PDF. A Type 3 font defines the font itself, while the other font dictionaries simply contain information about the font.

Type 3 fonts are more flexible than Type 1 fonts because the character-drawing streams may contain arbitrary PDF marking operators. However, Type 3 fonts have no mechanism for improving output at small sizes or low resolutions, and no built-in mechanism for hinting. Table 7.8 shows the attributes specific to Type 3 fonts.

Table 7.8 *Type 3 font additional attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **Type3**.
FirstChar	integer	*(Required)* The same as for Type 1.
LastChar	integer	*(Required)* The same as for Type 1.
Widths	array	*(Required; indirect reference preferred)* The same as for Type 1.
CharProcs	dictionary	*(Required)* Each key in this dictionary is a character name; the value associated with that key is a stream object that draws the character. Any operator that can be used in a PDF page description can be used in this stream. However, the stream must include as its first operator either **d0** (d zero) or **d1** (d one), equivalent to the PostScript language **setcharwidth** and **setcachedevice** operators.
FontBBox	Rectangle	*(Required)* The font's bounding box. The coordinates are measured in character space. The bounding box is the smallest rectangle enclosing the shape that would result if all characters in the font were placed with their origins coincident, and then painted. **FontBBox** is identical to the PostScript Type 3 font FontBBox.
FontMatrix	array	*(Required)* Specifies the transformation from character space to text space. **FontMatrix** is identical to the PostScript Type 3 font FontMatrix.

Resources	dictionary	*(Optional, but strongly recommended)* A list of the named resources such as fonts and images required by the CharProcs in this font (see page 138). If this dictionary is not present, then the resources used by the CharProcs are specified in the Resources dictionary of the Page on which this font is used.	PDF 1.2
Encoding	dictionary	*(Optional)* The same as for Type 1.	

Note *Character widths and* **FontBBox** *in Type 3 fonts are measured in character space. The transformation from character space to text space is specified by the value of the* **FontMatrix** *key in the Type 3 font dictionary.*

Example 7.3 *Type 3 font*

```
6 0 obj
<<
/Type /Font
/Subtype /Type3
/Name /T36
/CharProcs 1928 0 R
/FontBBox [–3 –241 875 856]
/FontMatrix [.001 0 0 .001 0 0]
/FirstChar 3
/LastChar 101
/Widths 7 0 R
/Encoding 1927 0 R
>>
endobj
7 0 obj
[55 0 0 589 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 31 31 0 0 0 270 0 0 410 40 640
40 0 40 0 40 40 0 0 0 0 0 0 0 0 60 0 58 61 54 52 603 0
29 0 0 853 73 60 62 504 0 659 44 58 60 60 0 0 603 0 0
0 0 0 0 0 0 0 35 0 35]
endobj
```

7.6.7 Type 0 Fonts

PDF 1.2

Type 0 fonts support single-byte or multi-byte encodings and can refer to one or more *descendant* fonts; see Figure 7.2 on page 143. These fonts are analogous to the Type 0 or composite fonts supported by Level 2 PostScript interpreters. However PDF Type 0 fonts only support character encodings defined by a CMap. The CMap defines the encoding for a Type 0 font. It specifies the mappings between character codes and the glyphs in the descendant fonts. The details of a CMap are defined in Section 7.8 on page 157.

The following Table shows the attributes of a Type 0 font.

Table 7.9 *Type 0 font attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **Type0**.
BaseFont	string or name	*(Required)* PostScript language name specifying the font, or a name with a style string specifying the TrueType font. A string should be used when the font name contains characters that are not legal in a name object.
DescendantFonts	array	*(Required)* An array of one or more fonts. These fonts are selected by the font number defined in the CMap.
Encoding	name or stream	*(Required)* The name of a predefined CMap, or a stream containing a CMap.
ToUnicode	stream	*(Optional)* A CMap that defines the mapping from character codes to Unicode values. This entry is recommended for fonts that do not use one of the predefined CMaps. If present this allows strings in the encoding to be converted to Unicode values for export to other applications or plug-ins.

The following example shows a Type 0 font which refers to a single CIDFont. The CMap used is one of the predefined CMaps and is referenced by name.

Example 7.4 *Type 0 font referring to a single CIDFont*

```
14 0 obj
<<
/Type /Font
/Subtype /Type0
/Name /F2
/BaseFont /HeiseiMin-W5-90ms-RKSJ-H
/Encoding /90ms-RKSJ-H
/DescendantFonts [15 0 R]
>>
endobj
```

7.6.8 CIDFonts

PDF 1.2

A CIDFont is designed to contain a large number of glyph procedures. Instead of being accessed by a name, each glyph procedure is accessed by an integer known as a *character identifier* or *CID*. A CIDFont does not have an **Encoding** attribute. It is used only as a descendant of a Type 0 font. It is the CMap in the Type 0 font that defines the encoding that maps character codes to CIDs in the CIDFont.

There are two types of CIDFonts. A CIDFont Type 0 font contains glyph procedures based on Adobe's Type 1 font format; a CIDFont Type 2 font contains glyph procedures based on the TrueType font format.

7.6.9 CIDFontType 0

A CIDFont Type 0 font uses Adobe Type 1 charstrings for glyph procedures and the CIDFont file format.

Table 7.10 *CIDFontType 0 font attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Type of object. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **CIDFontType0**.
BaseFont	name	*(Required)* PostScript language name specifying the CIDFont.
CIDSystemInfo	dictionary	*(Required)* A dictionary containing the Registry, Ordering, and Supplement strings that define the character collection of the CIDFont.
FontDescriptor	dictionary	*(Required; must be an indirect reference)* A font descriptor describing the CIDfont's default metrics other than its character widths.
DW	integer	*(Optional)* Default width for glyphs in the CIDFont.
W	array	*(Optional)* The **W** array consists of a set of lists that define the widths for the glyphs in the CIDfont. Each list can specify individual widths for consecutive CIDs, or one width for a range of CIDs. (See the section below for more details on this format.)
	Note	*CIDFonts do not have a* **Widths** *entry. Instead, all widths are defined by the* **W** *entry.*
DW2	array	*(Optional; applies only to CIDFonts that are used for vertical writing)* The default metrics for writing mode 1 (vertical writing, described below). This entry is an array of two numbers: the *y* component of the position vector and the *y* component of the displacement vector for writing mode 1. The *x* component of the position vector is always half the width of the character. The *x* component of the displacement vector is always 0. The default value is `[880 -1000]`.
W2	array	*(Optional; applies only to CIDFonts that are used for vertical writing)* This array defines the metrics for vertical writing. Its format is similar to the format of the **W** array. It defines the *x* and *y* components of the position vector, and the *y* component of the displacement vector. The *x* component of the displacement vector is always 0. The contents of this array is described in detail below.

7.6.10 CIDFontType 2

A CIDFont Type 2 font uses TrueType glyph procedures.

Table 7.11 *CIDFontType 2 font attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Type of object. Always **Font**.
Subtype	name	*(Required)* Type of font. Always **CIDFontType2**.
BaseFont	name	*(Required)* A name with a style string specifying the TrueType font.
CIDToGIDMap		*(Optional)* This entry is required if the font associated with this object is embedded in the PDF file. This entry can be a stream or a name.
	stream	If it is a stream, the bytes in the stream contain the mapping from CID to glyphindex ("GID"). The glyphindex for a particular CID value *c* is a 2-byte value stored in bytes 2*c* and 2*c*+1; the first byte is the high-order byte.
	name	If value of this key is a name, it must be **Identity**, indicating that the mapping between CIDs and glyphindices is the identity mapping.
CIDSystemInfo	dictionary	*(Required)* The same as for CIDFontType 0.
FontDescriptor	dictionary	*(Required; must be an indirect reference)* The same as for CIDFontType 0.
DW	integer	*(Optional)* Default width for glyphs in the CIDFont.
W	array	*(Optional)* The same as for CIDFontType 0.
DW2	array	*(Optional; applies only to CIDFonts that are used for vertical writing)* The same as for CIDFontType 0.
W2	array	*(Optional; applies only to CIDFonts that are used for vertical writing) The* same as for CIDFontType 0.

7.6.11 Character widths in CIDFonts

In any font, the *width* of a character (glyph) refers to the horizontal displacement between the origin of the character and the origin of the next character 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 the **W** attributes. The **DW** entry defines the default width. This entry is particularly useful for Chinese, Japanese, and Korean fonts, where many of the characters have the same width. The **W** attribute allows the definition of widths for single CIDs or a range of CIDs. The entry is an array of lists in either of the following two formats:

$c \ [w_1 \ w_2 \ \ldots \ w_n]$
$c_{first} \ c_{last} \ w$

In the first list, *c* is an integer specifying a starting CID value. It is followed by an array of numbers which specify the widths for *n* consecutive CIDs, starting with *c*. The second list defines the same width, *w*, for all the CIDs the range c_{first} to c_{last}. If the **DW** key is present, the **W** attribute need not specify the width of a CID that uses the default width.

Following is an example of a **W** attribute.

```
/W
[
120 [400 325 500]
7080 8032 1000
]
```

In this example, the CIDs 120, 121, and 122 have widths 400, 325, and 500 units respectively. The second list in this example specifies that CIDs in the range 7080 to 8032 have a width of 1,000 units.

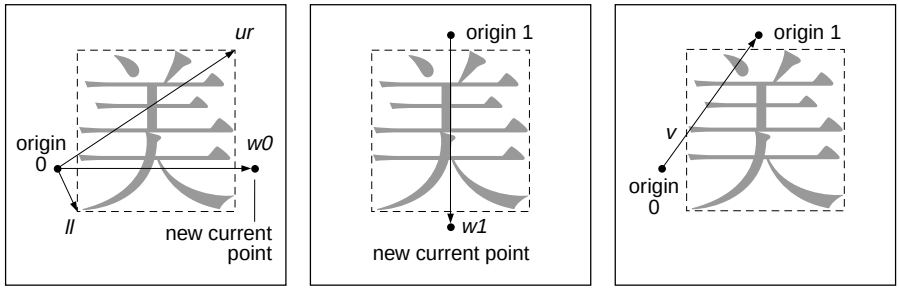
7.6.12 Vertical Writing

Vertical writing is specified by the **WMode** entry in a CMap. When this value is 1, the Type 0 font that uses the CMap positions text for vertical writing.

In vertical writing, the character 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). See the *PostScript Language Reference Manual, Second Edition*, section 5.4, for a detailed explanation of vertical writing.

Figure 7.3 illustrates the metrics for horizontal and vertical writing modes.

Figure 7.3 *Horizontal and vertical writing metrics*



The left diagram illustrates the character metrics associated with writing mode 0, horizontal writing. The coordinates *ll* and ur specify the bounding box of the character relative to origin 0. *w0* is the displacement vector that specifies how the current point is changed after the character is shown in writing mode 0. Its *x* component (horizontal displacement or *character width*), may be different from the width of the bounding box. Its *y* component (vertical displacement) is always 0.

The center diagram illustrates writing mode 1, vertical writing. *w1* is the displacement vector for writing mode 1; its *x* component is always 0.

In the right diagram, *v* is the *position vector* that defines the position of origin 1 relative to origin 0.

In vertical writing, the default position vector and displacement vector are specified by the **DW2** attribute, which is an array of two values: the *y* component of the position vector, and the *y* component of the displacement vector. The *x* value of the position vector is always half the character width. The *x* component of the displacement vector is always 0. For example, if the **DW2** entry is:

```
/DW2 [880 -1000]
```

the position vector and displacement vectors are:

$$v = (hw \div 2,\ 880)$$
$$w = (0,\ -1000)$$

where hw is the character width. Note that a negative value for the *y* component will place the origin of the next character *below* the current character, because in a standard coordinate system, the positive direction of the *y*-axis points upward.

Glyphs whose vertical metrics differ from the defaults must be included in the **W2** array. In this array, the position vector and displacement vector are defined by three numbers: the *x* and *y* values of the position vector, and the *y* component of the displacement vector. There are two formats that can be used to define these vectors, as shown below:

```
/W2 [
c [w1,y  v1,x  v1,y  w2,y  v2,x  v2,y  ...]
cfirst clast  w1,y  v1,x  v1,y
]
```

In the first entry, c gives a starting CID and is followed by an array. The array contains sets of three numbers: the *y* value for the displacement vector, followed by the *x* and *y* values for the position vector. The sets of three numbers defines the vertical metrics for consecutive character codes starting with CID value c. The second format defines a range of CIDs from c_{first} to c_{last}. These ranges use the same format as defined for the **W** array. The range is followed by three numbers that define the vertical metrics for all CIDs in the range.

Following is an example of a **W2** entry.

```
/W2
[
120 [-1000 250 772]
7080 8032 -1000 500 900
]
```

This entry defines the mode-1 (vertical) displacement vector for character CID 120 as (0,-1000) and the position vector as (250, 772). The second list in the array defines the displacement vector to be (0, -1000) and the position vector of (500, 900), for CIDs in the range 7080 to 8032.

7.7 Font encodings

An encoding describes a font's character encoding, the mapping between numeric character codes and character names. These character names are keys in the font dictionary and are used to retrieve the code which draws the character. Thus, the font encoding provides the link which associates numeric character codes with the glyphs drawn when those codes are encountered in text. An encoding is a dictionary whose contents are shown in Table 7.12.

Table 7.12 *Font encoding attributes*

Key	*Type*	*Semantics*
Type	name	*(Optional)* Object type. Always **Encoding**.
BaseEncoding	name	*(Optional)* Specifies the encoding from which the new encoding differs. This key is not present if the encoding is based on the base font's encoding. Otherwise it must be one of the predefined encodings **MacRomanEncoding**, **MacExpertEncoding**, or **WinAnsiEncoding**, described in Appendix C.
Differences	array	*(Optional)* Describes the differences from the base encoding.

The value of the **Differences** key is an array of character codes and glyph 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 characters to be changed. The first glyph name after the code becomes the name corresponding to that code. Subsequent names replace consecutive code indexes until the next code appears in the array or the array ends.

For example, in the encoding in Example 7.5, the glyph **quotesingle** (') is associated with character code 39. **Adieresis** (Ä) is associated with code 128, **Aring** (Å) with 129, and **trademark** (™) with 170.

Example 7.5 *Font encoding*

```
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
```

7.8 CMaps

PDF 1.2

A CMap is used with a Type 0 composite font to define the mapping from character codes to a font number and a character selector. (A CMap serves the same function for a Type 0 font as an Encoding does for a Type 1 font.) The font number selects the font from the **DescendantFonts** array in the Type 0 font. The character selector selects the glyph from the descendant font. A character selector can be a CID or a character code. A CMap can handle encodings that use single-byte or multi-byte character codes.

The following table shows the predefined CMap names. These CMaps map character codes to CIDs in a single descendant CIDFont.

Table 7.13 *Predefined CJK CMap names*

Name	*Description*
	Chinese (Simplified)
GB-EUC-H	Microsoft, EUC encoding, GB2312 CharSet (0x86), Code Page 936

GB-EUC-V	Vertical version of **GB-EUC-H**
GBpc-EUC-H	Macintosh, EUC encoding, Script Manager code 2
	Chinese (Traditional)
B5pc-H	Macintosh (Taiwan) Script Manager code 2
B5pc-V	Vertical version of **B5pc-H**
ETen-B5-H	Microsoft Chinese Big 5 Charset (0x88), Code Page 950
ETen-B5-V	Vertical version of **ETen-B5-H**
CNS-EUC-H	CNS 11643-1992, EUC encoding
CNS-EUC-V	Vertical version of **CNS-EUC-H**
	Japanese
83pv-RKSJ-H	Macintosh SHIFT-JIS encoding Script Manager code 1, KanjiTalk version 6.x
90ms-RKSJ-H	Microsoft SHIFT-JIS CharSet (0x80), Code Page 932, Win 3.1J, Win 95J
90ms-RKSJ-V	Vertical version of **90ms-RKSJ-H**
90pv-RKSJ-H	Macintosh SHIFT-JIS encoding Script Manager code 1, KanjiTalk version 7.x
Add-RKSJ-H	Fujitsu FMR character set, SHIFT-JIS encoding
Add-RKSJ-V	Vertical version of **Add-RKSJ-H**
Ext-RKSJ-H	NEC character set, SHIFT-JIS encoding
Ext-RKSJ-V	Vertical version of **Ext-RKSJ-H**
H	JIS encoding
Hojo-EUC-H	
Hojo-EUC-V	
V	Vertical version of **H**
	Korean
KSC-EUC-H	Microsoft (Default Win 3.1 and Win 95) Hangul Charset (0x81), Code Page 949
KSC-EUC-V	Vertical version of **KSC-EUC-V**
KSCms-UHC-H	Microsoft (Alternate in Win95) Hangul Charset (0x81), Code Page 949
KSCms-UHC-V	Vertical version of **KSCms-UHC-H**
KSCpc-EUC-H	Macintosh Korean, Script Manager Code 3
KSC-Johab-H	Microsoft, Code Page 1361
KSC-Johab-V	Vertical version of **KSC-Johab-H**
	Generic
Identity-H	The horizontal identity mapping for two byte CIDs. This may be used with CIDFonts using any Registry, Ordering and supplement. It maps two-byte character codes from 1 to 65536 to the same two-byte CID value.
Identity-V	Vertical version of the **Identity-H** mapping. The mapping is the same as for **Identity-H**. However the writing mode (**WMode**) value is set to 1 to indicate vertical writing.

For character encodings that are not predefined, the PDF file must contain a stream that defines the CMap. The stream dictionary must contain the entries defined in the table below. 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 the *Adobe CMap and CIDFont File Specification* [5]

Table 7.14 *CMap attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Must be **CMap**. (Note that although this object is the value of the key named **Encoding** in a Type 0 font, its type is **CMap**.)
CIDSystemInfo	dictionary or array	*(Required)* This entry defines the Registry, Ordering, and Supplement that define the character collection of the CIDFont. If the CMap refers to a single CIDFont then the entry may be a dictionary. If this CMap refers to more than one descendant font, then an array must be supplied. The elements of the array correspond to the font numbers of the descendant fonts. If the descendant font is a CIDFont, a dictionary containing the Registry, Ordering and Supplement must be supplied. If the descendant font is not a CIDFont, then **null** should be used for that element of the array.
WMode	integer	*(Optional)* An integer specifying the writing direction of the Type0 font that uses this CMap. A value of 0 indicates a horizontal writing direction, and a value of 1 indicates a vertical writing direction. The default is 0.
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 with only the character mappings that differ from the base CMap.

Below is a sample CMap for a Japanese Shift-JIS encoding. Character codes in this encoding can either be one or two bytes in length. In this CMap all character codes are mapped to CIDs in font number 0. 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 7.6 *CMap Encoding*

```
22 0 obj
<<
/Type /CMap
/CIDSystemInfo
<<
/Registry (Adobe)
/Ordering (Japan1)
/Supplement 2
>>
/CMapName /90ms-RKSJ-H
/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: 1.402
%%Copyright: Copyright 1990-1994 Adobe Systems Inc.
%%Copyright: All Rights Reserved.
%%Copyright:
%%Copyright: Patents Pending
%%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 1 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
```

Note *A CMap contained in a PDF file must not use the* **usematrix**, **beginrearrangedfont**, *or* **endrearrangedfont** *operators.*

7.9 Font descriptors

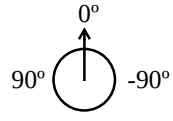
A font descriptor specifies a font's metrics, attributes, and glyphs. These metrics provide information needed to create a substitute multiple master font when the original font is unavailable. The font descriptor may also be used to embed the original font in the PDF file.

A font descriptor is a dictionary, as shown in Table 7.15, whose keys specify various font attributes. Most keys are similar to the keys found in Type 1 font and FontInfo dictionaries described in Section 5.2 of the *PostScript Language Reference Manual, Second Edition* and the *Adobe Type 1 Font Format*. All integer values are units in character space. The conversion from character space to text space depends on the type of font. See the discussion in Section 7.6, "Fonts."

Note *For detailed information on the coordinate system in which characters are defined, see Section 5.4 in the* PostScript Language Reference Manual, Second Edition *or Section 3.1 in the* Adobe Type 1 Font Format.

Table 7.15 *Attributes shared by all font descriptors*

Key	*Type*	*Semantics*
Type	name	(*Required*) Object type. Always **FontDescriptor**.
Ascent	integer	(*Required*) The maximum height above the baseline reached by characters in this font, excluding the height of accented characters.
CapHeight	integer	(*Required*) The *y*-coordinate of the top of flat capital letters, measured from the baseline.

Descent	integer	(*Required*) The maximum depth below the baseline reached by characters in this font. **Descent** is a negative number.	
Flags	integer	(*Required*) Collection of flags defining various characteristics of the font. See Table 7.17.	
FontBBox	Rectangle	(*Required*) The font's bounding box, which is the smallest rectangle enclosing the shape that would result if all characters in the font were placed with their origins coincident, and then painted.	
FontName	name	(*Required*) The name passed to the PostScript language **definefont** operator. (See page 147 for restrictions on the name.)	
ItalicAngle	integer	(*Required*) Angle in degrees counterclockwise from the vertical of the dominant vertical strokes of the font. **ItalicAngle** is negative for fonts that slope to the right, as almost all italic fonts do. 	
StemV	integer	(*Required*) The width of vertical stems in characters.	
AvgWidth	integer	(*Optional*) The average width of characters in this font. The default value is 0.	
FontFile	stream	(*Optional*) A stream that defines a Type 1 font.	PDF 1.1
FontFile2	stream	*(Optional)* A stream that defines a TrueType font.	PDF 1.1
FontFile3	stream	*(Optional)* A stream that contains an embedded font. The format of the font is specified by the **Subtype** in the stream attributes dictionary.	PDF 1.2
Leading	integer	(*Optional*) The desired spacing between lines of text. The default value is 0.	
MaxWidth	integer	(*Optional*) The maximum width of characters in this font. The default value is 0.	
MissingWidth	integer	(*Optional*) The width to use for unencoded character codes. The default value is 0.	
StemH	integer	(*Optional*) The width of horizontal stems in characters. The default value is 0.	
XHeight	integer	(*Optional*) The *y*-coordinate of the top of flat non-ascending lowercase letters, measured from the baseline. The default value is 0.	
CharSet	string	*(Optional)* A string which lists the glyph names corresponding to the entries in the CharStrings dictionary if the font described is a subset font. Each name must be preceded by a slash. The names may appear in any order. The name **.notdef** should be omitted; it is assumed to exist in the font subset.	PDF 1.1

7.9.1 Font files

Currently, a multiple master Type 1 font can be used to substitute only for fonts that use the Adobe Roman Standard Character Set as defined in Appendix E.5 of the *PostScript Language Reference Manual, Second Edition*. To make a document

portable, it is necessary to embed fonts that do not use this character set. The only exceptions are the fonts Symbol and ITC Zapf Dingbats, which are assumed to be present.

Fonts are embedded using the FontFile mechanism. If the embedded font is referred to by the name **FontFile**, then the font must be in the standard Type 1 format. If the embedded font is referred to by the name **FontFile2**, then the font must be in the TrueType format. If the embedded font is referred to by the name **FontFile3**, then the format of the font is specified by the **Subtype** key in the FontFile stream attributes dictionary.

A standard Type 1 font definition, as described in the *Adobe Type 1 Font Format* [6], consists of three parts: a clear text portion, 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. The stream dictionary for a font file contains the standard **Length** and **Filter** keys plus the additional keys shown in Table 7.16. While the encrypted portion of a standard Type 1 font may be in binary or ASCII hexadecimal format, PDF supports only the binary format. Example 7.7 shows the structure of an embedded standard Type 1 font.

Type 1 Compact fonts [8] may be embedded in a PDF 1.2 file using the **FontFile3** key. The **Subtype** value must by **Type1C**. **PDF 1.2**

TrueType fonts are embedded using the FontFile2 mechanism. The font descriptor for an embedded TrueType font should contain a **FontFile2** key whose value is a stream that contains the TrueType font definition as described in *TrueType 1.0 Font Files*. The stream dictionary should include a **Length1** key as specified in Table 7.16; that key specifies the length in bytes of the font file after it has been decoded using the filters specified by the stream's **Filter** key. The **Length2** and **Length3** keys should not be used for TrueType fonts.

Because the stream containing Type 1 or TrueType font data may include binary data, it may be desirable to convert this data to ASCII using either the ASCII hexadecimal or ASCII base-85 encoding.

Table 7.16 *Additional attributes for FontFile stream*

Key	*Type*	*Semantics*
Length1	integer	(*Required*) Length in bytes of the ASCII portion of the Type 1 font file after it has been decoded using the filters specified by the stream's **Filter** key.
Length2	integer	(*Required for Type 1 fonts*) Length in bytes of the encrypted portion of the Type 1 font file after it has been decoded using the filters specified by the stream's **Filter** key.
Length3	integer	(*Required for Type 1 fonts*) Length in bytes of the portion of the Type 1 font file that contains the 512 zeros, plus the **cleartomark** operator, plus any following data. This is the length of the data after it has been decoded using the filters specified by the stream's **Filter** key. If **Length3** is zero, it indicates that the 512 zeros and **cleartomark** have not been included in the **FontFile** and must be added.

Subtype	name	*(Required if in FontFile3)* The name specifies the format of the embedded font. The name must be **Type1C** for Type 1 Compact fonts. When additional font formats are added to PDF, more values will be defined for **Subtype**.	**PDF 1.2**

Note *Font descriptors for Type 0 fonts contain additional attributes. See* *Section 7.9.3 on page 166*.

Example 7.7 *Embedded Type 1 font definition*

```
12 0 obj
<<
/Filter /ASCII85Decode
/Length 13 0 R
/Length1 15 0 R
/Length2 14 0 R
/Length3 16 0 R
>>
stream
,p>`rDKJj'E+LaU0eP.@+AH9dBOu$hFD55nC
… omitted data …
JJQ&Nt')<=^p&mGf(%:%h1%9c//K(/*o=.C>UXkbVGTrr~>
endstream
endobj
13 0 obj
41116
endobj
14 0 obj
32393
endobj
15 0 obj
2526
endobj
16 0 obj
570
endobj
```

7.9.2 Font descriptor flags

The value of the **Flags** key in a font descriptor is a 32-bit integer that contains a collection of Boolean attributes. These attributes are *true* if the corresponding bit is set to 1 in the integer. Table 7.17 specifies the meanings of the bits, with bit 1 being the least significant. Reserved bits must be set to zero.

Table 7.17 *Font flags*

Bit position	*Semantics*
1	Fixed-width font

2	Serif font
3	Symbolic font
4	Script font
5	Reserved
6	Uses the Adobe Standard Roman Character Set
7	Italic
8–16	Reserved
17	All-cap font
18	Small-cap font
19	Force bold at small text sizes
20–32	Reserved

All characters in a *fixed-width font* have the same width, while characters in a proportional font have different widths. Characters in a *serif font* have short strokes drawn at an angle on the top and bottom of character stems, while sans serif fonts do not have such strokes. A *symbolic font* contains symbols rather than letters and numbers. Characters in a *script font* resemble cursive handwriting. An *all-cap font*, which is typically used for display purposes such as titles or headlines, contains no lowercase letters. It differs from a *small-cap font* in that characters in the latter, while also capital letters, have been sized and their proportions adjusted so that they have the same size and stroke weight as lowercase characters in the same typeface family. Figure 7.4 shows examples of these types of fonts.

Figure 7.4 *Characteristics represented in the Flags field of a font descriptor*

The quick brown fox jumped	Fixed-width font
The quick brown fox jumped…	Sans serif font
The quick brown fox jumped…	Serif font
✶❋❋ ❑◆❊❋✳ ✲❒❑◗■ ❄❑❙ ❊◆❍❒❋❄⑩	Symbolic font
The quick brown fox jumped…	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

Bit 6 in the flags field indicates that the font's character set is the Adobe Standard Roman Character Set, or a subset of that, and that it uses the standard names for those characters. The characters in the Adobe Standard Roman Character Set are shown in the first column of Table C.1 on page 338 (A, Æ, Á, etc.); the character names are shown in column 2 (**A**, **AE**, **Aacute**, etc.).

PDF 1.1

Finally, bit 19 is used to determine whether or not bold characters are drawn with extra pixels even at very small text sizes. Typically, when characters are drawn at small sizes on very low resolution devices such as display screens, features of bold characters may appear only one pixel wide. Because this is the minimum feature width on a pixel-based device, ordinary non-bold characters also appear with one-pixel wide features, and cannot be distinguished from bold characters. If bit 19 is set, features of bold characters may be thickened at small text sizes.

Example 7.1 *Font descriptor*

```
7 0 obj
<<
/Type /FontDescriptor
/FontName /AGaramond-Semibold
/Flags 262192              % Bits 5, 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
```

7.9.3 Font descriptors for Type 0 fonts

PDF 1.2

The following table lists the additional entries that may be found in the FontDescriptor dictionaries of Chinese, Japanese or Korean fonts, which are represented with Type 0 fonts.

Table 7.18 *Additional FontDescriptor attributes*

Key	*Type*	*Semantics*
Style	dictionary	*(Optional)* This entry is a dictionary that contains key-value pairs that describe the style of the glyphs in the font. See the table below for valid entries in this dictionary.
Lang	name	*(Optional)* This entry indicates the language of the font. It is used for encodings where the language is not implied by the encoding itself. The possible values are the two character names defined by ISO639; see Appendix I.
FD	dictionary	*(Optional)* The keys in this dictionary identify a subset of characters in a CIDFont. The values are dictionaries with entries that override the values in the FontDescriptor dictionary for the subset of characters.

CIDSet	stream	*(Optional)* This entry identifies which CIDs are present in the CIDFont file. It is required if the CIDFont file is embedded and only contains a subset of the glyphs in the character collection defined by the CIDSystemInfo. If this entry is missing, then it is assumed that the CIDFont file contains all the glyphs for the character collection. The stream's length should be rounded up to the nearest multiple of 8. The bits should be stored in bytes with the high-order bit first. Each bit corresponds to a CID. The first bit of the first byte corresponds to CID 0, the next bit corresponds to CID 1, and so on. If the subset contains a CID, the bit for that CID should be set. For compactness the stream may use one of the compression filters to encode the data.
FontFile3	stream	*(Optional)* A complete or subset version of the CIDFont file.

Style

The Style dictionary contains a set of key-value pairs that define style attribute and value for the font. Currently only the **Panose** key is defined. The value of the **Panose** key is the 12-byte string containing the Family Class ID, Family SubClass ID, and 10 bytes for the Panose classification number for the font. For additional details on the Panose number see the *TrueType 1.0 Font Files* specification from Microsoft. Following is an example of a **Style** entry in the FontDescriptor:

```
/Style
<< /Panose <01 05 02 02 03 00 00 00 00 00 00 00>
>>
```

FD

A CIDFont may be made up of different types of characters, each type requiring different sets of metrics. Numeric characters, for example, may require different metrics from ideographic characters. The font descriptor defines a set of default metrics that apply to all characters in the CIDFont; the **FD** entry in the font descriptor contains exceptions to these defaults. Each key in an **FD** dictionary is the name of a type of character; each type is defined as a particular subset of characters in the font.

It is strongly recommended that the **FD** dictionary contain at least the metrics for the proportional Roman characters. With the information for the proportional Roman characters, a more accurate substitution font can be created. For CIDFonts that use the Adobe-Japan1 character collection, the name for the proportional Roman characters is **Proportional**. For CIDFonts that use the Adobe-CNS1, Adobe-GB1, and Adobe-Korea1 character collections, the name for the proportional Roman characters is **Alphabetic**.

The following table gives the valid keys for the Adobe-Japan1, Adobe-Korea1, Adobe-Japan2, Adobe-GB1, and Adobe-CNS1 character collections.

Table 7.19 *Character Subsets in CJK fonts*

Character collection	*Key*	*Character subset*
Adobe-Japan1		
	AlphaNum	Numeric characters
	Alphabetic	Full-width Roman characters
	Dingbats	Special symbols
	HKana	Half-width Katakana characters.
	HRoman	Half-width Roman characters
	Kana	Full-width Kana characters
	Kanji	Full-width ideographic characters
	Proportional	Proportional Roman characters
Adobe-Korea1		
	Alphabetic	Proportional Roman characters
	Dingbats	Special symbols
	Hangul	Full-width ideographic and Hangul characters
Adobe-Japan2		
	Alphabetic	Roman characters
	Dingbats	Special symbols
	HojoKanji	Full-width ideographic characters
Adobe-GB1		
	Alphabetic	Proportional Roman characters
	GBHanzi	Full-width ideographic characters
	Dingbats	Special symbols
	Kana	Japanese Kana characters
Adobe-CNS1		
	Alphabetic	Proportional Roman characters
	CNSHanzi	Full-width ideographic characters
	Dingbats	Special symbols
	Kana	Japanese Kana characters

Following is an example with two entries in the **FD** dictionary.

Example 7.2 *FD entry*

```
/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 Gothic
/AvgWidth 500
/MaxWidth 500
/MissingWidth 500
/StemV 50
/StemH 75
/Ascent 720
/Descent 0
/Leading 83
>>
endobj
```

7.10 Color spaces

A color space specifies how color values should be interpreted. While some PDF operators implicitly specify the color space they use, others require that a color space be specified. As shown in Figure 7.5, PDF 1.2 supports nine color spaces: three device-dependent color spaces, **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**; three device-independent color spaces, **CalGray**, **CalRGB**, and **Lab**; and three special color spaces, **Indexed**, **Pattern**, and **Separation**. The color spaces follow the semantics described in Section 4.8 of the *PostScript Language Reference Manual, Second Edition*.

PDF 1.2

A Color Space is specified by a name if it is one of the device-dependent color spaces (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**). Otherwise it is specified as an array whose first element is the name of one of the device-independent color spaces (**CalGray**, **CalRGB**, or **Lab**) or special color spaces (**Indexed**, **Pattern**, or **Separation**) and whose remaining elements are parameters that complete the definition of the color space.

In a device-dependent color space, the color values are interpreted as specifying the percentage of device colorant to be used. This means that the exact color produced depends on the characteristics of the output device. For example, in the **DeviceRGB** color space, a value of 1 for the red component means "turn red all the way on." If the output device is a monitor, the color displayed depends strongly on the settings of the monitor's brightness, contrast, and color balance adjustments. In addition, the precise color displayed depends on the chemical composition of the compound used as the red phosphor in the particular monitor being used, the length of time the monitor has been turned on, and the age of the monitor.

In a device-independent color space, color values are defined by a mapping from the device-independent color space into a standard color space, the CIE (*Commission Internationale de l'Éclairage*) 1931 XYZ color space. Since the values in the XYZ space can be measured colormetrically, this establishes a device-independent specification of the desired color. When a device-independent color value is rendered on a device, the rendered color is based on the device-independent color specification as well as the color characteristics of the device. This may or may not result in a true colorimetric rendering. Variations from a colorimetric rendering may occur as a consequence of gamut limitations and rendering intents. See the discussion of color rendering intents on page 179.

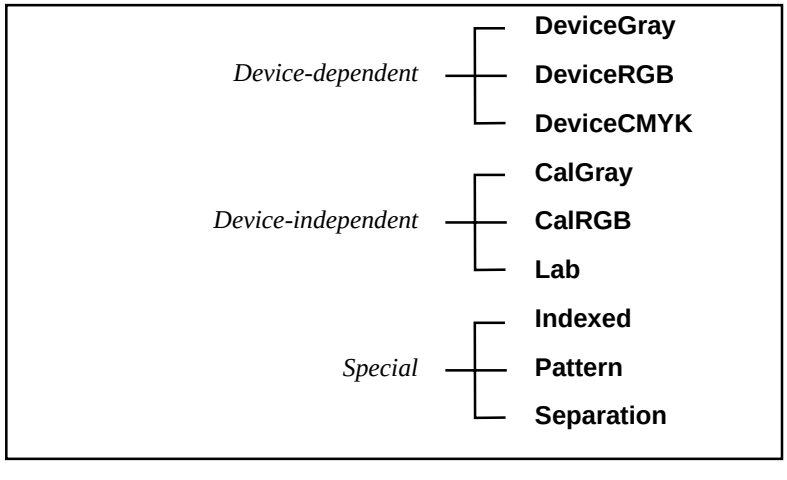
PDF 1.1

See the *PostScript Language Reference Manual, Second Edition* for further explanation of device-independent color.

7.10.1 DeviceGray color spaces

Colors in the **DeviceGray** color space are specified by a single value: the intensity of achromatic light. In this color space, 0 is black, 1 is white, and intermediate values represent shades of gray.

Figure 7.5 *Color spaces*



7.10.2 DeviceRGB color spaces

Colors in the **DeviceRGB** color space are represented by three values: the intensity of the red, green, and blue components in the output. **DeviceRGB** is commonly used for video displays because they are generally based on red, green, and blue phosphors.

7.10.3 DeviceCMYK color spaces

Colors in the **DeviceCMYK** color space are represented by four values. These values are the amounts of the cyan, magenta, yellow, and black components in the output. This color space is commonly used for color printers, where they are the colors of the inks traditionally used for four-color printing. Only cyan, magenta, and yellow are strictly necessary, but black is generally also used in printing because black ink produces a better black than a mixture of cyan, magenta, and yellow inks, and because black ink is less expensive than the other inks.

Note *In PDF 1.1, a color space named was partially defined with the expectation that its definition would be completed in a future version of PDF. That is no longer being considered, although other changes to color spaces are being planned, and they will achieve the same effect that* **CalCMYK** *would have. PDF viewers should ignore* **CalCMYK** *color space attributes and render colors specified in this color space as if they had been specified using* **DeviceCMYK**.

PDF 1.2

7.10.4 CalGray color spaces

PDF 1.1

Colors in a **CalGray** color space are represented by a single value. Input values are in the range 0 to 1, where 0 is black, 1 is white and intermediate values are gray.

A **CalGray** color space is specified by an array of the form

```
[/CalGray dict]
```

where the contents of *dict* are described in Table 7.20.

Table 7.20 *CalGray attributes*

Key	*Type*	*Semantics*
WhitePoint	array	*(Required)* Three numbers $[X_w\ Y_w\ Z_w]$ that specify the CIE 1931 (XYZ)-space tristimulus value of the diffuse white point. The numbers X_w and Z_w must be positive, and Y_w must be equal to 1. See discussion in 4.8.3 in the *PostScript Language Reference Manual, Second Edition* for further details.
BlackPoint	array	*(Optional)* Three numbers $[X_b\ Y_b\ Z_b]$ that specify the CIE 1931 (XYZ)-space tristimulus value of the diffuse black point. The numbers must be non-negative. The default value is `[0 0 0]`. See discussion in 4.8.3 in the *PostScript Language Reference Manual, Second Edition* for further details.

Gamma	number	*(Optional)* Defines the exponential relationship between the gray component and *Y*. The governing equation is $Y = gray^{Gamma}$. *Gamma* must be positive and is generally greater than or equal to 1. The default value is 1.

7.10.5 CalRGB color spaces

PDF 1.1

Colors in a **CalRGB** color space are represented by three values: the red, green and blue components of the color. Each value is in the range 0 to 1.

A **CalRGB** color space is specified by an array of the form:

```
[/CalRGB dict]
```

where the contents of *dict* are described in Table 7.21.

Table 7.21 *CalRGB attributes*

Key	*Type*	*Semantics*
WhitePoint	array	*(Required)* Same as for CalGray.
BlackPoint	array	*(Optional)* Same as for CalGray.
Gamma	array	*(Optional)* Three numbers $[G_r\ G_g\ G_b]$ that specify the gamma for the red, green, and blue components respectively. The governing equations are $R' = R^{G_r}$, $G' = G^{G_g}$, and $B' = B^{G_b}$, where R, G, and B are the input calibrated RGB values, and R′, G′, and B′ are the gamma-modified values. The default value is `[1 1 1]`.
Matrix	array	*(Optional)* Nine numbers $[X_r\ Y_r\ Z_r\ X_g\ Y_g\ Z_g\ X_b\ Y_b\ Z_b]$ that specify the linear interpretation of the gamma-modified red, green, and blue components, R′, G′, and B′. The default value is the identity matrix, `[1 0 0 0 1 0 0 0 1]`. The transformation from R′G′B′ to XYZ is given by: $X = R' \times X_r + G' \times X_g + B' \times X_b$ $Y = R' \times Y_r + G' \times Y_g + B' \times Y_b$ $Z = R' \times Z_r + G' \times Z_g + B' \times Z_b$

An example of a **CalRGB** color space is shown here for D65 white point, 1.8 gammas, and Trinitron phosphor chromaticities.

```
12 0 obj
[/CalRGB
<<
/WhitePoint [0.9505 1 1.0890]
/Gamma [1.8 1.8 1.8]
/Matrix [0.4497 0.2446 0.0252 0.3163 0.6720 0.1412
0.1845 0.0833 0.9227]
>> ]
```

```
endobj
```

7.10.6 Lab color spaces

PDF 1.1

Colors in a **Lab** color space are represented by three values: the L*, a* and b* components of the color. The ranges of each of the three values are specified under the **Range** key in Table 7.22.

A **Lab** color space is specified by an array of the form:

```
[/Lab dict]
```

where the contents of *dict* are described in Table 7.22.

Table 7.22 *Lab attributes*

Key	*Type*	*Semantics*
WhitePoint	array	*(Required)* Same as for **CalGray**.
BlackPoint	array	*(Optional)* Same as for **CalGray**.
Range	array	*(Optional)* Four numbers [a_{min} a_{max} b_{min} b_{max}] specifying the range of the a* and b* components. That is, a* and b* are limited by $a_{min} \le$ a* $\le a_{max}$, $b_{min} \le$ b* $\le b_{max}$. The default value is `[-100 100 -100 100]`. The range of L* is always 0 to 100.

7.10.7 Indexed color spaces

PDF 1.2

Indexed color spaces allow colors to be specified by small integers that are used as indexes into a table of color values. The values in this table are colors specified in some other "base" color space. For example, an indexed color space can have white as color number 1, dark blue as color number 2, turquoise as color number 3, and black as color number 4.

An **Indexed** color space is specified as follows:

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

The base color space is specified by *base*. In PDF 1.0 and PDF 1.1 it must be either **DeviceRGB** or **DeviceCMYK**; in PDF 1.2 it may be any color space except a **Pattern** color space or another **Indexed** color space. The maximum valid index value, specified by *hival*, is determined by the number of colors desired in the **Indexed** color space; it may not be greater than 255. Colors are specified by integers in the range 0 to *hival*. The color table values are contained in *lookup*, which is a stream in PDF 1.0 and 1.1 but may also be a string in PDF 1.2. The stream or string contains $m \times (hival + 1)$ bytes 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 divided by 255, yielding a color component value in the range 0 to 1. (In PDF 1.2, if the base is a **Lab** color space, then the unsigned integer is scaled to

the appropriate range for the component, as described in the previous section.) The color components for each entry in the table are adjacent in the stream. For example, if the base color space is **DeviceRGB** and the indexed color space contains two colors, the order of bytes in the stream is: R_0 G_0 B_0 R_1 G_1 B_1, where letters are the color component and numbers are the table entry.

Example 7.3 shows an indexed color space. Colors in the table are specified in the **DeviceRGB** color space, and the table contains 256 entries. The stream containing the table has been LZW and ASCII base-85 encoded.

Example 7.3 *Indexed color space*

```
12 0 obj
[/Indexed /DeviceRGB 255 13 0 R]
endobj
13 0 obj
<<
/Filter [/ASCII85Decode /LZWDecode]
/Length 554
>>
stream
J3Vsg-=dE=!]*)rE$,8^$P%cp+RI0B1)A)g_;FLE.V9
…omitted data…
bS/5%"OmlTJ=PC!c2]]^rh(A~>
endstream
endobj
```

7.10.8 Pattern color spaces

PDF 1.2

Colors in a **Pattern** color space are represented by Patterns (see page 201). A pattern is either *colored*, in which case the colors it uses are contained in the pattern's contents stream, or it is *uncolored*, in which case its color must be specified whenever the pattern is used, and an underlying color space must be specified with the pattern color space.

A **Pattern** color space for an uncolored pattern is specified as a 2-element array:

```
[/Pattern base]
```

base is the underlying color space, which must not itself be a **Pattern** color space. A **Pattern** color space for a colored pattern is specified as the name **Pattern** or an array containing just the name **Pattern**. It may also be specified by the 2-element array shown above; in this case, *base* is ignored.

7.10.9 Separation color spaces

PDF 1.2

Colors in a **Separation** color space are represented by a single value, called a *tint*, in the range of 0 to 1. The value 0 represents application of the minimum amount of colorant to the separation; 1 represents the maximum amount.

A **Separation** color space is specified as a 4-element array:

[/Separation *name alternativeSpace tintTransform*]

name is the name of the separation or colorant. At the moment when the color space is set to a separation color space, if the device can produce the named separation, then the remaining elements of the array are ignored.

If the device cannot produce the named separation, then subsequent painting operations are performed in the color space specified by *alternativeSpace*. The alternative color space can be a device-dependent color space or a device-independent color space, but not a special color space (**Pattern**, **Indexed**, or **Separation**). The color used in the alternative space is determined by applying *tintTransform* to the tint value. *tintTransform* is a Function (see page 185) whose input is the tint and whose output is a set of color values, one for each of the color components in the alternative color space.

7.10.10 Default color spaces

PDF 1.1

PDF 1.1 adds device-independent color spaces to the color spaces defined in PDF 1.0. Because viewers for PDF 1.0 generally do not expect these new color spaces and default gracefully when they are used, a second method for specifying the use of a device-independent color space is provided in PDF 1.1. This second method allows an appropriate color space to be substituted for either the DeviceGray or DeviceRGB color spaces. The substitution is controlled by two special keys, **DefaultGray** and **DefaultRGB**, that can be used in the ColorSpace dictionary of the current Resources dictionary. They are used as follows.

When a viewer is performing an operation that results in rendering to a medium, there is always a current color space, which is established using the operators of Section 8.5.2, “Color operators,” or using the **ColorSpace** key of an Image XObject or an in-line image. When the current color space is DeviceGray, the ColorSpace dictionary of the current Resources dictionary is checked for the presence of the **DefaultGray** key. If this key is present, then the color space that is the value of that key is used as the color space for the operation currently being performed. The value of the **DefaultGray** key may be either **DeviceGray** or a CalGray color space specification.

Similarly, when the current color space is DeviceRGB, the ColorSpace dictionary of the current Resources dictionary is checked for the presence of the **DefaultRGB** key. If this key is present, then the color space that is the value of that key is used as the color space for the operation currently being performed. The value of the **DefaultRGB** key may be either **DeviceRGB** or a CalRGB color space specification.

7.11 XObjects

XObjects are named resources. PDF currently supports three types of XObjects: images, forms, and pass-through PostScript language fragments. In the future it may support other object types.

XObjects are passed by name to the **Do** operator, described on page 236. The action taken by the **Do** operator depends on the type of XObject passed to it. In the case of images and forms, the **Do** operator draws the XObject.

7.11.1 Images

An Image is an XObject whose **Subtype** is **Image**. Images allow a marking context to specify a sampled image or image mask. PDF supports image masks, 1-bit, 2-bit, 4-bit, and 8-bit grayscale images, and color images with 1, 2, 4, or 8 bits per component. Color images may have one color component (indexed-color values or separation tints), three color components (RGB, CalRGB, or Lab), or four color components (CMYK).

The sample data format and sample interpretation conform to the conventions required by the PostScript language **image** and **imagemask** operators. However, all PDF images have a size of 1×1 unit in user space, and the data must be specified left to right, top to bottom. Like images in the PostScript language, PDF images are sized and positioned by adjusting the current transformation matrix in the marking context.

An Image XObject is specified by a stream object. The stream dictionary must include the standard keys required of all streams as well as additional ones described in the following table. Several of the keys are the same as those required by the PostScript language **image** and **imagemask** operators. Matching keys have the same semantics.

Table 7.23 *Image XObject attributes*

Key	*Type*	*Semantics*
Type	name	(*Required*) Object type. Always **XObject**.
Subtype	name	(*Required*) XObject subtype. Always **Image**.
Name	name	(*Required only for compatibility with PDF 1.0*) Resource name, used as an operand of the **Do** operator. **Name** must match the name used in the XObject dictionary within the page's Resources dictionary.
Width	integer	(*Required*) Width of the source image in samples.
Height	integer	(*Required*) Height of the source image in samples.
BitsPerComponent	integer	(*Required*) The number of bits used to represent each color component.
ColorSpace	name or array	(*Required for images, not allowed for image masks*) Color space used for the image samples. This may be any type of color space except **Pattern**. However, for compatibility with 1.0 viewers, the **DefaultRGB** or **DefaultGray** name should be used to reference a device-independent color space, as described on page 175. The Separation color space is not permitted in PDF 1.0 or PDF 1.1. **PDF 1.2**

Decode	array	(*Optional*) An array of numbers specifying the mapping from sample values in the image to values appropriate for the current color space. The number of elements in the array must be twice the number of color components in the color space specified in the **ColorSpace** key. The default value results in the image sample values being used directly. Decode arrays are described further on page 178.
Interpolate	Boolean	*(Optional)* If *true*, requests that image interpolation be performed. Interpolation attempts to smooth transitions between sample values. Interpolation may be performed differently by different devices, and not at all by some. The default value is *false*.
ImageMask	Boolean	(*Optional*) Specifies whether the image should be treated as a mask. If *true*, the image is treated as a mask: **BitsPerComponent** must be 1, **ColorSpace** should not be provided, and the mask is drawn using the current fill color. If *false*, the image is not treated as a mask. The default value is *false*.
Intent	name	*(Optional)* A name which is a color rendering intent indicating the style of color rendering that should occur. For example, one might want to render images in a perceptual or pleasing manner while rendering line art colors with exact color matches. Intents are meaningful only for the device-independent color spaces. For further details, see page 179. **PDF 1.1**
OPI	dictionary	*(Optional)* An OPI dictionary. See Section 7.11.6 on page 182. **PDF 1.2**

Example 7.4 shows an image object. It is a monochrome image (1 bit per component, DeviceGray) that is 24 samples wide and 23 samples high. Interpolation is not requested and the default decode array is used. The image is given the name `/Im0`, which is used to refer to the image when it is drawn.

Example 7.4 *Image with length specified as an indirect object*

```
5 0 obj
<<
/Type /XObject
/Subtype /Image
/Name /Im0
/Width 24
/Height 23
/BitsPerComponent 1
/ColorSpace /DeviceGray
/Filter /ASCIIHexDecode
/Length 6 0 R
>>
stream
003B00 002700 002480 0E4940 114920 14B220 3CB650
75FE88 17FF8C 175F14 1C07E2 3803C4 703182 F8EDFC
B2BBC2 BB6F84 31BFC2 18EA3C 0E3E00 07FC00 03F800
1E1800 1FF800>
endstream
endobj
```

```
6 0 obj
174
endobj
```

7.11.2 Decode arrays

A Decode array can be used to invert the colors in an image or to compress or expand the range of values specified in the image data. Each pair of numbers in a Decode array specifies the upper and lower values to which the range of sample values in the image is mapped. A Decode array contains one pair of numbers for each component in the color space specified in the image. The mapping for each color component is a linear mapping that, for a Decode array of the form [D_{Min} D_{Max}], can be written as:

$$o = D_{Min} + i \times \frac{D_{Max} - D_{Min}}{2^n - 1}$$

where:

n is the value of **BitsPerComponent**
i is the input value, in the range 0 to $2^n - 1$
D_{Min} and D_{Max} are the values specified in the Decode array
o is the output value, to be interpreted in the color space of the image.

Samples with a value of zero are mapped to D_{Min}, samples with a value of 2^n - 1 are mapped to D_{Max}, and samples with intermediate values are mapped linearly between D_{Min} and D_{Max}. The default Decode array for each color component is `[0 1]`, causing sample values in the range 0 to 2^n - 1 to be mapped to color values in the range 0 to 1. Table 7.24 shows the default Decode arrays for various color spaces.

Table 7.24 *Default Decode arrays for various color spaces*

Color space	*Default Decode array*
DeviceGray	`[0 1]`
DeviceRGB	`[0 1 0 1 0 1]`
DeviceCMYK	`[0 1 0 1 0 1 0 1]`
CalGray	`[0 1]`
CalRGB	`[0 1 0 1 0 1]`
Lab	`[0 100` a_{Min} a_{Max} b_{Min} b_{Max}`]` where a_{Min}, a_{Max}, b_{Min}, and b_{Max} correspond to the entries in the **Range** array of the image's color space. 0 and 100 are the first two entries since the range of L* is always 0 to 100.
Pattern	(cannot be used with images)
Indexed	`[0` *N*`]` where $N = 2^n - 1$
Separation	`[0 1]`

As an example of a Decode array, consider a DeviceGray image with 8 bits per component. The color of each sample in a DeviceGray image is represented by a single number. The default Decode array maps a sample value of 0 to a color value of 0 and a sample value of 255 to a color value of 1. A negative image is produced by specifying a Decode array of `[1 0]`, which maps a sample value of 0 to a color value of 1 and a sample value of 255 to a color value of 0. If the image contains only values from 0 to 63 and is to be displayed using the full gray range of 0 to 1, a Decode array of `[0 4]` should be used. With this Decode array, a sample value of 0 maps to a color value of 0, a sample value of 255 maps to a color value of 4, and a sample value of 63 (the maximum value in the example) maps to a color value of 0.99.

7.11.3 Color rendering intent

PDF 1.1

The supported color rendering intents and their meanings are given below in Table 7.25. Other intents are permitted, but a viewer based on the PDF 1.1 specification will most likely ignore other values. The default intent is **RelativeColorimetric**.

Table 7.25 *Color rendering intents*

Name	*Semantics*
AbsoluteColorimetric	Requests an exact color (hue, saturation, and brightness) match. This is appropriate for uses such as some line art or spot colors. If the exact color cannot be displayed, the closest available one is substituted.
RelativeColorimetric	Requests an exact hue/saturation match, but scales the brightness range so that all brightnesses fit into the display device's brightness range. This is often appropriate for line art and spot color. As a result of the brightness scaling, the exact colors produced will differ on devices having different brightness range capabilities. If the exact hue/saturation cannot be displayed, the closest available one is substituted.
Perceptual	Scales the hue, saturations, and brightness ranges so that all values can be displayed on the output device. This generally provides a pleasing rendering of scanned images. As a result of the scaling, all colors are modified somewhat.
Saturation	Emphasizes saturation. This is appropriate for business graphics.

7.11.4 Form XObjects

A *form* is a self-contained description of any text, graphics, or sampled images that may be drawn multiple times on several pages or at different locations on a single page.

Note *In PDF 1.2, the term "form" also refers to a completely separate object, a database described on page 118 and stored in the file's Catalog under the name* **AcroForm**. *There is only of those per document. The form described here is a subtype of XObject, corresponding to forms in the PostScript language. There can be any number of these forms in a document. These forms are referred to as "Form XObjects."*

A Form XObject is specified by a PDF stream. It is a marking context (see page 138), The keys in the stream dictionary correspond to the keys in a PostScript language Form dictionary. Unlike a PostScript language Form dictionary, the Form dictionary does not contain a **PaintProc** key. Instead, the stream contents specify the painting procedure. As usual, the stream must also include a **Length** key and may include **Filter** and **DecodeParms** keys if the stream is encoded. Table 7.26 describes the attributes of a Form XObject.

To draw a form, the **Do** operator is used, with the name of the form to be drawn given as an operand. As discussed in the introduction on page 133, this name is mapped to an object using the current Resources dictionary.

Table 7.26 *Form XObject attributes*

Key	*Type*	*Semantics*
Type	name	(*Required*) Object type. Always **XObject**.
Subtype	name	(*Required*) XObject subtype. Always **Form**.
BBox	Rectangle	(*Required*) The form's bounding box, in the form coordinate system. This bounding box is used to clip the output of the form and to determine its size for caching.
FormType	integer	(*Required*) Must be 1.
Matrix	matrix	(*Required*) A transformation matrix that maps from the form's coordinate space into user space.
Name	name	(*Required only for compatibility with PDF 1.0*) Resource name, used as an operand of the **Do** operator. **Name** must match the name used in the XObject dictionary within the Resources dictionary of the marking context where the form appears.
Resources	dictionary	(*Optional but strongly recommended*) A list of the resources such as fonts and images required by this form. **PDF 1.2** In PDF 1.0 and 1.1, all named resources used in the form must be included in the Resources dictionary of each Page object on which the form appears, regardless of whether or not they also appear in the Resources dictionary of the form. It can be useful to specify them in the form's own Resources dictionary as well, in order to determine which resources are used inside the form. If a resource is included in both dictionaries, it should have the same name in both locations. In PDF 1.2, forms can be independent of the marking contexts in which they appear, and this is strongly recommended although not required. In an independent form, the Resources dictionary of the form is required and contains all the named resources used by the form. Those resources are not "promoted" to the outer context's Resources dictionary, although that context's Resources dictionary will refer to the form itself.

XUID	array	(*Optional*) An ID that uniquely identifies the form. This allows the form to be cached after the first time it has been drawn in order to improve the speed of subsequent redraws. XUID arrays may contain any number of elements. The first element in an XUID array is the organization ID. Forms that are used only in closed environments may use 1000000 as the organization ID. Any value can be used for subsequent elements, but the same values must not be used for different forms. Organizations that plan to distribute forms widely and wish to use XUIDs must obtain an organization ID from Adobe Systems Incorporated, as described in Appendix E. Section 5.8 of the *PostScript Language Reference Manual, Second Edition* provides a further explanation of XUIDs.
OPI	dictionary	*(Optional)* An OPI dictionary. See Section 7.11.6 on page 182.

Example 7.1 *Form XObject*

```
6 0 obj
<<
/Type /XObject
/Subtype /Form
/Name /Fm0
/FormType 1
/BBox [0 0 1000 1000]
/Matrix [1 0 0 1 0 0]
/Length 38
>>
stream
0 0 m 0 1000 l 1000 1000 l 1000 0 l f
endstream
endobj
```

7.11.5 PostScript XObjects

PDF 1.1

PDF enables a document to include PostScript language fragments in a page description. These fragments are printer-dependent and take effect only when printing on a PostScript printer. They have no effect either when viewing the file or when printing to a non-PostScript printer. In addition, applications that understand PDF are unlikely to be able to interpret the PostScript language fragments. Hence, this capability should be used only if there is no other way to achieve the same result.

A PostScript XObject is an XObject whose **Subtype** key has the value **PS**. When a document is printed to a PostScript printer, the contents of the XObject's stream replace the **Do** command that references the XObject. This stream is copied without interpretation and may include PostScript comments. In any other case, the XObject is ignored. When printing to a PostScript Level 1 printer, if the XObject contains a **Level1** key, the value of that key, which must be a stream, is used instead of the contents of the PostScript XObject's stream.

The PostScript fragment may use Type 1 and TrueType fonts listed in the current Resources dictionary. It may not use Type 3 fonts.

Note *PostScript XObjects should be used with extreme caution and only to obtain results not otherwise possible in PDF. Inappropriate use of PostScript XObjects can cause PDF files to print incorrectly.*

Table 7.27 *PostScript XObject attributes*

Key	*Type*	*Semantics*
Type	name	(*Required*) Object type. Always **XObject**.
Subtype	name	(*Required*) XObject subtype. Always **PS**.
Level1	stream	*(Optional)* A contents stream to be used in place of the XObject's stream when printing to a Level 1 PostScript printer.

The PostScript XObject is not compatible with 1.0 viewers. The following method can be used instead to create PostScript pass-through data when compatibility with 1.0 viewers is necessary. A form should be defined with an empty stream content. It should include a **BBox** of all zeros, a **FormType** of 1, and a **Matrix** that is the identity matrix. It should include a **Subtype2** key whose value is **PS**, and a **PS** key whose value is a stream that contains the PostScript language pass-through data. It may also contain a **Level1** key as described above.

7.11.6 OPI dictionary

PDF 1.2

In PDF 1.2, Image and Form XObjects can contain an **OPI** key. OPI stands for *Open Prepress Interface*. It is a mechanism for representing a placeholder for an image, typically a high-resolution image. The placeholder often includes a low-resolution image or *proxy*, which might be a downsampled version of the high-resolution image. Before a document containing OPI references is printed, it typically passes through a filter known as an OPI server, which replaces the proxies with the high-resolution images.

The workflow in a prepress environment often involves several applications in areas such as graphic design, photo manipulation, word processing, page layout, and document construction. As pieces of the final document are moved from one application to another, it is useful to maintain an "indirect reference" or external pointer to the data of high-resolution images; they can be quite large, many times the size of the rest of the document.

The Open Prepress Interface, originally developed by Aldus Corporation, is represented in PostScript language programs by a set of comments surrounding the PostScript code for the proxy. In PDF, the proxy may be represented as an Image XObject if the proxy is, in fact, a single image; otherwise the proxy, which can be any set of graphic objects such as a gray rectangle with text or nothing at all, is represented as a Form XObject. In either case, the information that corresponds to the OPI comments in a PostScript file is stored in the XObject's dictionary under the name **OPI**.

Table 7.28 *OPI dictionary*

Key	*Type*	*Semantics*
version	dictionary	There are two standard versions of OPI, the older version 1.3 and the newer version 2.0, which is replacing 1.3. While they serve the same purpose, they represent information in different ways; an OPI dictionary contains one of the keys **1.3** or **2.0**. The value of each key is a version-specific dictionary containing the OPI information, described below.

Note *The keys in an OPI dictionary are name objects, as all keys are in PDF dictionaries. An OPI dictionary would be written thus:*

```
<<
/1.3 20 0 R                % OPI 1.3 dictionary
>>
<<
/2.0 27 0 R                % OPI 2.0 dictionary
>>
```

The following two tables describe the version-specific OPI dictionaries. For details on the meaning of these keys and the effect they have on OPI servers, refer to the *Open Prepress Interface Specifications* for both versions 1.3 and 2.0. These are available on Adobe's Web site.

Table 7.29 *OPI 1.3 dictionary*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **OPI**.
Version	real	*(Required)* The version of OPI used in this dictionary. Must be the number 1.3 (not the name **1.3**, as in the OPI dictionary).
F (File) File specification		*(Required)* An external file containing image data.
CropRect	array	*(Optional)* An array of integers [*left top right bottom*] specifying the portion of the external image that is to be included.
Color	array	*(Optional)* An array of 4 numbers and a string, [*C M Y K colorname*]. Default is `[0 0 0 1 (Black)]`.
ColorType	name	*(Optional)* Must be one of **Spot**, **Process**, or **Separation**. Default is **Spot**.
Comments	string	*(Optional)* Documentation.
CropFixed	array	*(Optional)* An array of real numbers [*left top right bottom*] with the same semantics as **CropRect**.

GrayMap	array	*(Optional)* An array of 2^n 16-bit numbers, where *n* is the number of bits per sample (1, 4, or 8), recording changes made to the brightness or contrast of the image.
ID	string	*(Optional)* An identifier for TIFF images in the external file.
ImageType	array	*(Optional)* An array of 2 integers, [*samplesPerPixel bitsPerSample*].
Overprint	Boolean	*(Optional)* Default is *false*.
Position	array	*(Optional)* An array [ll_x ll_y ul_x ul_y ur_x ur_y lr_x lr_y] of user-space coordinates, for the placement of the image on the page.
Resolution	array	*(Optional)* An array [*horizRes vertRes*] specifying the resolution of the image, in samples per inch.
Tags	array	*(Optional)* An array of pairs [*tagNumber tagName* ...] describing the TIFF ASCII tag values in the external file. Each *tagNumber* is an integer; each *tagName* is a string.
Tint	real	*(Optional)* Default is 1.0.
Transparency	Boolean	*(Optional)* Default is *true*.

Table 7.30 *OPI 2.0 dictionary*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **OPI**.
Version	number	*(Required)* The version of OPI used in this dictionary. Must be the number 2 or 2.0 (not the name **2.0**, as in the OPI dictionary).
F (File)	File specification	*(Required)* An external file containing image data.
CropRect	array	*(Optional)* An array of real numbers [*left top right bottom*] specifying the portion of the external image that is to be included.
IncludedImageDimensions	array	*(Optional)* An array [*width height*].
IncludedImageQuality	real	*(Optional)*

Inks	name or array	*(Optional)* Legal values include:
		`/full_color`
		`/registration`
		`[/monochrome` *name tint name tint* ...`]`
		where each *name* is a string and each *tint* is a real number.

MainImage	string	*(Optional)*
Overprint	Boolean	*(Optional)* Default is *false*.
Tags	array	*(Optional)* An array of pairs [*tagNumber tagName …*], describing the TIFF ASCII tag values in the external file. Each *tagNumber* is an integer; each *tagName* is be either a string or an array of strings.

7.12 Functions

PDF 1.2

PDF does not provide a direct representation for functions, as procedures in the PostScript language do. Instead, PDF provides several types of *Function dictionaries* that represent parameterized classes of functions, including mathematical formulas, and sampled representations with arbitrary resolution.

Each class of Function dictionary has a *function type* that specifies the representation of the function, the set of attributes that parameterize that representation, and the additional data needed by that representation.

Functions may be thought of as "*m*-in, *n*-out" numerical transformations. Each Function dictionary implicitly declares the values of *m* and *n*. It explicitly declares a domain of input values for which the function is defined, and a range outside which no result value will fall. Domain and range intervals are bounded; they must be rectangular in the input or output space of the function, and they are assumed to be closed in the mathematical sense, i.e., the edges of the interval are included in the interval. The function must be defined (but need not be continuous or smooth) across its entire domain. If any input in the declared domain of the function would cause the function to output a value outside the declared range, then that output value is clipped to the declared range. Functions should never be called with input values outside the declared domain. If they are, the result is not defined; an error may be raised, or a valid or meaningless result may be returned.

Each client of a Function must specify how it uses the function, and how it maps the client's domain into the function's domain. If the output of the function is modified by the client before use, that must also be specified by each client. Clients of functions should note that the function's declared domain may be a subset of the actual domain of the function, and the declared range may be a superset of the actual range of the function. Because of this, it is usually necessary to selectively specify the function so that its domain and range are appropriate for the client's use.

All Function dictionaries share the following attributes.

Table 7.31 *Function dictionary attributes shared by all functions*

Key	*Type*	*Semantics*
FunctionType	integer	*(Required)* Must be one of the defined function type values, as described in the following sections.

Domain	array	*(Required)* An array of numbers, interpreted in pairs. Each pair of numbers declares the domain of one input value. The smaller bound must precede the larger bound in each pair. If an input value is outside the bounds, the nearest value inside the bounds is used. The size of the **Domain** array implicitly declares the input dimensionality *m* of an *m*-in, *n*-out function, as *m* is one-half the number of elements in **Domain**.
Range	array	*(Optional for most function types)* An array of numbers, interpreted in pairs. Each pair of numbers declares the range of one output value. The smaller bound must precede the larger bound in each pair. If the function produces a value outside the bounds, the nearest value inside the bounds is used. The size of the **Range** array implicitly declares the output dimensionality *n* of an *m*-in, *n*-out function, as *n* is one-half the number of elements in **Range**.

In addition, each type of Function dictionary must include attributes appropriate to the Function type. The output dimensionality can usually be deduced from other attributes of the function; if not, the **Range** attribute is required. The dimensionality as inferred from the **Domain** and **Range** declarations must be consistent with the dimensionality inferred from other attributes of the function.

Note *The only type of function defined in PDF 1.2 is Type 0. Other types will be defined in the future.*

7.12.1 Sampled functions (Function Type 0)

Type 0 functions use a sequence of sample values to provide an approximation for functions whose domains and ranges are bounded. The samples are organized in a table, or array. The dimensionality of the sample table is equal to the dimensionality of the input domain. Samples may have more than one component; the number of components in each sample is equal to the dimensionality of the output range.

The sample values determine the range of the function. The sample points are interpolated linearly, unless the optional higher-order spline interpolation is specified by the **Order** attribute.

Sampled functions are highly general, and offer reasonably accurate representations of arbitrary analytic functions at low expense. For instance, a single-input sinusoidal function can be represented over the range [0 180] with an average error of only 1%, using just 10 samples and linear interpolation. 2-input functions require significantly more samples, but usually not a prohibitive number as long as the function does not have high-frequency variations.

The dimensionality of a sampled function is restricted only by implementation limits. However, the number of samples required to represent high-dimensionality functions multiplies very rapidly unless the sampling resolution is very low, and the process of multi-linear interpolation becomes very intensive computationally if the input dimensionality is greater than two. (The multi-dimensional spline interpolation is even more intensive computationally.)

A Function dictionary of Type 0 includes the following attributes:

Table 7.32 *Attributes of sampled functions (FunctionType 0)*

Key	*Type*	*Semantics*
FunctionType	integer	*(Required)* Specifies a sampled function. Always 0.
Domain	array	*(Required)* As in Table 7.31.
Range	array	*(Required)* As in Table 7.31.
Size	array	*(Required)* The number of samples in each input dimension of the sample table.
BitsPerSample	integer	*(Required)* Specifies the number of bits used to represent each sample value. Limited to 1, 2, 4, 8, 12, 16, 24, or 32.
Order	integer	*(Optional; default is 1)* Specifies the order of interpolation between samples. Allowed values are 1 or 3, specifying linear or cubic-spline interpolation, respectively.
Encode	array	*(Optional)* Specifies the linear mapping of input values into the domain of the function's sample table (see below). Default value: `[0 (`*Size*$_0$`-1) 0 (`*Size*$_1$`-1)...]`.
Decode	array	*(Optional)* Specifies the linear mapping of sample values into the range of values appropriate for the function's output variables. Default value: same as **Range**.
other stream attributes		*(Optional)* The sample values that specify the function are provided in a stream. The stream's attributes are included in the Function dictionary, as appropriate. See below for details.

The **Domain**, **Encode**, and **Size** attributes determine how the function's input values are mapped into the sample table. For example, if the **Domain** is `[-1 1 -1 1]`, and the **Size** is `[21 31]`, the default for **Encode** would be `[0 20 0 30]`, which maps the entire **Domain** into the full set of sample table entries. Other values of **Encode** may be used. In general, the i^{th} input variable x_i is clipped to the interval *Domain*$_{2i}$ to *Domain*$_{2i+1}$. From this value, the corresponding encoded value e_i is computed:

$$e_i = (x_i - \text{Domain}_{2i}) \times \frac{\text{Encode}_{2i+1} - \text{Encode}_{2i}}{\text{Domain}_{2i+1} - \text{Domain}_{2i}} + \text{Encode}_{2i}$$

e_i is then clipped to the interval 0 to *Size*$_i$ - 1. The encoded input values are real numbers, not restricted to integers, and bilinear interpolation is used to determine an output value from the four nearest-match sample table values.

Similarly, the **Range**, **Decode**, and **BitsPerSample** attributes determine how the function's sample values are mapped into output values. The value of **BitsPerSample** implies that all sample values are in the range 0 to ($2^{\text{BitsPerSample}}$ -

1). This range is linearly transformed by the **Decode** array to an output range. The default **Decode** array is equal to the **Range** array, indicating a mapping of the entire possible sample range into the entire possible output range. Other values of **Decode** may be used. In general, for the j^{th} sample component s_j, the corresponding output value y_i is

$$y_j = s_j \times \frac{\text{Decode}_{2j+1} - \text{Decode}_{2j}}{2^{\text{BitsPerSample}} - 1} + \text{Decode}_{2j}$$

Finally, y_j is clipped to the interval $Range_{2j}$ to $Range_{2j+1}$.

Samples are encoded and interpreted exactly as image samples are, except for the requirement that each row of image data begin on a byte boundary. No row padding is done with sampled function data. Like image data, a sequence of samples is considered to represent an array in which the first dimension of the array varies fastest. For example, in a two-dimensional array of data, the *x* component varies fastest, and the *y* component next fastest.

As an example, consider a sampled function with 4-bit samples in an array containing 21 columns and 31 rows, and consider using this function to represent a halftone spot function. A spot function takes two arguments, *x* and *y*, in the domain [-1 1], and returns one value, *z*, in that same range. The Function dictionary is shown in the following example:

Example 7.2 *Example of a spot function*

```
14 0 obj
<<
/FunctionType 0
/Domain [-1 1 -1 1]
/Range [-1 1]
/Size [21 31]
/BitsPerSample 4
/Encode [0 20 0 30]
/Decode [-1 1]
/Length …
/Filter …
>>
stream
… 651 sample values …
endstream
endobj
```

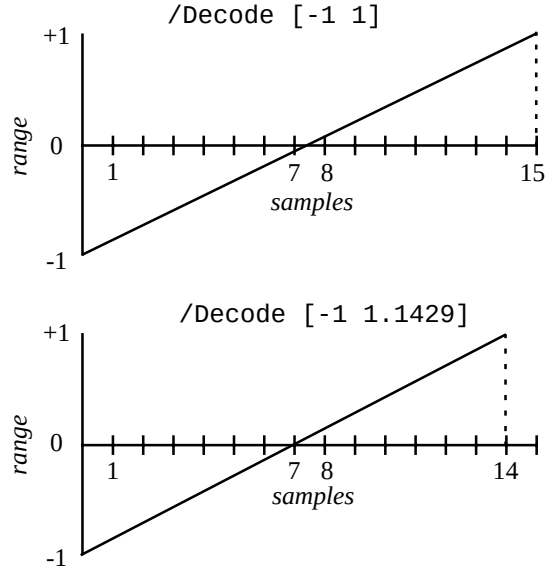
The *x* argument would be linearly transformed by the encoding to the domain [0 20], and the *y* argument to the domain [0 30]. Using bilinear interpolation between sample points, the function computes a value for *z*, which will be in the range [0 15], and the decoding transforms *z* to a number in the range [-1 1] for the result. The 651 sample values (31 × 21) are stored in a stream of 326 bytes = ⌈31 rows ×

21 samples/row × 4 bits/sample ÷ 8 bits/byte⌉. The first byte contains the sample for the point (-1, -1) in the high-order 4 bits, and the sample for the point (-0.9, -1) in the low-order 4 bits.

Encode gives the linear mapping between **Domain** and **Size**. The default value for **Encode** is `[0 (`*Size*$_0$`-1) 0 (`*Size*$_1$`-1) ...]`.

Decode may be used creatively to increase the accuracy of encoded samples corresponding to certain values in the range. For example, if the desired range of the function is [-1 1] and **BitsPerSample** is 4, the default value for **Decode** would be the same as the **Range**, [-1 1], and the sample values would be integers in the interval [0 15]. But if these values were used, the midpoint of the range of the function, 0, would not be represented by any sample value, since it would fall halfway between 7 and 8. Instead, one could use a **Decode** array of `[-1 1.1429]` (1.1429 ≅ 16/14) and sample values in the interval [0 14]. In this way, the desired effective range of [-1 1] would be achieved, and the range value 0 would be represented by the sample value 7. See Figure 7.6.

Figure 7.6 *Mapping with the Decode array*



7.13 Extended graphics states

PDF 1.2

A Resources dictionary may include an **ExtGState** key (“extended graphics state”), whose value is a dictionary that specifies a set of parameters in the graphics state, as shown in Table 7.33. Other parameters may be added in the future. Note that most of these parameters have device-dependent effects. They should not be used in a page description that is intended to be device-independent.

To set these parameters in the graphics state, use the **gs** operator; see Section 8.4.7, "Generic Graphics State operator."

Table 7.33 *ExtGState attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **ExtGState**.
SA	Boolean	*(Optional)* Stroke adjustment.
OP	Boolean	*(Optional)* Overprint.
BG	function	*(Optional)* Black-generation function, which maps the interval [0 1] to the interval [0 1].
UCR	function	*(Optional)* Undercolor removal function, which maps the interval [0 1] to [-1 1].
TR	function or array	*(Optional)* Transfer function, which maps the interval [0 1] to [0 1]. The value is either a single function (corresponding to the PostScript **settransfer** operator) or an array of four functions (corresponding to the PostScript **setcolortransfer** operator). The name **Identity** may be used to represent the identity function. (See Example 7.4 on page 198.)
HT	halftone or **Default**	*(Optional)* The halftone parameter. Specifying the name **Default** has the effect of resetting the halftone parameter to its default value.
HTP	array	*(Optional)* Halftone phase, specified as an array of two integers. *(Used by viewers that are based on Display PostScript)*

Example 7.3 shows two ExtGStates. In the first one, stroke adjustment is on, and it includes a transfer function that inverts its value, $f(x) = 1 - x$. In the second one, overprint is turned off, and it includes a parabolic transfer function, $f(x) = (2x - 1)^2$, with a sample of 21 values. The domain of the transfer function, [0 1], is mapped to [0 20], and the range of the sample values, [0 255], is mapped to the range of the transfer function, [0 1].

Example 7.3 *ExtGStates*

```
2 0 obj
<<
/Type /Page
/Parent 6 0 R
/Resources 33 0 R
/Contents 3 0 R
>>
endobj

33 0 obj
<<
```

```
/ProcSet [/PDF /Text]
/Font << /F1 5 0 R >>
/ExtGState << /GS1 10 0 R /GS2 12 0 R>>
>>
endobj

10 0 obj
<< /Type /ExtGState /SA true /TR 11 0 R >>
endobj

11 0 obj
<< /Size 2 /Length 7 /Filter /ASCIIHexDecode >>
stream
01 00>
endstream
endobj

12 0 obj
<< /Type /ExtGState /OP false /TR 14 0 R >>
endobj

14 0 obj
<< /Size 21 /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
```

7.14 Halftones

PDF 1.2

PDF supports halftones of type 1, 5, 6, and 10, corresponding to those same types in the PostScript language. A halftone is represented by a dictionary; it contains the same key-value pairs as a PostScript language halftone dictionary, with the following exceptions:

- The **Type** key is required.
- Spot functions and transfer functions are represented by Function objects.
- Threshold arrays are specified as streams.
- In Type 5 halftone dictionaries, the keys for colorants must be names; they may not be strings.

7.14.1 Type 1 halftones

Table 7.34 *Entries in a Type 1 halftone dictionary*

Key	*Type*	*Semantics*
Type	name	*(Required)* Object type. Always **Halftone**.
HalftoneType	number	*(Required)* Must be 1.
HalftoneName	string	*(Optional)* If present, this supplies the name of the halftone dictionary.
Frequency	number	*(Required)*
Angle	number	*(Required)*
AccurateScreens	Boolean	*(Optional; the default is false,)*
SpotFunction	function or name	*(Required)* The spot function, which maps points in "spot space" (the square whose corners are at ±1 in *x* and *y*) to values in the interval [-1 1]. **SpotFunction** is represented as a Function with a 2-dimensional array of sample values; or it may be the name of a predefined spot function (see below).
TransferFunction	function	*(Optional)* The transfer function, which maps the interval [0 1] to the interval [0 1]. If present, this overrides any transfer function specified by the **TR** key in the ExtGState dictionary. This is required in a Type 1 halftone dictionary that is used as an element of a Type 5 halftone dictionary for a non-primary color component.

7.14.2 Predefined spot functions

Table 7.35 shows the predefined spot-function names. The description shows the function corresponding to the name, as well as the PostScript code. The image on the left shows the relative values of the function in spot space, indicating approximately the order in which pixels are whitened in the halftone cell, darker points corresponding to pixels that are whitened later than lighter points.

Table 7.35 *Predefined spot functions*

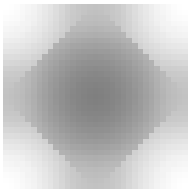
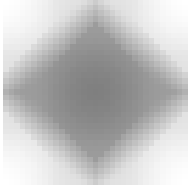
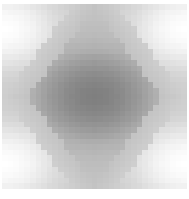
Name	*Definition*
Round 	**{ abs exch abs 2 copy add 1 le** **{ dup mul exch dup mul add 1 exch sub }** **{ 1 sub dup mul exch 1 sub dup mul add 1 sub }** **ifelse }** *if* $\|x\| + \|y\| \le 1$ *then* $1 - (x^2 + y^2)$ *else* $(\|x\| - 1)^2 + (\|y\| - 1)^2 - 1$
Diamond 	**{ abs exch abs 2 copy add .75 le** **{ dup mul exch dup mul add 1 exch sub }** **{ 2 copy add 1.23 le** **{ .85 mul add 1 exch sub }** **{ 1 sub dup mul exch 1 sub dup mul add 1 sub }** **ifelse }** **ifelse }** *if* $\|x\| + \|y\| \le 0.75$ *then* $1 - (x^2 + y^2)$ *else if* $\|x\| + \|y\| \le 1.23$ *then* $1 - (0.85\|x\| + \|y\|)$ *else* $(\|x\| - 1)^2 + (\|y\| - 1)^2 - 1$
Ellipse 	**{ abs exch abs 2 copy 3 mul exch 4 mul add 3 sub dup 0 lt** **{ pop dup mul exch .75 div dup mul add 4 div 1 exch sub }** **{ dup 1 gt** **{ pop 1 exch sub dup mul exch 1 exch sub** **.75 div dup mul add 4 div 1 sub }** **{ .5 exch sub exch pop exch pop }** **ifelse }** **ifelse }** *let* $w = 4\|x\| + 3\|y\| - 3$ *if* $w < 0$ *then* $1 - \dfrac{x^2 + \left(\dfrac{\|y\|}{0.75}\right)^2}{4}$ *else if* $w > 1$ *then* $\dfrac{(1 - \|x\|)^2 + \left(1 - \dfrac{\|y\|}{0.75}\right)^2}{4} - 1$ *else* $0.5 - w$

Table 7.35 *Predefined spot functions*

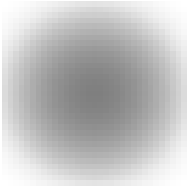
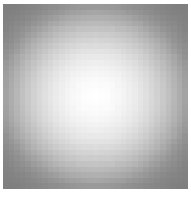
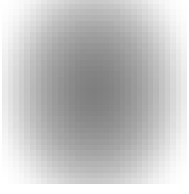
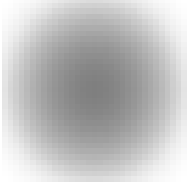
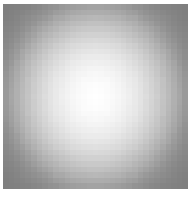
Name	*Definition*
EllipseA 	**{ dup mul .9 mul exch dup mul add 1 exch sub }** $1 - (x^2 + 0.9y^2)$
InvertedEllipseA 	**{ dup mul .9 mul exch dup mul add 1 sub }** $x^2 + 0.9y^2 - 1$
EllipseB 	**{ dup 5 mul 8 div mul exch dup mul exch add sqrt 1 exch sub }** $1 - \sqrt{x^2 + \frac{5}{8}y^2}$
EllipseC 	**{ dup mul .9 mul exch dup mul add 1 exch sub }** $0.9x^2 + y^2 - 1$
InvertedEllipseC 	**{ dup mul .9 mul exch dup mul add 1 sub }** $1 - (0.9x^2 + y^2)$

Table 7.35 *Predefined spot functions*

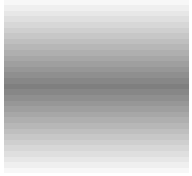
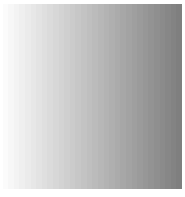
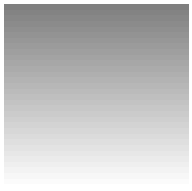
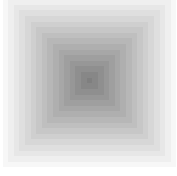
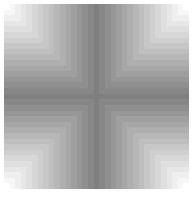
Name	*Definition*
Line 	**{ exch pop abs neg }** $-\lvert y \rvert$
LineX 	**{ pop }** x
LineY 	**{ exch pop }** y
Square 	**{ abs exch abs 2 copy lt { exch } if pop neg }** $-\max(\lvert x \rvert, \lvert y \rvert)$
Cross 	**{ abs exch abs 2 copy gt { exch } if pop neg }** $-\min(\lvert x \rvert, \lvert y \rvert)$

Table 7.35 *Predefined spot functions*

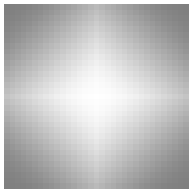
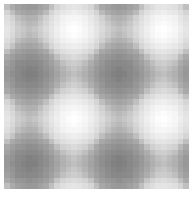
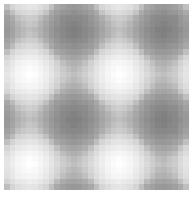
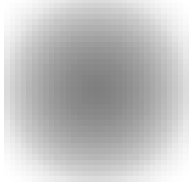
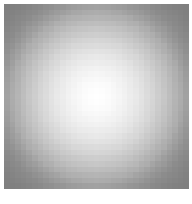
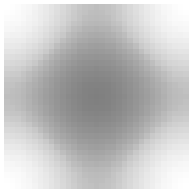
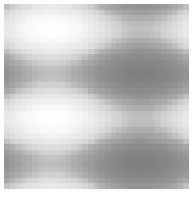
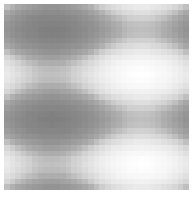
Name	*Definition*
Rhomboid 	**{ abs exch abs 0.9 mul add 2 div }** $\frac{0.9\lvert x\rvert + \lvert y\rvert}{2}$
DoubleDot 	**{ 2 {360 mul sin 2 div exch } repeat add }** $\frac{\sin(360x)}{2} + \frac{\sin(360y)}{2}$
InvertedDoubleDot 	**{ 2 {360 mul sin 2 div exch } repeat add neg }** $-\left(\frac{\sin(360x)}{2} + \frac{\sin(360y)}{2}\right)$
SimpleDot 	**{ dup mul exch dup mul add 1 exch sub }** $1 - (x^2 + y^2)$
InvertedSimpleDot 	**{ dup mul exch dup mul add 1 sub }** $x^2 + y^2 - 1$

Table 7.35 *Predefined spot functions*

Name	*Definition*
CosineDot 	**{ 180 mul cos exch 180 mul cos add 2 div }** $\frac{\cos(180x) + \cos(180y)}{2}$
Double 	**{ exch 2 div exch 2 { 360 mul sin 2 div exch } repeat add }** $\frac{\sin\left(360\frac{x}{2}\right)}{2} + \frac{\sin(360y)}{2}$
InvertedDouble 	**{ exch 2 div exch 2 { 360 mul sin 2 div exch } repeat add neg }** $-\left(\frac{\sin\left(360\frac{x}{2}\right)}{2} + \frac{\sin(360y)}{2}\right)$

Example 7.4 shows an ExtGState that includes a Type 1 halftone.

Example 7.4 *Halftone with spot-function dictionary*

```
28 0 obj
<<
/Type /Halftone
/HalftoneType 1
/Frequency 120
/Angle 30
/SpotFunction /CosineDot
/TransferFunction /Identity
>>
endobj
```

7.14.3 Type 5 halftones

Type 5 halftones allows specification of individual halftones for an arbitrary number of color components.

Table 7.36 *Entries in a Type 5 halftone dictionary*

Key	*Type*	*Semantics*
Type	name	*(Required)* Must be **Halftone**.
HalftoneType	number	*(Required)* Must be 5.
HalftoneName	string	*(Optional)* If present, this supplies the name of the halftone dictionary.
separation	halftone	*(Required, one per separation)* The separation key is the name of a separation or colorant, and the value is a halftone for that separation. The halftone must be of type 1, 6, or 10. Note that the key must be a name object; strings are not permitted here, although they are in PostScript type 5 halftone dictionaries.
Default	halftone	*(Required)* The halftone that is to be used for any separation that does not have its own entry.

Example 7.5 shows an ExtGState dictionary that includes a type-5 halftone dictionary with the primary colorants for a CMYK device. In this example, the halftone dictionaries for the colorants and for the default all use the same spot function.

Example 7.5 *Halftone dictionary for type 5*

```
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
/Angle 45
/SpotFunction /Round
/AccurateScreens true
>>
endobj
```

7.14.4 Type 6 halftones

A Type 6 halftone defines a halftone screen directly by specifying a threshold array at device resolution. The halftone is represented as a stream; the threshold values are stored in the stream, which must contain *width* × *height* bytes. For more details, see the *PostScript Language Reference Manual Supplement for Version 2016* [2]. The attributes are stored in the stream's dictionary.

Table 7.37 *Type 6 halftone attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Must be **Halftone**.
HalftoneType	number	*(Required)* Must be 6.
HalftoneName	string	*(Optional)* If present, this specifies the name of the halftone dictionary.
Width	integer	*(Required)* Width of the threshold array, in pixels.
Height	integer	*(Required)* Height of the threshold array, in pixels.
TransferFunction	function	*(Optional)* Transfer function, which maps the interval [0 1] to the interval [0 1]. If present, this overrides the transfer function specified by the **TR** key in the ExtGState dictionary specified by the **gs** operator. This is required in a Type 6 halftone dictionary that is used as an element of a type 5 halftone dictionary for a non-primary color component.

7.14.5 Type 10 halftones

A Type 10 halftone can be used to specify a threshold array that represents a halftone cell with a non-zero screen angle. A Type 6 halftone can be used to specify a threshold array representing a zero-angle halftone cell, but there is no provision for other angles. Zero-angle halftone cells are easy to specify because they line up nicely with scan lines and because it is not difficult to determine where

a sampled point goes. The Type 10 halftone applies a simple transformation to the halftone cell that converts it into two squares, thus making it easier to specify non-zero angle cells.

The halftone is represented as a stream; the threshold values are stored in the stream. There must be $Xsquare^2 + Ysquare^2$ bytes in the threshold array. For more details, see the *PostScript Language Reference Manual Supplement for Version 2016* [2]. The attributes are stored in the stream's dictionary.

Table 7.38 *Type 10 halftone attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Must be **Halftone**.
HalftoneType	number	*(Required)* Must be 10.
HalftoneName	string	*(Optional)* If present, this specifies the name of the halftone dictionary.
Xsquare	integer	*(Required)* Length of one side of the upper square, in pixels.
Ysquare	integer	*(Required)* Length of one side of the lower square, in pixels.
TransferFunction	function	*(Optional)* Transfer function, which maps the interval [0 1] to the interval [0 1]. If present, this overrides the transfer function specified by the **TR** key in the ExtGState dictionary specified by the **gs** operator. This is required in a Type 10 halftone dictionary that is used as an element of a Type 5 halftone dictionary for a non-primary color component.

7.15 Patterns

PDF 1.2

Like a form, a pattern is a marking context: a self-contained description of text, graphics, or sampled images. Unlike forms, which are drawn one at a time, patterns are replicated ("tiled") at fixed intervals in *x* and *y* to cover the areas to be painted. Patterns are treated as colors; a Pattern colorspace is installed with the **cs** and **CS** operators just as for other colorspaces, and a particular pattern is installed as the current color with the **scn** and **SCN** operators. See Section 8.5.2, "Color operators."

A Pattern is specified by a PDF stream. The keys in the stream dictionary are the same as in a PostScript language pattern dictionary, with some exceptions:

- The dictionary does not contain a **PaintProc** key. Instead, as with Form XObjects, the stream contents specify the painting procedure.
- The dictionary contains a **Matrix** key, whose value is a 6-element array representing a transformation matrix that maps pattern space into default user space.

As with any stream, the dictionary must also include a **Length** key, and it may include **Filter** and **DecodeParms** keys if the stream is encoded.

Table 7.39 *Pattern attributes*

Key	*Type*	*Semantics*
Type	name	*(Required)* Must be **Pattern**.
Length	integer	*(Required)* The length of the stream containing the painting procedure.
Resources	dictionary	*(Required)* A list of all the named resources required by the pattern. See page 138.
PatternType	integer	*(Required)* Must be 1.
PaintType	integer	*(Required)* Determines how the color of the pattern cell is to be specified. The choices are: 1 *Colored pattern*. The stream itself specifies the colors used to paint the pattern cell. 2 *Uncolored pattern*. The 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.
TilingType	integer	*(Required)* Controls adjustments to the tiling to quantize it to the device pixel grid. The choices are: 1 *Constant spacing*. Pattern cells are spaced consistently—that is, by a multiple of a device pixel. 2 *No distortion*. The pattern cell is not distorted, but the spacing between pattern cells may vary by as much as one device pixel in both *x* and *y* dimensions when the pattern is painted. This achieves the spacing requested by **XStep** and **YStep** on average, but not for individual pattern cells. 3 *Constant spacing and faster tiling*. Like **TilingType** 1, but with additional distortion of the pattern cell permitted to enable a more efficient implementation.
BBox	Rectangle	*(Required)* The pattern cell bounding box, which is used to clip the pattern cell and to determine its size for caching.
XStep	number	*(Required)* The desired horizontal spacing between pattern cells, measured in the pattern cell coordinate system.
YStep	number	*(Required)* The desired vertical spacing between pattern cells, measured in the pattern cell coordinate system.
Matrix	array	*(Optional; the default is the identity matrix)* A transformation matrix that maps the pattern's coordinate space into default user space.
XUID	array	*(Optional)* An extended unique ID that uniquely identifies the pattern; see section 5.8.2, "Extended Unique ID Numbers," of the *PostScript Language Reference Manual, Second Edition* for more details.

Note **XStep** *and* **YStep** *may differ from the dimensions of the pattern cell implied by the* **BBox** *entry. This enables tiling with irregularly shaped figures.* **XStep** *and* **YStep** *may be either positive or negative, but not zero.*

Example 7.6 shows a "bitmap pattern," represented by an 8-by-8 cell that contains an imagemask. In this example, we set the color space to a Pattern color space named `/CS1` and set the color to a pattern named `/P1`; both of those are defined in the Page's Resources dictionary (object 3). The Page then draws a 1-inch square using that color. The pattern itself contains an inline image; its Resources dictionary (object 8) therefore includes the **ImageB** ProcSet; the Page's Resources dictionary does not.

Example 7.6 *Bitmap pattern*

```
1 0 obj
<<
/Type /Page
/Parent 7 0 R
/Resources 3 0 R
/Contents 2 0 R
>>
endobj
2 0 obj
<< /Length 31 >>
stream
/CS1 cs /P1 scn
0 0 72 72 re
f
endstream
endobj
3 0 obj
<<
/ProcSet [/PDF]
/Pattern << /P1 4 0 R >>
/ColorSpace << /CS1 [/Pattern /DeviceGray] >>
>>
endobj
4 0 obj
<<
/Type /Pattern
/PatternType 1
/Resources 8 0 R
/PaintType 1
/TilingType 1
/BBox [0 0 8 8]
/XStep 8
/YStep 8
/Length 49 0 R
>>
stream
```

```
0 g
q
8 0 0 -8 0 8 cm
BI
/W 8
/H 8
/BPC 1
/IM true
/F /AHx
ID
7F FF DD FF FF 7F DD FF>
EI
Q
endstream
endobj
8 0 obj
<< /ProcSet [/PDF /ImageB] >>
endobj
```

Figure 7.7 shows the PDF version of the Star pattern from Example 4.18 on page 206 of the *PostScript Language Reference Manual, Second Edition*. The painting procedure is more detailed in PDF because it does not have the programming constructs such as **repeat** that PostScript does. Consequently, the positions, which involve constants like sin π/5, are expressed as literal numbers, as opposed to expressions that evaluate to those numbers.

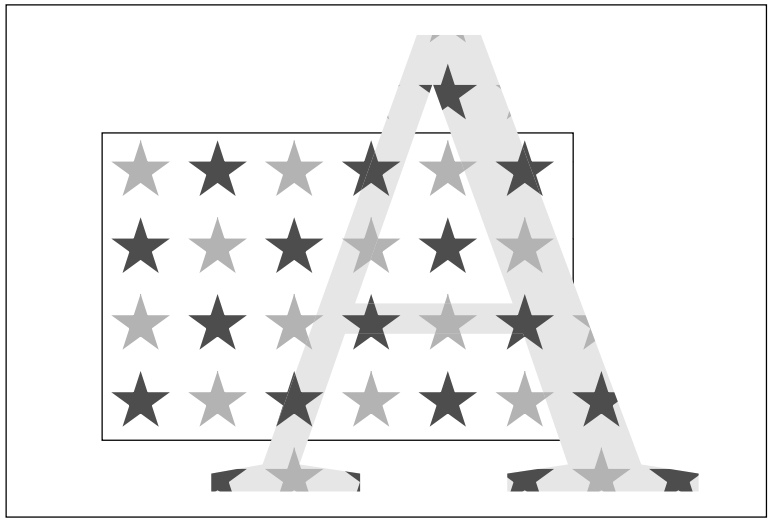
Example 7.7 *Star pattern*

```
1 0 obj
<<
/Type /Page
/Parent 8 0 R
/Resources 3 0 R
/Contents 2 0 R
>>
endobj
2 0 obj
<< /Length 18 0 R>>
stream                        % The Page content
/Pattern cs /P1 scn           % Set the pattern
120 120 184 120 re
f                  % Fill the rectangle with it
0 G
120 120 184 120 re
S
BT
/F1 1 Tf
270 0 0 270 160 100 Tm
0.9 g
```

```
(A)Tj
/CS1 cs /P1 scn              % Set the pattern
0 0 TD
(A)Tj                       % Show text with it
ET
endstream
endobj
3 0 obj                 % The Page's resources
<<
/ProcSet [/PDF /Text]
/Font << /F1 4 0 R >>
/Pattern << /P1 5 0 R >>
/ColorSpace << /CS1 [/Pattern /DeviceGray] >>
>>
endobj
4 0 obj
<<
/Type /Font
/Subtype /Type1
/Name /F1
/BaseFont /Times-Roman
>>
endobj

5 0 obj                          % The pattern
<<
/Type /Pattern
/PatternType 1
/Resources 9 0 R
/PaintType 1 /TilingType 1
/BBox [0 0 60 60]
/XStep 60 /YStep 60
/Length 33 0 R
>>
stream
0.3 g
15 27 m
7.947 5.292 l
26.413 18.708 l
3.587 18.708 l
22.053 5.292 l
f
45 57 m
37.947 35.292 l
56.413 48.708 l
33.587 48.708 l
52.053 35.292 l
f
```

```
0.7 g
15 57 m
7.947 35.292 l
26.413 48.708 l
3.587 48.708 l
22.053 35.292 l
f
45 27 m
37.947 5.292 l
56.413 18.708 l
33.587 18.708 l
52.053 5.292 l
f
endstream
endobj
9 0 obj              % The pattern's resources
<< /ProcSet [/PDF] >>
endobj
```

Figure 7.7 *Star Pattern*



7.16 Property lists

PDF 1.2

A Property List is a dictionary that provides information about a sequence of graphics objects or a particular place with the stream of marking operators. It may be written in-line, or it may appear in the **Properties** sub-dictionary of the current Resources dictionary. Property lists are used by the Marked Content operators (see Section 8.10.3 on page 239).

Property Lists provide an open-ended extension mechanism that allows a set of key-value pairs to be associated with particular marks in a content stream. With the exception of the **Subtype** key, no particular keys are defined by PDF. It is suggested, however, that any particular extension use keys 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). The meanings of the extended keys are determined by the extensions that create and use those keys.

Table 7.40 *Property List attributes*

Key	*Type*	*Semantics*
Subtype	name	*(Optional)* Object subtype. Intended to indicate the application or extension that defines the property list.

CHAPTER 8

Page Descriptions

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

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

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

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

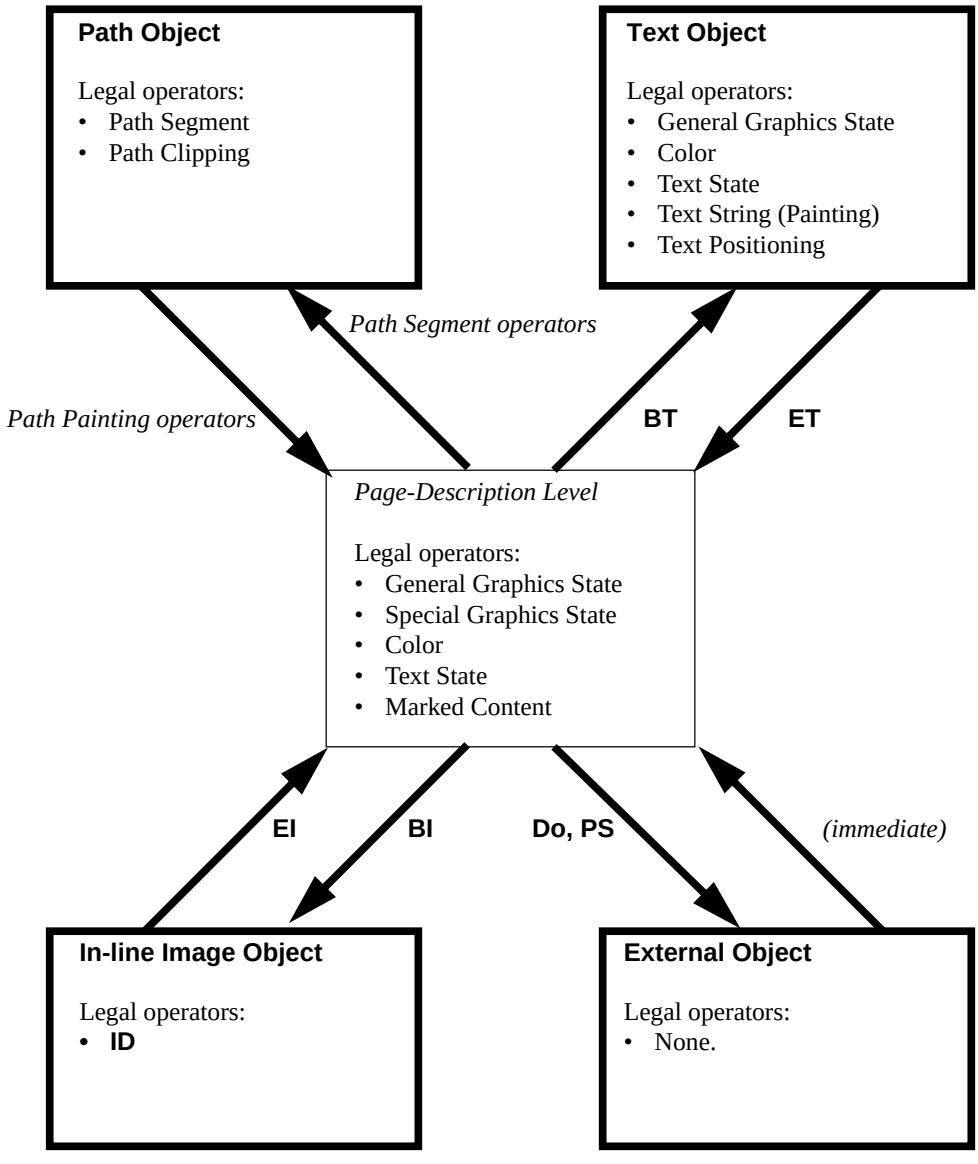
8.1 Overview

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

PDF provides four types of graphics objects:

- A *path object* is an arbitrary shape made of straight lines, rectangles, and cubic curves. A path may intersect itself and may have disconnected sections and holes. A path object includes a painting operator that specifies whether the path is filled, stroked, and/or serves as a clipping path.
- A *text object* consists of one or more character strings that can be placed anywhere on the page and in any orientation. Like a path, text can be stroked, filled, and/or serve as a clipping path.
- An *image object* consists of a set of samples using a specified color model. Images can be placed anywhere on a page and in any orientation.

Figure 8.1 *Graphics Objects*



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

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

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

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

8.2 Graphics state

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

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

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

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

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

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

8.3 Special Graphics State

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

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

8.3.1 Special Graphics state parameters

8.3.1.1 Clipping path

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

8.3.1.2 Current transformation matrix

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

8.3.1.3 Current point

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

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

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

8.3.2 Special Graphics State operators

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

q **gsave**

Saves the current graphics state on the graphics state stack.

Q **grestore**

Restores the graphics state to the most recently saved state. Removes the most recently saved state from the stack and makes it the current state.

a b c d e f **cm** **concat**

Modifies the CTM by concatenating the specified matrix. Although the operands specify a matrix, they are passed as six numbers, not an array.

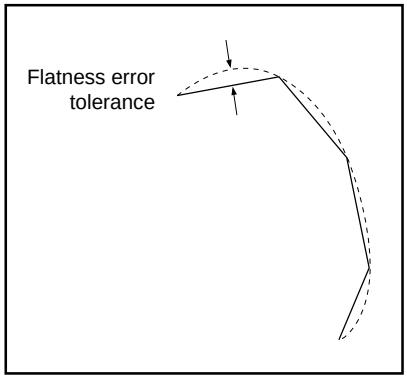
8.4 General Graphics state

8.4.1 Flatness

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

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

Figure 8.2 *Flatness*



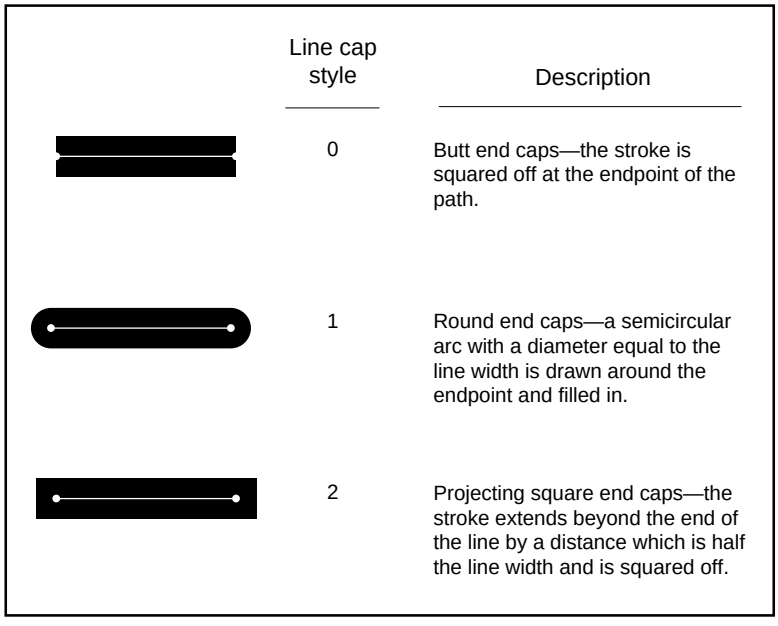
flatness **i** **setflat**

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

8.4.2 Line cap style

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

Figure 8.3 *Line cap styles*



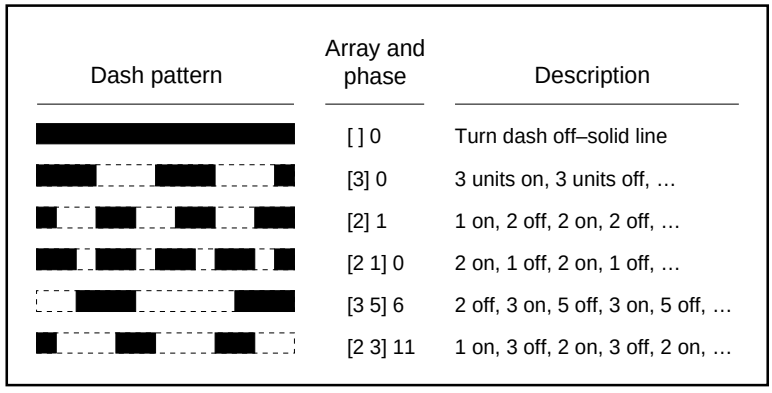
linecap **J** **setlinecap**

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

8.4.3 Line dash pattern

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

Figure 8.4 *Line dash pattern*



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

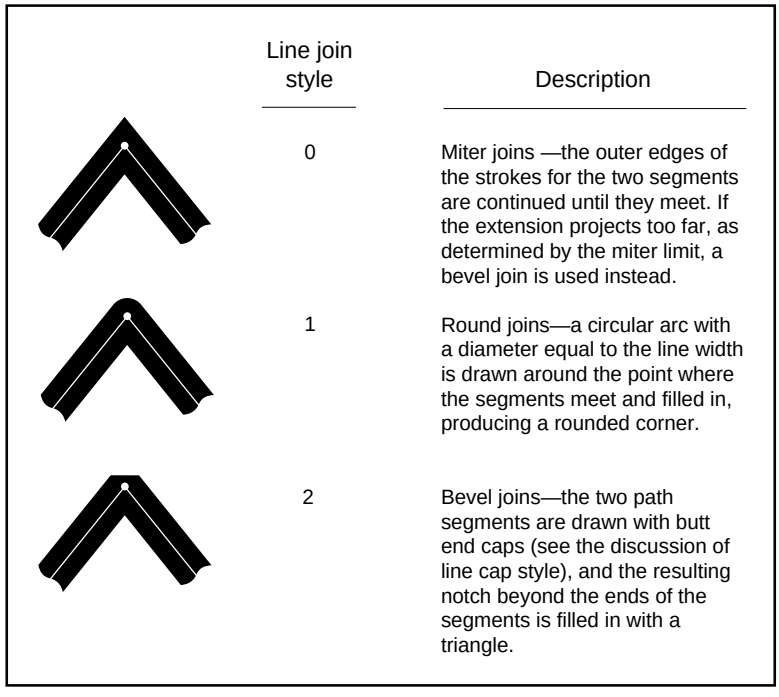
[array] phase **d** **setdash**

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

8.4.4 Line join style

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

Figure 8.5 *Line join styles*



linejoin **j** **setlinejoin**

Sets the line join parameter in the graphics state. *linejoin* has a default value of 0.

8.4.5 Line width

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

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

linewidth **w** **setlinewidth**

Sets the line width parameter in the graphics state. *linewidth* is a number and has a default value of 1.

8.4.6 Miter limit

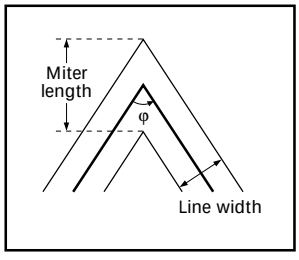
When two line segments meet at a sharp angle and mitered joins have been specified as the line join style, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a maximum on the ratio of the miter length to the line width, as shown in Figure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

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

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

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

Figure 8.6 *Miter length*



miterlimit **M** **setmiterlimit**

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

8.4.7 Generic Graphics State operator

All the remaining parameters in the General Graphics State are set with the **gs** operator, whose operand is an "extended graphics state" dictionary. (See page 189.) Each parameter uses a different keyword in this dictionary.

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

name **gs** **PDF 1.2**

Sets the chosen device-dependent parameters in the graphics state: stroke adjustment, overprint, black generation, undercolor removal, transfer function, halftone, and halftone phase. *name* is the name of an ExtGState dictionary in the current Resources dictionary.

8.4.8 Stroke adjustment **PDF 1.2**

The stroke adjustment parameter controls whether the line width and the coordinates of a stroke are automatically adjusted as necessary to produce lines of uniform thickness. For details, see section 6.5.2, "Automatic Stroke Adjustment," of the *PostScript Language Reference Manual, Second Edition*. The keyword for stroke adjustment is **SA**.

8.4.9 Overprint

PDF 1.2

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

8.4.10 Black generation

PDF 1.2

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

8.4.11 Undercolor removal

PDF 1.2

The undercolor removal function computes the amount to subtract from the cyan, magenta, and yellow components during conversion of color values from **DeviceRGB** color space to **DeviceCMYK**. See section 6.2.3, “Conversion from **DeviceRGB** to **DeviceCMYK**,” of the *PostScript Language Reference Manual, Second Edition*. The keyword for undercolor removal is **UCR**.

8.4.12 Transfer function

PDF 1.2

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

8.4.13 Halftone

PDF 1.2

The halftone parameter of the graphics state specifies how halftones should be produced. See 7.13, “Extended graphics states,” for details about halftones. For general information on halftones, see section 6.4.3, “Halftone Dictionaries,” of the *PostScript Language Reference Manual, Second Edition*. The keyword for halftone is **HT**.

8.4.14 Halftone phase

PDF 1.2

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

8.5 Color

8.5.1 Color parameters

8.5.1.1 Fill color

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

8.5.1.2 Stroke color

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

8.5.1.3 Fill color space

PDF 1.1

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

8.5.1.4 Stroke color space

PDF 1.1

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

8.5.1.5 Rendering intent

PDF 1.1

The rendering intent is a name that indicates the style of color rendering that should occur. See Section 7.11, "XObjects," and especially Table 7.25, "Color rendering intents," for further detail.

8.5.2 Color operators

PDF 1.1

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

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

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

8.5.2.1 Device-dependent color space operators

gray **g** **setgray** (fill)

Sets the color space to **DeviceGray**, and sets the gray tint to use for filling paths. *gray* is a number between 0 (black) and 1 (white).

gray **G** **setgray** (stroke)

Sets the color space to **DeviceGray**, and sets the gray tint to use for stroking paths. *gray* is a number between 0 (black) and 1 (white).

c m y k **k** **setcmykcolor** (fill)

Sets the color space to **DeviceCMYK**, and sets the color to use for filling paths. Each operand must be a number between 0 (no ink) and 1 (maximum ink).

c m y k **K** **setcmykcolor** (stroke)

Sets the color space to **DeviceCMYK**, and sets the color to use for stroking paths. Each operand must be a number between 0 (no ink) and 1 (maximum ink).

r g b **rg** **setrgbcolor** (fill)

Sets the color space to **DeviceRGB**, and sets the color to use for filling paths. Each operand must be a number between 0 (minimum intensity) and 1 (maximum intensity).

r g b **RG** **setrgbcolor** (stroke)

Sets the color space to **DeviceRGB**, and sets the color to use for stroking paths. Each operand must be a number between 0 (minimum intensity) and 1 (maximum intensity).

8.5.2.2 Generic color space operators

PDF 1.1

colorspace **cs** **setcolorspace** (fill)

Sets the color space to use for filling paths. *colorspace* must be a name. If the color space is specified by a name (the device-dependent color spaces **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**), then that name may be used. If it is specified by an array (the device-independent and special color spaces), then *colorspace* must be a name defined in the current Resources dictionary.

For example, the following expression is illegal:

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

Instead, one would write

```
/CS42 cs
```

and the Resources dictionary would contain

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

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

colorspace **CS** **setcolorspace** (stroke)

Same as **cs**, but for strokes.

$c_1\ c_2\ c_3\ c_4$ **sc** **setcolor** (fill)

Sets the color to use for filling paths. The number of operands required and their interpretation is based on the current fill color space. For **DeviceGray**, **CalGray**, and **Indexed** color spaces, one operand is required. For **DeviceRGB**, **CalRGB**, and **Lab** color spaces, three operands are required. For **DeviceCMYK** and **CalCMYK**, four operands are required.

$c_1\ c_2\ c_3\ c_4$ **SC** **setcolor** (stroke)

Same as **sc**, but for stroking paths.

$c_1 \ldots c_n$ **scn** **setcolor** (fill) **PDF 1.2**
$c_1 \ldots c_n$ *name* **scn** **setcolor** (fill for patterns)
tint **scn** **setcolor** (fill for separations)

scn accepts the same parameters, and has the same effect, as **sc**. In addition, it supports **Pattern** and **Separation** colors.

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

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

$c_1 \ldots c_n$ **SCN** **setcolor** (stroke) **PDF 1.2**
$c_1 \ldots c_n$ *name* **SCN** **setcolor** (stroke for patterns)
tint **SCN** **setcolor** (stroke for separations)

Same as **scn**, but for strokes.

8.5.2.3 Color rendering intent

intent **ri** Sets the color rendering intent in the graphics state.

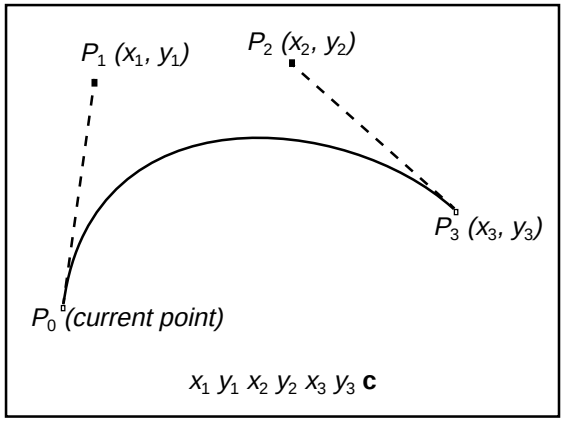
intent is a name of a color rendering intent, which indicates the style of color rendering that should occur, as described in Table 7.25 on page 179. The default rendering intent is **RelativeColorimetric**.

8.6 Paths

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

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

Figure 8.7 *Bézier curve*



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

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

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

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

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

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

<path> ::= <subpath>+
{path clipping operator}
<path painting operator>

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

8.6.1 Path segment operators

All operands are numbers that are coordinates in user space.

x y **m** **moveto**

Moves the current point to *(x, y)*, omitting any connecting line segment.

x y **l** (operator is lowercase L) **lineto**

Appends a straight line segment from the current point to *(x, y)*. The new current point is *(x, y)*.

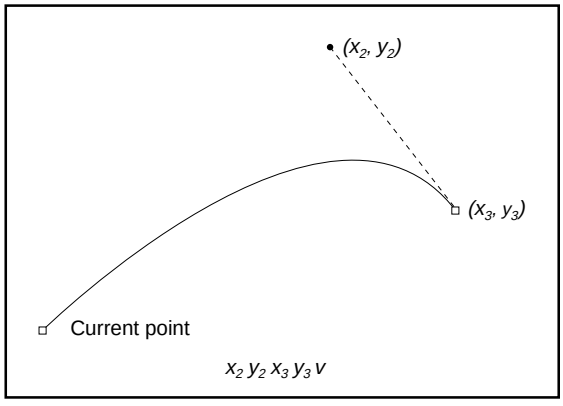
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$ **c** **curveto**

Appends a Bézier curve to the path. The curve extends from the current point to (x_3, y_3) using (x_1, y_1) and (x_2, y_2) as the Bézier control points, as shown in Figure 8.7. The new current point is (x_3, y_3).

$x_2\ y_2\ x_3\ y_3$ **v** **curveto** (first control point coincides with initial point on curve)

Appends a Bézier curve to the current path between the current point and the point (x_3, y_3) using the current point and (x_2, y_2) as the Bézier control points, as shown in Figure 8.8. The new current point is (x_3, y_3).

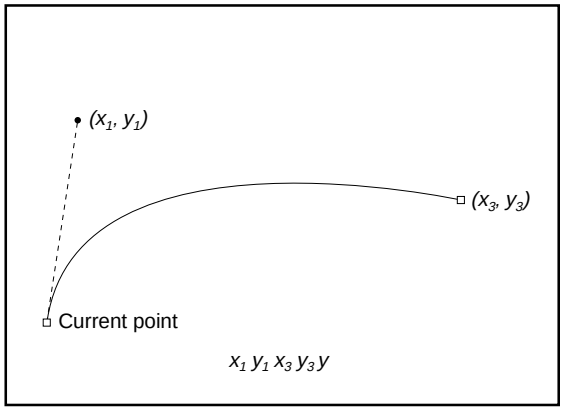
Figure 8.8 v *operator*



x$_1$ *y*$_1$ *x*$_3$ *y*$_3$ **y** **curveto** (second control point coincides with final point on curve)

Appends a Bézier curve to the current path between the current point and the point (x_3, y_3) using (x_1, y_1) and (x_3, y_3) as the Bézier control points, as shown in Figure 8.9. The new current point is (x_3, y_3).

Figure 8.9 y *operator*



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

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

```
x y width height re
```

is defined to have the same effect as the sequence

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

h

h closepath

Closes the current subpath by appending a straight line segment from the current point to the starting point of the subpath.

8.6.2 Path painting operators

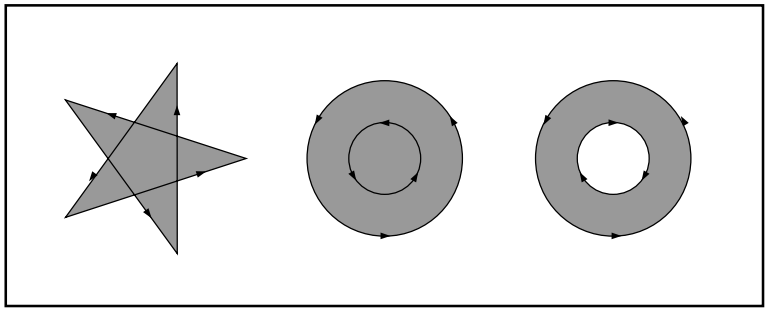
Paths may be stroked and/or filled. As in the PostScript language, painting completely obscures any marks already on the page under the region that is painted.

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

The process of filling a path paints the entire region enclosed by the path, using the fill color and fill color space. If a path consists of several disconnected subpaths, each is filled separately. Any open subpaths are implicitly closed before being filled. Closing is accomplished by adding a segment between the first and last points on the path. For a simple path, it is clear what lies inside the path and should be painted by a fill. For more complicated paths, it is not so obvious. One of two rules is used to determine which points lie inside a path.

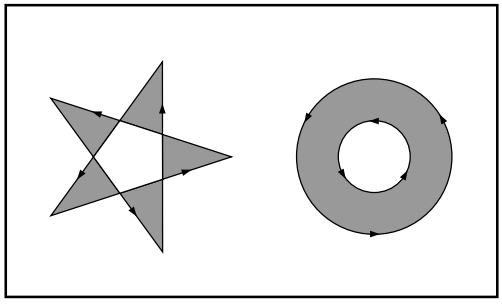
The *non-zero winding number rule* uses the following test to determine whether a given point is inside a path and should be painted. Conceptually, a ray is drawn in any direction from the point in question to infinity, and the points where the ray crosses path segments are examined. Starting from a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

Figure 8.10 *Non-zero winding number rule*



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

Figure 8.11 *Even–odd rule*



n **newpath**

Ends the path without filling or stroking it. This is a "path painting no-op," primarily used with a path clipping operator (see the next section), but like the other path painting operators, it terminates a Path Object.

S **stroke**

Strokes the path.

s **closepath** and **stroke**

Similar to the **S** operator, but closes the path before stroking it.

f **fill**

Fills the path, using the non-zero winding number rule to determine the region to fill.

F **fill**

Same as the **f** operator. Included only for compatibility. Although applications that read PDF files must be able to accept this operator, applications that generate PDF files should use the **f** operator instead.

f* **eofill**

Fills the path, using the even–odd rule to determine the region to fill.

B **fill** and **stroke**

b **closepath**, **fill**, and **stroke**

B* **eofill** and **stroke**

b* **closepath**, **eofill**, and **stroke**

8.6.3 Path clipping operators

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

Although the path clipping operator appears before the path painting operator, the path clipping operator does not alter the clipping path at the point it appears. Rather, it modifies the effect of the path painting operator. After the path is filled or stroked by the path painting operator, it is set to be the current clipping path. If the path is both filled and stroked, the painting is done in that order before making the path the current clipping path.

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

W **clip**

Uses the non-zero winding number rule to determine which regions are inside the clipping path.

W* **eoclip**

Uses the even–odd rule to determine which regions are inside the clipping path.

8.7 Text state

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

8.7.1 Text State parameters and operators

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

1. T_c is the character spacing parameter.
2. T_w is the word spacing parameter.
3. T_h is the horizontal spacing parameter.

4. T_l is the “leading” parameter.
5. T_f is the text font.
6. T_{fs} is the text font size.
7. T_m is the text matrix.
8. T_{mode} is the rendering mode.
9. T_{rise} is the “text rise.”

There are two additional parameters of the text state:

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

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

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

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

8.7.1.1 Character spacing

The character spacing parameter, T_c, is a number specified in text space units. It is added to the displacement between the origin of one character and the origin of the next. See Figure 7.3 on page 154 for examples of character origins and displacements. In the default coordinate system, the positive direction of the *x*-axis points to the right, and the positive direction of the *y*-axis points upward. So for horizontal writing, a positive value of T_c has the effect of expanding the space between characters; see Figure 8.12. For vertical writing, however, a *negative* value of T_c has the effect of expanding the space between characters.

Figure 8.12 *Character spacing for horizontal writing*

Character	0 (default)
C h a r a c t e r	5

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

charSpace **Tc** Set character spacing

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

8.7.1.2 Word spacing

The word spacing parameter, T_w, is a number specified in text space units. It works in the same way as character spacing, but applies only to the space character, <20>. T_w is added to the displacement between the origin of the space character and the origin of the following character. For horizontal writing, a positive value for T_w has the effect of increasing the spacing between words. For vertical writing, a positive value for T_w *decreases* the space between words, since the positive direction of the *y*-axis points upward; therefore a negative value will increase the space between words. Figure 8.13 illustrates the effect of word spacing in horizontal writing.

Figure 8.13 *Effect of word spacing in horizontal writing*

Word Space	0 (default)
Word Space	10

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

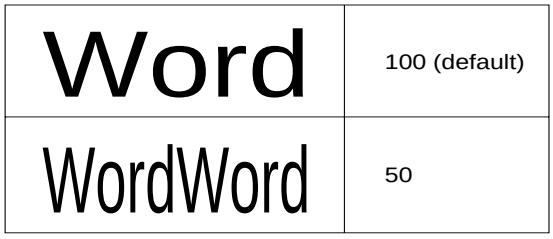
wordSpace **Tw** Set word spacing

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

8.7.1.3 Horizontal scaling

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

Figure 8.14 *Horizontal scaling*



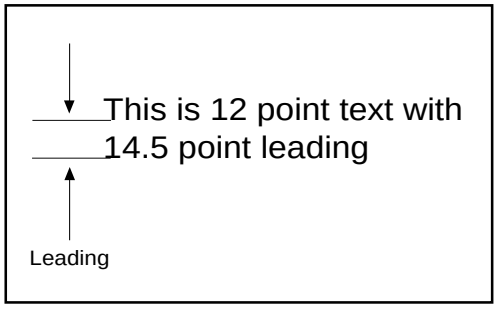
scale **Tz** Set horizontal scaling

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

8.7.1.4 Leading

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

Figure 8.15 *Leading*



leading **TL** Set text leading

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

8.7.1.5 Text font and size

fontname size **Tf** Set font and size

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

8.7.1.6 Text matrix

The text matrix specifies the transformation from text space (see Section 3.3, “Text space”) to user space. The text matrix is set with the **Tm** operator (see page 233).

8.7.1.7 Text rendering mode

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

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

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

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

Figure 8.16 *Text rendering modes*

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

render **Tr** Set the text rendering mode

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

8.7.1.8 Text rise

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

Figure 8.17 *Text rise*

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

rise **Ts** Set text rise

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

8.7.2 Text Object operators

A PDF text object consists of operators that specify character strings, movement of the current point, and text state. A text object begins with the **BT** operator and ends with the **ET** operator. See Figure 8.1 on page 210.

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

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

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

BT Begins a Text Object. Initializes the text matrix, T_m, and the line matrix, T_{LM}, to the identity matrix.

ET Ends a Text Object. Discards the text matrix.

8.7.3 Text positioning operators

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

Note *These operators may appear only within text objects.* See Figure 8.1 on page 210.

$t_x\ t_y$ **Td** Moves to the start of the next line, offset from the start of the current line by (t_x, t_y). t_x and t_y are numbers expressed in text space units. More precisely, **Td** performs the following assignments:

$$T_m = T_{LM} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \times T_{LM}$$

$t_x\ t_y$ **TD** Moves to the start of the next line, offset from the start of the current line by (t_x, t_y). As a side effect, this sets the leading parameter in the text state.

$t_x\ t_y$ `TD` is defined to have the same effect as `-`t_y `TL` $t_x\ t_y$ `Td`

a b c d x y **Tm** Sets the text matrix, T_m, and the text line matrix, T_{LM}. It also sets the current point and line start position to the origin. **Tm** performs the following assignments:

$$T_m = T_{LM} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ x & y & 1 \end{bmatrix}$$

The operands are all numbers, and the initial value for T_m and T_{LM} is the identity matrix, [1 0 0 1 0 0]. Although the operands specify a matrix, they are passed to **Tm** as six numbers, not as an array.

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

T* Moves to the start of the next line.

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

where T_l is the leading parameter of the text state.

8.7.4 Text rendering

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

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

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

This calculation occurs, in effect, whenever any of the text parameters change, before **Tj** or **TJ** occur. When text is rendered, the text line matrix, T_{LM}, is unaffected, but the text matrix, T_m, is translated by the origin-displacement of the text, which affects subsequent rendering operations, as shown above. For horizontal-mode writing, the origin-displacement is along the *x* axis; for vertical writing (see Section 7.6.12 on page 154), the displacement is along the *y* axis.

8.7.5 Text string operators

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

PDF supports the same conventions as the PostScript language for specifying non-printable ASCII characters. That is, a character can be represented by an escape sequence, as described in Table 4.1 on page 44.

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

string **Tj** **show**

Shows text string, using the character and word spacing parameters from the text state.

string **'** **show**

Moves to next line and shows text string, using the character and word spacing parameters from the text state.

string **'** is defined to have the same effect as `T*` *string* `Tj`

a_w a_c string **"** **show**

Moves to next line and shows text string. a_w and a_c are numbers expressed in text space units. a_w specifies the additional space width, and a_c specifies the additional space between characters.

a_w a_c string **"** is defined to have the same effect as *a_w* `Tw` *a_c* `Tc` *string* `'`

Note *The values specified by a_w and a_c remain the word and character spacings after the* **"** *operator is executed.*

[number or string …] **TJ**
show with displacements

Shows text string, allowing individual character positioning, and using the character and word spacing parameters from the text state.

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

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

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

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

8.7.6 Text strings in multi-byte fonts

PDF 1.2

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

8.8 External objects (XObjects)

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

8.8.1 XObject operators

The **Do** operator permits the execution of an arbitrary object whose data is encapsulated within a PDF object. The currently defined XObjects are images and PostScript language forms, discussed in Section 7.11, "XObjects."

xobject **Do** Executes the specified XObject. *xobject* must be the name of an Image, Form, or PostScript XObject in the current Resources dictionary. See Section 7.11, "XObjects.

string **PS** **PDF 1.1**

The **PS** operator provides an in-line equivalent to a PostScript XObject. The **PS** operator has one argument, a string. When a **PS** operator is encountered while a document is being printed to a PostScript printer, the contents of the string are placed into the PostScript output as the argument of an instance of the PostScript operator **exec**. This string is copied without interpretation and may include PostScript comments. In any other case, the **PS** operator has no other effect. See Section 7.11.5 on page 181 for additional information.

8.9 In-line image objects

In addition to the Image XObject described in Section 7.11, "XObjects," PDF supports in-line images. An in-line Image Object consists of the operator **BI**, followed by Image XObject key–value pairs, followed by the operator **ID**, followed by the image data, followed by **EI**:

<in-line image> ::=
BI
<Image XObject key–value pairs>
ID
{<lines of data>}+
EI

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

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

Table 8.1 *Abbreviations for in-line image names*

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

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

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

BI Begins image

ID Begins image data

EI Ends image

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

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

where *base* is a device-dependent color space and *lookup* is a string; see Section 7.10.7, "Indexed color spaces." The name `/Indexed` may be abbreviated as `/I`.

In PDF 1.2, the value may also be the name of a color space in the current Resources dictionary. In this case, any color space that may be used with an Image XObject may be used for the in-line image (see Section 7.11.1, "Images").

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

Example 8.1 *In-line image*

```
q
17 0 0 17 298 388 cm
BI
/W 17
/H 17
/BPC 8
/CS /RGB
/F [/A85 /LZW]
ID
J1/gKA>.]AN&J?]-<HW]aRVcg*bb.\eKAdVV%/PcZ
… omitted data …
R.s(4KE3&d&7hb*7[%Ct2HCqC~>
EI
Q
```

8.10 Other operators

8.10.1 Type 3 font operators

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

w_x w_y **d0** (d zero) **setcharwidth**

The operands are both numbers.

w_x w_y ll_x ll_y ur_x ur_y **d1** (d one) **setcachedevice**

The operands are all numbers.

8.10.2 Compatibility operators

PDF 1.1

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

BX

This operator directs a viewer to not report any undefined operators until a matching **EX** is encountered. (**BX**–**EX** pairs may nest.)

EX

This operator ends a section of page description in which undefined operators should not be reported.

8.10.3 Marked Content operators

PDF 1.2

The Marked Content operators are used in page descriptions such as the **Contents** stream of a page to indicate a part of the stream that may be significant to an application other than a strict PDF consumer, such as a PDF Viewer. The content that is marked is not a sequence of bytes in the stream, but a sequence of graphics objects. Each graphics object is fully qualified by the graphics state in which it is rendered.

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

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

These operators may appear only *between* graphics objects; they may not occur within a graphics object nor between a graphics state operator and its operands. See Figure 8.1 on page 210.

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

The **BMC** and **MP** operators have only one operand, a *tag* which indicates the role of the operator. The **BDC** and **DP** operators have an additional operand, a list of *properties* that are associated with the mark. The properties are represented by a dictionary. This dictionary may be written inline in the content stream if all its values are direct objects. If any value is an indirect object (referring to an object outside the stream), then the list is specified by the name of a Property List in the current Resources dictionary. (See page 140.)

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

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

tag **BMC** Begin marked content

Indicates the beginning of a sequence of graphics objects that is “marked” in some way. *tag* must be a name; it should indicate the role of the content that is marked.

tag properties **BDC** Begin marked content with a property list

Indicates the beginning of a sequence of graphics objects that is “marked” in some way. *tag* must be a name; it should indicate the role of the content that is marked. *properties* is either an inline dictionary, that is, a direct object dictionary in the content stream, or it is the name of a Property List in the current Resources dictionary.

EMC End marked content

Indicates the end of a marked sequence of graphics objects. Sequences may be nested.

tag **MP** Mark a point in the content

Indicates a place within the sequence of graphics objects that is "marked." **MP** is not intended for use when some subsequence of the content is being marked: **BMC** and **EMC** should be used when the beginning and end of a subsequence is to be indicated.

tag must be a name; it should indicate the role of the place that is marked.

tag properties **DP** Mark a point in the content and include a property list

Same as **MP**, but includes a property list, as **BDC** does.

CHAPTER 9

Linearized PDF

9.1 Introduction

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 it is compatible with all existing viewers and other PDF applications. Enhanced viewers can recognize that a PDF file has been linearized and can take advantage of that organization to enhance viewing performance.

The primary goal of the linearized PDF organization is to achieve the following behavior:

1. 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).

2. When the user requests another page of an open document (either by going to the next page or by following a link to an arbitrary page), display that page as quickly as possible.

3. When data for a page is delivered over a slow channel, display the page incrementally as it arrives. Insofar as is possible, the most useful data should be displayed first.

4. Permit user interaction, such as following a link, to be performed even before the entire page has been received and displayed.

The above 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. Re-linearizing it requires reprocessing the entire file.

Linearized PDF requires two additions to the PDF specification:

1. Rules for ordering of objects in the PDF file.

2. 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, the following presentation goes considerably beyond a simple specification of PDF extensions; it includes background, motivation, and strategies.

Section 9.2, "Background and Assumptions," provides background about the properties of the World Wide Web that are relevant to the design of linearized PDF. Section 9.3, "Linearized PDF document structure specification," specifies the file format and object-ordering requirements of linearized PDF. Section 9.4, "Hint Tables," specifies the detailed representation of the hint tables. Section 9.5, "Access Strategies," outlines strategies for accessing linearized PDF over a network, which in turn determine the optimal way in which to organize the PDF file itself.

The reader is assumed to be familiar with PDF document structure and with the basic architecture of the World Wide Web, and is assumed not to be intimidated by terms such as URL, HTTP, and MIME.

9.2 Background and Assumptions

The principal problem addressed by the linearized PDF design is accessing PDF documents through the World Wide Web. This environment has the following important properties:

1. 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.

2. 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.

3. Round-trip delay can be significant. A request-response transaction can take up to several seconds, independent of the amount of data requested.

4. The data rate may be limited. A typical bottleneck is a 14.4K or 28.8K bit/sec modem link between the client and the Internet service provider.

The above properties are generally shared by other wide-area network architectures aside from the World Wide Web. Additionally, CD-ROMs share some of these properties, since they have relatively slow seek times and limited data rates compared to magnetic media. In the remainder of this presentation, we concentrate on the World Wide Web exclusively.

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.

5. When a PDF file is initially accessed (say, 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, then inspects the MIME tag of the response as it arrives. Only at that point is the document known to be PDF. Additionally, the length of the document becomes known at that time.

6. The client can abort a response while it is still in progress, if it decides that the remainder of the data is not of any immediate interest. How quickly the abort takes effect depends on round-trip time and server responsiveness. In HTTP, aborting the transaction requires closing the connection, which will interfere with the strategy of caching the open connection between transactions.

7. The client can request retrieval of portions of a document by specifying one or more byte ranges (*offset, count*) as part of the URL. 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.

8. The client can initiate multiple concurrent transactions in an attempt to obtain multiple responses in parallel. This is commonly done, for instance, to retrieve in-line images referenced from a HTML document. This strategy isn't always reliable and may backfire if the transactions interfere with each other by competing for scarce resources in the server or the communication channel.

   We have experimentally determined that multiple concurrent transactions don't 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, we make some additional assumptions about the PDF viewer and its local environment.

9. The viewer 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.

10. The viewer 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.

9.3 Linearized PDF document structure specification

Except as noted below, all elements of a linearized PDF file are as specified in Chapter 5.

9.3.1 File structure

Except as noted, all indirect objects in the PDF file are numbered sequentially in two groups, based on their order of appearance in the file.

- The first group consists of the 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 second group. (The stream containing the hint tables may be numbered out of sequence; see Section 9.3.5, “Hint Streams.”)

- The second group consists of all remaining objects in the document, including all pages after the first, all shared objects, etc. These are numbered sequentially, starting at 1.

These groups of objects are indexed by precisely two cross-reference table sections, located as shown below. The composition of these groups is discussed in more detail in the sections that follow. All objects have a generation number of 0.

Example 9.1 *Outline of a linearized PDF file*

Part 1: Header

```
%PDF-1.1
% binary stuff
```

Part 2: Linearization parameters

```
43 0 obj
<<
/Linearized 1                                   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 xref table (Part 11)
>>
endobj
```

Part 3: First Page xref 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 xref table entries in document
/Prev 52776                 offset of main xref table (Part 12)
/Root 44 0 R          indirect reference to Catalog (Part 4)
…any other attributes, e.g., Info, Encrypt… (Part 9)
>>
startxref
0                                       dummy xref table offset
%%EOF
```

Part 4: Catalog and other required document-level objects

```
44 0 obj
<<
/Type /Catalog
/Pages 42 0 R
>>
endobj
…other objects…
```

Part 5: Primary Hint Stream (Note: Parts 5 and 6 may be placed in the opposite order)

```
56 0 obj
<<
/Length 457
…possibly other stream attributes, e.g., Filter…
```

```
/P 0                          position of Page Offset hint table
/S 221                        position of Shared Objects 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's objects

```
45 0 obj
<<
/Type Page
…
>>
```
Outlines tree (if the PageMode in the Catalog is **UseOutlines***)*
…objects for first page, including both shared and non-shared resources…

Part 7: All remaining pages (each Page object is followed by the non-shared objects for that page)

```
1 0 obj
<<
/Type /Page
```
…other Page attributes, such as **MediaBox**, **Parent**, *and* **Contents**…
```
>>
```
…objects for that page, including only non-shared resources…
…other pages and their non-shared objects…
last page
…objects for last page, including only non-shared resources…

Part 8: Shared objects for all pages except the first
…shared objects…

Part 9: Other objects not associated with pages, if any
…other objects…

Part 10: Overflow Hint Stream (optional)
Overflow Hint Stream

Part 11: Main xref table and trailer

```
xref
0 43
0000000000 65535 f
```
…cross-reference entries for all except First Page's objects…
```
trailer
<<
/Size 43
```
This trailer does not need to contain any other attributes; in particular, it should not have a **Prev** *attribute.*
```
>>
startxref
257                           offset of First Page xref table (Part 3)
```

%%EOF

9.3.2 Header and linearization information

The file begins with the standard `%PDF-1.1` or `%PDF-1.2` header line. Linearization is independent of PDF version number and can be applied to any PDF file version 1.1 or greater.

The *binary stuff* following the percent sign on the second line is some text that includes characters with codes 128 or greater, as recommended in Section 5.1 on page 61.

Following this, the first object in the body of the file (Part 2) must be an indirect dictionary object containing the parameters listed in Table 9.1. 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.)

Table 9.1 *Linearization parameters*

Parameter	*Type*	*Semantics*
Linearized	number	(*Required*) Linearized format version identification. As usual, a change in the integer part indicates an incompatible change in the linearized format. A change in the fractional part indicates an upward-compatible change. The current version is 1.0.
L (file Length)	integer	(*Required*) Length of entire file in bytes. This must be exactly equal to the actual length of the PDF file. A mismatch indicates that the PDF is not linearized and must be treated as ordinary PDF, ignoring linearization information.
H (Hints)	array	(*Required*) Array of two or four integers, [*offset1 length1*] or [*offset1 length1 offset2 length2*]. *Offset1* is the offset from beginning of the file of the Primary Hint Stream. (This is the beginning of the stream object, not the beginning of the stream data.) *Length1* is the length of this stream (including stream object overhead). If there is an Overflow Hint Stream, *offset2* and *length2* specify its offset and length.
O (Object number)	integer	(*Required*) Object number of the first page's Page object.
E (End of first page)	integer	(*Required*) Offset of the end of the first page (i.e., the end of Part 6), relative to the beginning of the file.
N (Number of pages)	integer	(*Required*) Number of pages in the document.
T (main xref Table)	integer	(*Required*) Location of the first entry of the main cross-reference table (the entry for object number 0). Note that this differs from the **Prev** attribute of the First Page Trailer, which gives the location of the **xref** line that precedes the table.
P (First Page number)	integer	(*Optional*) Page number of the first page (see Section 9.3.6, "First Page's objects"). The default value is 0.

The **Linearized** dictionary must be entirely contained within the first 1024 bytes of the PDF file. This limits the amount of data a viewer must read before deciding that the file is not linearized.

9.3.3 First Page Cross-Reference and Trailer

(Part 3) This is the cross-reference table for all the first page's objects (discussed in Section 9.3.6, "First Page's objects"), as well as for the Catalog and document-level objects appearing before the first page (discussed in Section 9.3.4, "Catalog and document-level objects"). Additionally, it contains entries for the **Linearized** dictionary (at the beginning) and the Primary Hint Stream (at the end).

It is a valid cross-reference table section as defined in Section 5.4 on page 63, although its position in the file is rather unconventional. The table consists of a single cross-reference subsection, with no free entries.

The **startxref** line at the end of the file gives the offset of the First Page Cross-Reference Table. The First Page Trailer's **Prev** entry gives the offset of the main cross-reference table near the end of the file. Once again, this is valid PDF, though the trailers are linked in an unusual order. A PDF viewer 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** attributes, as well as any other attributes needed to display the document. The **Size** must be the combined number of entries in both the First Page Cross-Reference Table and the main cross-reference table.

This trailer may optionally end with **startxref**, an integer, and **%%EOF**, just the same as an ordinary trailer. This information is ignored.

9.3.4 Catalog and document-level objects

(Part 4) After the First Page Cross-Reference Table must appear the Catalog dictionary and other objects that are required when opening the document. These objects include:

- The **Catalog** object.
- If the Catalog contains **PageMode** or **OpenAction** entries, those objects must be located here also, with the exception of the Outlines tree: if **PageMode** is **UseOutlines**, the entire Outlines tree is located in Part 6; otherwise it is located in Part 9. See Section 9.3.9 on page 254 for details.
- The Encryption dictionary referenced from the **Encrypt** attribute, if any, of the First Page Trailer. All attribute values in this dictionary must be located here also.
- The **Threads** array in the Catalog, if any, along with all the thread dictionaries that it refers to. This does not include the threads' **Info** dictionaries or the individual beads of the threads.

- The **AcroForm** dictionary in the Catalog, if any. Only the top-level dictionary is needed, not the objects that it refers to.

Objects that are not ordinarily needed when opening the document should not be located here but instead should be at the end of the file; see Section 9.3.9, "Other objects." This includes **Info**, **Pages**, and **Dests**.

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.

9.3.5 Hint Streams

(Part 5) The core of the linearization information is stored in data structures known as *hint tables*, whose format is described in Section 9.4 on page 256. 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-ins to application-specific information.

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 is still a valid PDF file that just isn't linearized any more.

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 this stream anywhere in the document, so it is not logically part of the document; any operation that regenerates the document will 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 9.1, and the First Page section is shown as Part 6. The order of these two parts may be reversed. See Section 9.5, "Access Strategies," for considerations on the choice of placement.

The Overflow Hint Stream, Part 10, is optional.

The location and length of the Primary Hint Stream, and of the Overflow Hint Stream if present, are given in the **Linearized** dictionary at the beginning of the file.

The hint streams are assigned the last object numbers in the file, i.e., 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.)

All attributes in the hint streams' dictionaries must be direct objects. The streams may have **Filter** and **DecodeParms** attributes.

In addition to the standard stream attributes, the dictionary of the Primary Hint Stream contains attributes giving the position of the beginning of each hint table in the stream. These positions are in bytes relative to the beginning of the stream data (after applying decoding filters, if any), and with the Overflow Hint Stream concatenated if present. The dictionary of the Overflow Hint Stream should not contain these attributes. The standard hint tables are:

Table 9.2 *Standard Hint Tables*

Key	*Hint Table*
P	(*Required*) Page Offset hint table
S	(*Required*) Shared Objects hint table
T	(*Only if thumbnails exist*) Thumbnails hint table
O	(*Only if outlines exist*) Outline hint table
A	(*Only if threads exist*) Thread info hint table
E	(*Only if named destinations exist*) Dests hint table
V	(*Only if AcroForm dictionary exists*) Forms hint table
I	(*Only if Info dictionary exists*) Info dictionary hint table

New keys may be registered for additional hint tables required for new PDF features or for application-specific data accessed by plug-ins.

Section 9.4, "Hint Tables," documents the format of the standard hint tables that are enclosed in this stream.

9.3.6 First Page's objects

(Part 6) As mentioned above, this section may either precede or follow the Primary Hint Stream. The starting file offset and length of this section may be determined from the hint tables. Additionally, the **E** attribute in the **Linearized** dictionary specifies the end of the first page, and the **O** attribute gives the first page's Page object number.

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, i.e., the leftmost leaf Page object in the Pages tree. However, if the Catalog contains an **OpenAction** 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 **Linearized** dictionary at the beginning of the file.

Implementation note *Acrobat 3.0 always treats page 0 as the first page for linearization, regardless of OpenAction.*

The objects contained here should include:

1. **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 **Linearized** dictionary. This Page object must explicitly specify all required attributes, such as **Resources** and **MediaBox**; the attributes cannot be inherited from ancestor **Pages** objects.

2. The entire Outlines tree, if the **PageMode** key in the Catalog is **UseOutlines**. (If the **PageMode** key is omitted or has some other value and the document has an Outlines tree, then it appears in Part 9. See Section 9.3.9 on page 254 for details.

3. All objects that the Page object refers to, to arbitrary depth. This includes **Contents**, **Resources**, **Annots**, and **B** (Beads), but excludes **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 objects, 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) must contain a **T** attribute referring to the associated thread dictionary.

3. The **Resources** 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 Contents stream. If Contents is 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 FontFiles referenced from FontDescriptors (see below).

5. **Contents**. If it is large, it should be represented as an array of indirect references to streams, which in turn are interleaved with the resources that they require. If it is small, the entire Contents should be a single 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 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 draws substitute fonts until the real ones have arrived. Only those fonts for which substitution is possible can be deferred in this way.[1]

See Section 9.5, “Access Strategies,” for additional discussion about object order and incremental drawing strategies.

9.3.7 Objects contained in remaining pages

(Part 7) This part of the file contains the non-shared 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, described below. 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, 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, since rendering of images is deferred.

9.3.8 Shared objects

(Part 8) This portion of the file contains objects, primarily named resources, that are referenced from more than one page and that are not referenced (directly or indirectly) from the first page. The hint tables contain an index of these objects.

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.

Implementation note *Acrobat 3.0 does not generate Shared Object groups containing more than one object.*

9.3.9 Other objects

(Part 9) Following the shared objects are any other objects that are part of the document but aren’t 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.

---

1. Currently, this includes any Type 1 or TrueType font that has a **FontDescriptor** and whose Flags bit 6 is set (indicating Adobe standard Roman character set). The base 14 Type 1 fonts cannot be deferred, although it is unlikely that they would be embedded.

- Pages tree. This can be located here, since the linearized PDF viewer never needs to consult it. Note that all Resources and other inheritable attributes of the Pages objects must be pushed down and replicated in each of the leaf Page objects (but they may contain indirect references to shared objects).

- Thumbnails. These should simply be ordered by page number. Note that the thumbnail for page 0 should be first, even if the first page of the linearized PDF is some page other than 0. Each thumbnail consists of one or more objects. These objects may refer to objects in the Thumbnail Shared Objects section (see the next item).

- Thumbnails Shared Objects. These are objects that are shared among some or all thumbnail objects and are not referenced from any other objects.

- Outline tree, 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. This is a preorder traversal of the tree, skipping over any subtree that is closed (i.e., whose parent's **Count** is negative). Following that should be the subtrees that were skipped over, in the order that they would have appeared if they were all open.

- Thread info dictionaries, referenced from the **I** (Info) attributes of thread dictionaries. Note that the thread dictionaries themselves are co-located with the Catalog, and the beads with the individual pages.

- Named destinations. These objects include the **Dests** or **Names** attribute of the Catalog and all the destination objects that it refers to. See Section 9.5.2, "Opening at an arbitrary page."

- Info dictionary and the objects contained within it.

- AcroForm tree. This does not include the top-level AcroForm dictionary, which is co-located with the Catalog.

- Other entries in the Catalog that aren't referenced from any page.

9.3.10 Main cross-reference and trailer

(Part 11) This 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 (prior to appending any "updates"). It must conform to the PDF rules for this table:

- 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**.

9.4 Hint Tables

There are two or more hint tables, as indicated by the attributes of the Primary Hint Stream (see Section 9.3.5, “Hint Streams”). The format of the standard hint tables is described below.

There can be additional hint tables for application-specific data accessed by plug-ins. A generic format for such hint tables is defined; see Section 9.4.4, “Generic hint tables.” Alternatively, the format of a hint table can be private to the application.

Each hint table consists of a portion of the stream, beginning at the position in the stream indicated by the corresponding stream attribute. (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 *offset1* and *length1* in the **H** array in the **Linearized** 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 that gets put into 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 **Linearized** 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.

9.4.1 Page Offset hint table

The Page Offset hint table gives information required to locate each page. Additionally, for each page except the first, the page entry enumerates all shared objects that the page references, directly or indirectly.

This table consists of a header section, described in Table 9.3, followed by one or more per-page entries, described in Table 9.4.

Table 9.3 *Page Offset hint table, header section*

Item	*Size (bits)*	*Description*
1	32	Least number of object in a page.
2	32	Location of first page's Page object.
3	16	Bits needed to represent the greatest number of objects in a page.
4	32	Least length of page in bytes.
5	16	Bits needed to represent the greatest page length.
6	32	Least start of Contents offset.
7	16	Bits needed to represent the greatest start of Contents offset.
8	32	Least Contents length.
9	16	Bits needed to represent the greatest Contents length.
10	16	Bits needed to represent the greatest number of Shared Object references.
11	16	Bits needed to identify a Shared Object.
12	16	Bits needed to represent numerator of fraction (see below).
13	16	Denominator used to divide page Contents into fractions. For each shared object referenced from a page, there is an indication of where in the page's Contents the object is first referenced. That position is given as the numerator of a fraction, whose denominator is specified once for the entire document. The fraction is explained in more detail below.

Table 9.4 *Page Offset hint table, per-page entry*

Item	*Size (bits)*	*Description*
1	(see Table 9.3, item 3)	This value, when added to the least number of objects in a page (Table 9.3, 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** attribute in the **Linearized** dictionary at

		the beginning of the file. The first object of the second page has an object number of 1. Objects numbers for subsequent pages can be determined by accumulating the number of objects in all previous pages.
2	(Table 9.3, item 5)	This value, when added to the least page length (Table 9.3, item 4), gives the total length of the page in bytes. The location of the first object of the first page can be determined from the cross-reference table entry for that object (see above). The locations of subsequent pages can be determined by accumulating the lengths of all previous pages. Note that one must skip over the Primary Hint Stream, wherever it is located.
3	(Table 9.3, item 7)	This value, when added to the least start of Contents offset (Table 9.3, item 6), gives the offset in bytes of the start of the Contents stream, relative to the beginning of the page. This is the offset of the stream object, not the stream data.
4	(Table 9.3, item 9)	This value, when added to the least Contents length (Table 9.3, item 8), gives the length of the Contents stream in bytes. This includes object overhead preceding and following the stream data
5	(Table 9.3, item 10)	Number of shared objects referenced from page. Note that this must be 0 in the first page's entry.
6…	*variable*	Shared Object references, each consisting of a Shared Object identifier (see Table 9.3, item 11) followed by the numerator of a fractional position (see Table 9.3, item 12). A Shared Object identifier is an index into the Shared Object hint table, described in Section 9.4.2, "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. The fraction indicates where in the page's Contents the shared object is first referenced. It is interpreted as the numerator of a fraction, whose denominator is specified once for the entire document. If the denominator is d, a numerator of 0 indicates that the first reference lies in the interval $0/d$ to $1/d$ of the Contents. Similarly, a numerator of $d-1$ indicates that the first reference lies in the interval $(d-1)/d$ to the end of the Contents. The numerator can take on two (or more) additional values, which indicate that the shared object is not referenced from the Contents 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 non-shared objects that are at the end of the page. The value $d+1$ 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 Contents stream. The relationship between positions in the unencoded and encoded streams is not necessarily linear.

9.4.2 Shared Object hint table

The Shared Object hint table gives information required to locate shared objects (see Section 9.3.8, “Shared objects”). Shared objects can be physically located in either of two places. Objects that are referenced from the first page are co-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 Page Offset hint table refers to an entry in the Shared Object hint table by a simple index that is its sequence in the table, counting from 0.

This table consists of a header section, described in Table 9.5, followed by one or more Shared Object Group entries, described in Table 9.6. 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.

For convenience of representation, the first page is treated as if it consisted entirely of shared objects. That is, the first entry refers to the beginning of the first page and has an object count and length that span all the initial non-shared objects. The next entry refers to a group of shared objects. Subsequent entries span additional groups of either shared or non-shared objects consecutively, until all shared objects in the first page have been enumerated. (Obviously, the entries that refer to non-shared objects will never be used.)

Table 9.5 *Shared Object hint table, header section*

item	*Size (bits)*	*Description*
1	32	Object number of first object in Shared Objects section (part 8).
2	32	Location of first object in Shared Objects section.
3	32	Number of Shared Object entries for first page.
4	32	Number of Shared Object entries for Shared Objects section.
5	16	Bits needed to represent the greatest number of objects in a Shared Object Group.
6	32	Least length of a Shared Object Group in bytes.
7	16	Bits needed to represent the greatest length of a Shared Object Group.

Table 9.6 *Shared Object hint table, Shared Object Group entry*

Item	Size (bits)	Description
1	(see Table 9.5, item 5)	This value plus 1 gives 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** attribute in the **Linearized** dictionary at the beginning of the file. Objects 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 Shared Object hint table header (Table 9.5, item 1).
2	(Table 9.5, item 7)	This value, when added to the least Shared Object Group length (Table 9.5, item 6), gives the total 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 9.3, 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 Shared Objects hint table, header section (Table 9.5, item 2).
3	1	Signature present flag (1 if present, 0 if absent).
4	128	(*Optional*) Signature. The signature is 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.

Implementation note *Signatures are not implemented in Acrobat 3.0. The signature present flag must be 0.*

9.4.3 Thumbnails hint table

This table consists of a header section, described in Table 9.7, followed by one or more per-page entries, described in Table 9.8.

Each entry describes the thumbnail for a single page. The pages are considered in page number order, starting at page 0 (even if page 0 is not the first page of the file). Thumbnails can exist for some but not all pages.

Table 9.7 *Thumbnails hint table, header section*

Item	Size (bits)	Description
1	32	Object number of first object in Thumbnail section.
2	32	Location of first object in Thumbnail section.
3	32	Number of entries in thumbnail table.

4	16	Bits needed to represent the greatest number of consecutive pages that have no thumbnails. This can be zero, meaning that all pages have thumbnails.
5	32	Least length of thumbnail in bytes.
6	16	Bits needed to represent the greatest length of a thumbnail.
7	32	Least number of objects in a thumbnail.
8	16	Bits needed to represent the greatest number of objects in a thumbnail.
9	32	Object number of first object in Thumbnail Shared Objects section (a subsection of Part 8). Those are objects 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.
10	32	Location of the first object in Thumbnail Shared Objects section.
11	32	Number of Thumbnail Shared Objects.

Table 9.8 *Thumbnails hint table, per-page entry*

Item	*Size (bits)*	*Description*
1	(see Table 9.7, item 4)	(*Optional*) Count of preceding pages lacking thumbnails. This indicates how many pages without thumbnails lie between the previous entry's page and this one.
2	(Table 9.7, item 8)	This value, when added to the least number of objects in a thumbnail (Table 9.7, item 7), gives the number of objects in this thumbnail.
3	(Table 9.7, item 6)	This value, when added to the least length of a thumbnail (Table 9.7, item 5), gives the length of this thumbnail in bytes.

9.4.4 Generic hint tables

Certain categories of objects are associated with the document as a whole rather than with individual pages (see Section 9.3.9, "Other objects"). 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 9.3.5, "Hint Streams").

There is a generic representation for such hints, specified below. This representation is useful for some standard categories of objects, such as outlines, threads, and named destinations. It may also be useful for application-specific objects accessed by plug-ins. It is considerably more convenient for a plug-in to use the generic hint representation than to specify custom hints.

A generic hint table describes a single group of objects that are located together in the PDF file. See Table 9.9.

Table 9.9 *Generic Hint Table*

item	*Size (bits)*	*Description*
1	32	Object number of first object in group.
2	32	Location of first object in group.
3	32	Number of objects in group.
4	32	Length of object group in bytes.

9.4.5 Outline, Thread Info, Dests, and Info hint tables

These tables use the generic hint table representation; see Section 9.4.4, "Generic hint tables." The objects that they refer to are grouped together as described in Section 9.3.9, "Other objects."

9.4.6 Forms hint table

If an AcroForm dictionary is present, this table refers to the contents of that dictionary; see Section 9.3.9, "Other objects." A form can refer to objects that are also shared with other parts of the document. The table lists those shared objects.

The Forms hint table begins with a generic hint table, described in Section 9.4.4, "Generic hint tables." It then continues as described in Table 9.10.

Table 9.10 *Forms hint table, continued*

Item	*Size (bits)*	*Description*
5	32	Number of Shared Object references.
6	16	Bits needed for each Shared Object reference.
7…	(Table 9.3, item 11)	Shared Object references, each consisting of a Shared Object identifier. See Section 9.4.2, "Shared Object hint table."

9.5 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 to explain the rationale for the organization.

9.5.1 Opening at the first page

As indicated 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 Catalog contains an **OpenAction** 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, 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 **Linearized** 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 objects. 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.

9.5.2 Opening at an arbitrary page

The viewer 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 (**GoToR** link action, integer page specifier);
- by named destination (**GoToR** link action, name or string page specifier);
- by thread (**Thread** link action).

Additionally, an indexed search results in opening a document by page number. Handling this case efficiently is considered 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 9.5.3, “Going to another page of an open document.”

The position of the Primary Hint Stream (Part 5) with respect to the First Page objects (Part 6) determines how quickly this can be done. If the Primary Hint Stream precedes the First Page objects, 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 objects, 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.

At the time a PDF file is linearized, one must decide whether to favor opening at the first page or opening at an arbitrary page.

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, 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 allows writing the PDF file 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 first to read the entire **Dests** or **Names** dictionary, for which a hint is present. Using this information, one can determine the page containing the specific destination identified by the name.

Opening at a thread requires the viewer first to read the entire Threads array, which is located with the Catalog at the beginning of the document. Using this information, one can 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.

9.5.3 Going to another page of an open document

Given 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:

1. The objects of the page itself, whose byte range can be determined from the entry in the Page Offset hint table.

2. The portion of the main cross-reference table referring to those objects. This can be computed from main cross-reference table location (**T** attribute in the **Linearized** dictionary) and the cumulative object number in the Page Offset hint table.

3. The shared objects referenced from the page, whose byte ranges can be determined from information in the Shared Object hint table.

4. The portion of the main cross-reference table referring to those objects, as in

(2) 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.

9.5.4 Drawing a page incrementally

The ordering of objects in pages and the organization of the hint tables is intended to allow progressive update of the display and early opportunities for user interaction when the data is arriving slowly. The viewer must recognize instances in which the targets of indirect object references haven't arrived yet 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 don't 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 hasn't arrived yet, skip over it. Whenever there is a reference to a font whose definition is an embedded **FontFile** that hasn't arrived yet, draw the text using a substitute font (if that is possible).

3. Draw the annotations.

4. Draw the images, 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.)

9.5.5 Following an article thread

As indicated earlier, the bead objects 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 can obtain the object number from the **N** (Next) or **P** (Prev) attribute of the bead. This identifies a target bead object, which is co-located with the page to which it belongs. Given this object number, the viewer can perform a binary search in the Page Offset hint table to determine which page. It can then go to that page, as discussed in Section 9.5.3, "Going to another page of an open document."

9.5.6 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 must do some additional work to validate them.

When the viewer sees that the file is longer than the length given in the **Linearized** dictionary, it must issue an additional transaction to read everything that was appended. It must then analyze the objects in that update to see if any of them modifies 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.

Section II

Optimizing PDF Files

CHAPTER 10

General Techniques for Optimizing PDF Files

The first section of this book describes the syntax allowed in a PDF file. In many cases there is more than one way to represent a particular construct, and the previous chapters do not indicate which alternative is preferred. This section describes techniques to optimize PDF files. Most optimizations reduce the size of a PDF file, reduce the amount of memory needed to display pages, or improve the speed with which pages are drawn. Some optimizations, such as sharing of resources, allow a viewer application to display a document when it may not have otherwise been possible in low memory situations. A few optimizations improve the appearance of pages.

This chapter contains techniques that can be generally applied to PDF files. Following chapters discuss optimizations specifically for text, graphics, and images.

While it may not be possible to take advantage of all the techniques described here, it is worth taking more time producing a PDF file to improve its viewing performance. A PDF file will be produced only once but may be viewed many times.

File size is a good gauge of the level of optimization, but of course the most accurate measure is the time it takes to view and print the pages of a document.

10.1 Use short names

Names in PDF files specify resources, including fonts, forms, images, and other objects. Whenever a name is used, it should contain as few characters as possible. This minimizes the space needed to store references to the object.

Instead of specifying a name as:

```
/FirstFontInPage4
/SecondImageInPage8
```

use names such as:

```
/F1
/Im8
```

Resource names need not be unique throughout a document. The names of resource objects must be unique within a given resource type within a single marking context. For example, the names of all fonts on a page must be unique.

10.2 Use direct and indirect objects appropriately

As mentioned in Chapter 4, objects contained in composite objects such as arrays and dictionaries may either be specified directly in the composite object or be referred to indirectly. Using indirect objects frequently improves performance and reduces the size of a PDF file. In addition, programs that produce PDF files sometimes must write a reference to an object into a PDF file before the object's value is known. Indirect objects are useful in this situation.

10.2.1 Minimizing object size

Although PDF allows random access to objects in a file, it does not permit random access to the substructure that may be present in a single object, such as the individual key–value pairs in a dictionary object. If a PDF viewer application needs to access a particular piece of information contained in an object, it reads the entire object. However, if it encounters an indirect object reference, it will not read the indirect object until needed. Using indirect objects minimizes the amount of extra data a PDF viewer application must read before locating the desired information.

As an example, if a PDF viewer application needs to obtain the PostScript language name of a font, it must search the appropriate Font dictionary object. If (in that dictionary object) the **Widths** array is specified directly, the application must read the entire array. If the **Widths** array is specified by an indirect reference, the application only needs to read the few bytes that specify the indirect reference and can avoid reading the **Widths** array itself.

In general, using indirect references improves the performance of a PDF file. However, there is some overhead associated with locating an indirect object, and an indirect object takes up more space than a direct object in a PDF file. Because of this, small objects should not be specified indirectly. A rough rule of thumb is that arrays with more than five elements and dictionaries with more than three key–value pairs should be stored as indirect objects.

10.2.2 Sharing objects

Indirect objects can be referred to from more than one location in a file. Because of this, using indirect objects can decrease the size of a PDF file by defining an object only once and making multiple references to it.

As an example, suppose each page in a document require the same ProcSets. Each page's Resources dictionary can refer to the same ProcSet array indirectly instead of duplicating the array.

10.2.3 Placeholder for an unknown value

Indirect objects can also be used when an object must be written at one location in a file, but its value will not be known until later in the process of writing the file. The best example of this situation is the **Length** key in the dictionary of a Stream object. The dictionary must be placed in the file ahead of the stream data itself, and must include the **Length** key, which specifies the length of the stream that follows. It may not be possible to know the length of the stream until after the data has been written, however. By specifying the value of the **Length** key as an indirect object, the length of the stream can be written after the stream.

10.3 Take advantage of combined operators

PDF provides several operators that combine the function of two or more other operators. For example, PDF defines operators that close (**h**) and stroke (**S**) a path, but also provides an operator that performs both operations (**s**). These combined operators should be used whenever possible. Table 10.1 lists the combined operators provided by PDF. Some operators in the table require one or more operands; the operands have been omitted from the table.

Table 10.1 *Optimized operator combinations*

Use…	*Instead of…*
s	**h S**
b	**h B**
TD	**Td TL**
TJ	Repeated series of **Tj** and **Td** operators
'	**Td Tj** or **T* Tj**
"	**Tc Tw Td Tj**

Note *To both fill and stroke a path, the combination operators must be used. Using the fill operator followed by the stroke operator does not work. The fill operator ends the path, leaving nothing for the stroke operator to stroke. Unlike the PostScript language, PDF does not allow you to save the path, fill it, restore the path, and stroke it, because the current path is not part of the PDF graphics state.*

10.4 Remove unnecessary clipping paths

Whenever anything is drawn on a page, all marks are made inside the current clipping path. When a clipping path other than the default (the crop box) is specified, rendering speed is reduced. If a portion of a page requires the use of a clipping path other than the default, the default clipping path should be restored as soon as possible. Text, graphics, and images are all clipped to the current clipping path, so it is important for the performance of all three to not use unnecessary clipping paths.

Restoration of a clipping path can be accomplished by saving the graphics state (which includes the clipping path) using the **q** operator before setting the new clipping path, and subsequently using the **Q** operator to restore the previous clipping path as soon as the new clipping path is no longer needed.

Note *Remember that the* **Q** *operator restores more than just the clipping path. See Section 8.3.2, "Special Graphics State operators" for a list of the graphics state parameters restored by the* **Q** *operator.*

10.5 Omit unnecessary spaces

Spaces are unnecessary before (, after), and before and after [and]. This slightly reduces the size of files.

10.6 Omit default values

A number of the parameters that affect drawing have default values that are initialized at the start of every page. (See Chapter 8.) For example, the default stroke and fill colors are both black. When drawing, do not explicitly set a drawing parameter unless the default value is not the desired value.

Similarly, many PDF objects are represented by dictionaries and some of the keys in these dictionaries have default values. Omit any keys whose default value is the desired value.

Omitting unnecessary key–value pairs and graphics and text state operators reduces the size of a PDF file and the time needed to process it.

10.7 Take advantage of forms

PDF files may contain forms, which are arbitrary collections of PDF operators that draw text, graphics, or images. The structure of a Form object is discussed in Section 7.11, "XObjects." A Form object may be used to draw the same marks in one or more locations on one or more pages.

Forms can be used, for example, to draw a logo, a heading for stationery, or a traditional form. The location and appearance of a form is controlled by the CTM in effect when the form is drawn.

The use of forms can reduce the size of a PDF file. In addition, forms that contain an XUID can be cached by PDF viewer applications and PostScript printers, improving rendering speed if the form is used multiple times.

10.8 Limit the precision of real numbers

The pixel size on most monitors is 1/72 of an inch, or 1 unit in default user space. The dot size on printers and imagesetters generally ranges from 1/300 of an inch (.24 units) to 1/2400 of an inch (.03 units). For this range of devices, it suffices to store coordinates to two digits to the right of the decimal point. However, because coordinates can be scaled, they should be written using more than two digits, but generally not more than five. Acrobat Exchange and Acrobat Reader store numbers in a fixed format that allows 16 bits for a fraction, which is equivalent to four or five decimal places.

Most monitors and printers cannot produce more than 256 shades of a given color component. Color component values, which are numbers between 0 and 1, should not be written using more than four decimal places.

10.9 Write parameters only when they change

Graphics state operators should be written only when the corresponding graphics state parameters change. Changes to graphics state parameters typically occur both when the application explicitly changes them and when the graphics state is restored using the **Q** operator.

When explicit changes are made to the value of a graphics state parameter, new and old values of the parameter should be compared with the precision with which they will be written, not their internal precision.

A pair of **q** and **Q** operators is commonly used to bracket a sequence of operators that uses a non-default clipping path. The **q** operator saves the default clipping path, and the **Q** operator discards the clipping path when it is no longer needed. However, the **q** and **Q** operators save and restore the entire graphics state, not just the clipping path. To avoid unnecessarily setting all graphics state parameters to achieve a known state after a **Q** operator, an application that produces PDF files may wish to maintain its own graphics state stack, mimicking the PDF graphics state stack. This enables the application to determine the values of all graphics state parameters at all times, and only write operators to change graphics state parameters that do not have the desired value after the **Q** operator.

10.10 Don’t draw outside the crop box

Objects entirely outside the crop box do not appear on screen or on the final printout. Nevertheless, if such objects are present in a PDF file, each time the page is drawn, time is spent determining if any portion of them is visible. Simply omit any objects that are entirely outside of the crop box, instead of relying on clipping to keep them from being drawn.

10.11 Consider target device resolution

When producing a PDF file, it is extremely important to consider the device that is the primary target of the document contained in the file. A number of decisions may be made differently depending on whether the document will be primarily viewed on a low-resolution device such as a computer screen or printed to an extremely high-resolution device such as an imagesetter.

If the primary target of the document is a computer screen, users are generally most interested in small file sizes and fast display, and are willing to accept somewhat reduced resolution in exchange for those. If, on the other hand, the primary target is a 1200-dpi imagesetter, file size and drawing time are not as important as obtaining the highest quality possible.

PDF, like the PostScript language, allows graphics objects to be drawn at an arbitrary size and scaled to the desired size. It is often convenient to design objects at a standard size and scale them for a particular situation. Greatly reducing the size of an object, however, can result in unnecessary detail and slow drawing. Choose a level of detail that is appropriate for both monitors and common printer resolutions. In some cases it may be appropriate to replace a complex element of a page with an equivalent image.

Decisions related to the target device primarily affect text, images, and blends. They are discussed further in the following chapters.

10.12 Share resources

Typically, many pages of a document share the same set of fonts. A PDF file will be smaller, display faster, and use less memory if the page's Resources dictionaries refer to the same Font objects. Similarly, if multiple fonts use the same custom encoding, one Encoding object should be shared. The same holds true for ProcSets—if multiple pages require the same combination of ProcSets, they should refer to the same ProcSet array.

10.13 Store common Page attributes in the Pages object

Several Page attributes need not be specified directly in the Page object, but can be inherited from a parent Pages object. Attributes that are the same for all pages in a document may be written once in the root Pages object. If a particular page has a different value, it can directly specify that value and override its parent's value. For example, all pages except one in a document might have the same media box. This value can be stored in the root Pages object, and the media box for the odd-size page can be specified directly in its Page object.

10.14 Use strings for named destinations

PDF 1.2

Named destinations (see page 95) can be represented with names stored in the Catalog's **Dests** dictionary, or with strings stored in the **Dests** name tree in the Catalog's **Names** dictionary. For a small number of named destinations, either method is acceptable, and for compatibility with PDF 1.0 and 1.1, names are required. But applications that generate many named destinations should use the PDF 1.2 feature of storing these as strings, where there are essentially no implementation limits.

CHAPTER 11

Optimizing Text

Most text optimizations relate to using appropriate operators and taking advantage of the automatic line, character, and word spacing operators supported by PDF. A few optimizations relate to searching.

11.1 Don’t produce unnecessary text objects

A PDF viewer application initializes the text environment at the beginning of each text object, and this initialization takes some time. Minimizing the number of text objects used reduces this overhead and reduces file size.

It is not necessary to end one text object and begin another whenever the text matrix is changed using the **Tm** operator. Instead, the text matrix can be changed inside the text object. For example, to create a text object containing several lines of text at various rotations, the following text object could be used:

Example 11.1 *Changing the text matrix inside a text object*

```
BT
/F13 24 Tf
200 100 Td
(Horizontal text) Tj
0.866 0.5 -0.5 0.866 186 150 Tm
(Text rotated 30 degrees counterclockwise) Tj
0.5 0.866 -0.866 0.5 150 186 Tm
(Text rotated 60 degrees counterclockwise) Tj
0 1 -1 0 100 200 Tm
(Text rotated 90 degrees counterclockwise) Tj
ET
```

This sequence draws the text in the font whose name in the current Resources dictionary is **F13**, at a size of 24 points. Keep in mind that the matrix specified using the **Tm** operator replaces the text matrix; it is not concatenated onto the text matrix.

Similarly, font and most other graphics state parameters can change inside a text object. There is one exception—if one of the clipping text-rendering modes is used, the text object must end before changing the text-rendering mode again.

11.2 Use automatic leading

Several of the text string operators make use of the text leading setting to position the drawing point at the start of the next line of text. This makes generating multiple lines of text easy and compact. Use automatic leading whenever possible. The **'** and **"** operators automatically move to the next line of text, as defined by the leading, and the **T*** operator can be used to manually move to the next line of text. Define leading using either the **TD** or **TL** operators.

Note *Don't use the* **TD** *or* **TL** *operator unless you use a text operator that has automatic leading.*

For example, the text object in Example 11.2 can be more efficiently written using automatic leading and the **'** operator as in Example 11.3.

Example 11.2 *Multiple lines of text without automatic leading*

```
BT
/F13 12 Tf
200 400 Td
(First line of text) Tj
0 -14 Td
(Second line of text) Tj
0 -14 Td
(Third line of text) Tj
0 -14 Td
(Fourth line of text) Tj
ET
```

Example 11.3 *Multiple lines of text using automatic leading*

```
BT
/F13 12 Tf
200 414 Td
14 TL
(First line of text) '
(Second line of text) '
(Third line of text) '
(Fourth line of text) '
ET
```

Note in Example 11.3 that the initial point has been offset vertically by one line. This is because the **'** operator moves to the next line before drawing the text.

If it is not possible to use either the **'** or **"** operators to draw a line of text (for example, because the **TJ** operator is used to adjust spacing between particular characters within the line), you can still use the **T*** operator, which advances the point to the beginning of the next line, using the current leading. For example, the text object in Example 11.4 can be more efficiently written using automatic leading and the **T*** operator, as in Example 11.5.

Example 11.4 **TJ** *operator without automatic leading*

```
BT
/F13 12 Tf
200 700 Td
[(First line) 100 ( of text)] TJ
0 -14 Td
[(Second line) 50 ( of text)] TJ
0 -14 Td
[(Third line) 40 ( of text)] TJ
0 -14 Td
[(Fourth line) 50 ( of text)] TJ
ET
```

Example 11.5 *Use of the* **T*** *operator*

```
BT
/F13 12 Tf
200 700 Td
14 TL
[(First line) 100 ( of text)] TJ
T*
[(Second line) 50 ( of text)] TJ
T*
[(Third line) 40 ( of text)] TJ
T*
[(Fourth line) 50 ( of text)] TJ
ET
```

Finally, you can set the leading in either of two ways. The **TL** operator sets the leading directly, while the **TD** operator sets the leading as a side effect of moving the line start position. The methods shown in Example 11.6 and Example 11.7 are equivalent.

Example 11.6 *Using the* **TL** *operator to set leading*

```
BT
/F13 12 Tf
200 500 Td
14 TL
[(First line) 100 ( of text)] TJ
T*
[(Second line) 50 ( of text)] TJ
T*
[(Third line) 40 ( of text)] TJ
T*
[(Fourth line) 50 ( of text)] TJ
ET
```

Example 11.7 *Using the* **TD** *operator to set leading*

```
BT
/F13 12 Tf
200 500 Td
[(First line) 100 ( of text)] TJ
0 -14 TD
[(Second line) 50 ( of text)] TJ
T*
[(Third line) 40 ( of text)] TJ
T*
[(Fourth line) 50 ( of text)] TJ
ET
```

When using the **TD** operator to set the leading, keep in mind that any horizontal component supplied as an operand to **TD** affects the movement of the drawing point, but not the leading. As a result, the commands

```
0 –14 TD
```

and

```
10 –14 TD
```

both set the leading to 14, although in the latter case the drawing point is ten units to the right of where it is in the former case.

11.3 Take advantage of text spacing operators

The **Tc** and **Tw** operators adjust the spacing between characters and the spacing between words, respectively. Use these operators to make constant adjustments on one or more lines of text. Example 11.8 shows a text object in which one half unit of space has been added between characters on a line and two units between words.

Example 11.8 *Character and word spacing using the* **Tc** *and* **Tw** *operators*

```
BT
/F13 12 Tf
200 514 Td
14 TL
.5 Tc
2 Tw
(Line of text) '
(Line of text) '
ET
```

Equivalently, the same two lines of text could be produced using the **"** operator instead of the **Tc**, **Tw**, and **'** operators, as shown in Example 11.9.

Example 11.9 *Character and word spacing using the* **"** *operator*

```
BT
/F13 12 Tf
200 514 Td
14 TL
2 .5 (Line of text) "
(Line of text) '
ET
```

Using the **"** operator is preferable if entire lines of text are being written, because it is more compact. If more than one text string operator is used to produce a line of text, the **"** operator can be used to position the first string of the line and **Tj** or **TJ** for subsequent strings. Remember that the **"** operator changes the character and word spacing settings for subsequent **Tj**, **TJ**, and **'** operators.

11.4 Don’t replace spaces between words

When deciding how to represent a line of text in a PDF file, keep in mind that text can be searched. In order to search text accurately, breaks between words must be found. For this reason, it is best to leave spaces in strings, instead of replacing them with an operator that moves the drawing point.

For example, text containing three words could be drawn by:

```
(A few words) Tj
```

Or, replacing the spaces between words with movements of the drawing point:

```
[(A) -300 (few) -300 (words)] TJ
```

The first method is preferred.

11.5 Use the appropriate operator to draw text

In most cases, a line of text can be represented in several ways. When deciding among the various methods, try to draw the line using as few operations as possible. Table 11.1 provides guidelines for selecting the appropriate text string operator.

Table 11.1 *Comparison of text string operators*

Use…	*When…*
'	Complete line of text can be drawn together No need for individual character spacings

"	Complete line of text can be drawn together Non-zero character or word spacings on each line No need for individual character spacings
Tj	Multiple text operators per line of text No need for individual character spacings
TJ	Individual character spacings needed

When laying out a line of text with non-default character spacings, such as kerned text, use the **TJ** operator rather than a series of pairs of **Tj** and **Td** operators. For example, both of the following lines produce the same output for the Helvetica Bold Oblique font at a size of 12 points:

```
(A f) Tj 15.64 0 Td (ew w) Tj 28.08 0 Td (ords) Tj
[(A f) 30 (ew w) 50 (ords)] TJ
```

The second method is preferred because it minimizes the size of the file and the number of text operators.

11.6 Use the appropriate operator to position text

The **TD**, **Td**, **Tm**, and **T*** operators each change the location at which subsequent text is drawn. Use each of these operators under different circumstances. Table 11.2 provides guidelines for selecting the appropriate text positioning operator.

Table 11.2 *Comparison of text positioning operators*

Use…	*When…*
Td	Changing only the text location
TD	Changing text location and leading
Tm	Rotating, scaling, or skewing text
T*	Moving to start of next line of text, as defined by the leading

11.7 Remove text clipping

After text has been used as a clipping path through one of the clipping text-rendering modes (4–7), the original clipping path must be restored. Restoration of the original clipping path is accomplished using the **q** and **Q** operators to save and subsequently restore the clipping path, respectively.

Neither **q** nor **Q** may appear inside a text object. Save the original clipping path using the **q** operator before beginning the text object in which a new clipping path is set. When you want to restore the original clipping path, the text object must be

ended using the **ET** operator. Then, use the **Q** operator to restore the original clipping path. Following this, another text object can be entered if more text is to be drawn.

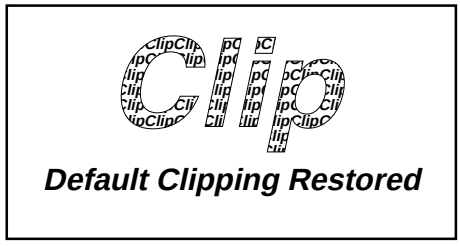
Example 11.1 illustrates the proper way to save and restore a clipping path when using one of the clipping text-rendering modes.

Example 11.1 *Restoring clipping path after using text as clipping path*

```
q
BT
/F13 48 Tf
200 414 Td
                                % Set clip path
0.25 w
5 Tr
(Clip) Tj
ET
BT
200 450 Td
/F13 6 Tf
0 Tr
6 TL
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
(ClipClipClipClipClipClipClipClip) '
ET
Q

BT
/F13 12 Tf
175 395 Td
(Default Clipping Restored) Tj
ET
```

Figure 11.1 shows the output produced by this example when F13 is Helvetica Bold Oblique. The presence of the words “Default Clipping Restored” at the bottom of the figure demonstrates that the clipping path has been restored to its previous value.

Figure 11.1 *Restoring clipping path after clipping to text*



11.8 Consider target device resolution

Although text in a PDF file is resolution-independent (unless a document contains bitmapped Type 3 fonts), there are still reasons to consider the resolution of the target device. Text positioning, in particular, may depend on the primary target device.

It is possible to individually position each character in a string using, for example, the **TJ** operator. This allows precise layout of text. However, adjusting the location of each character increases the size of a PDF file because the positioning must be specified by numbers that are otherwise not needed. In addition, drawing text is slower when each character is individually positioned. As mentioned in Section 10.11, “Consider target device resolution,” if the primary target is a low-resolution device such as a computer screen, producing a small file and one that draws quickly is generally more important than having extremely precise positioning. If the primary target is an imagesetter, extremely precise positioning is generally the primary concern.

As an example of the choices that can be made, suppose the positions of each character on a 60-character line are adjusted from their normal positions by an amount corresponding to 0.01 pixels on a 72 pixel per inch computer screen. The total adjustment across the entire line is just over half a pixel on the screen. If the document is primarily intended to be viewed on a computer screen, omitting the adjustments would make sense because such a small adjustment is invisible. The result would be a smaller file that can be drawn more quickly. On the other hand, the same adjustment corresponds to 10 pixels on a 1200 pixel per inch imagesetter. If the primary target is such an imagesetter, it may be worthwhile retaining the individual position adjustment. Note that precise text positioning is most important for justified text, where positioning errors are easily detected by users.

CHAPTER 12

Optimizing Graphics

12.1 Use the appropriate color-setting operator

Use `0 g` to set the fill color to black, rather than the equivalent, but longer, `0 0 0 rg` or `0 0 0 1 k`. Similarly, `0 G` should be used to set the stroke color to black instead of `0 0 0 RG` or `0 0 0 1 K`. In general, if a color contains equal color components, use either **g** or **G**, as appropriate. For example, use `.8 G` instead of `.8 .8 .8 RG`.

12.2 Defer path painting until necessary

When representing graphics in a PDF file, each path segment can be treated as a separate path, or a number of segments can be grouped together into a single path. Wherever possible, group segments together into a single path. This reduces the size of the file and improves drawing speed. However, a path should not contain more than approximately 1500 segments. For further information, see Appendix B of the *PostScript Language Reference Manual, Second Edition*.

Because a path can only be filled with a single color and stroked with a single color, line width, miter limit, and line cap style, a new path must be started whenever one or more of these values is changed.

As an illustration, Example 12.1 and Example 12.2 produce identical output, but the technique shown in Example 12.2 is preferred. Note that Example 12.2 still contains two paths. These paths cannot be combined, because they have different stroke colors.

Example 12.1 *Each path segment as a separate path*

```
.5 0 1 RG
100 100 m
100 200 l
S
100 200 m
200 200 l
S
200 200 m
200 100 l
S
200 100 m
100 100 l
S
0 .2 .4 RG
300 300 m
400 300 l
S
```

Example 12.2 *Grouping path segments into a single path*

```
.5 0 1 RG
100 100 m
100 200 l
200 200 l
200 100 l
s
0 .2 .4 RG
300 300 m
400 300 l
S
```

12.3 Take advantage of the closepath operator

The **h** (**closepath**) operator closes the current subpath by drawing a straight segment from the endpoint of the last segment drawn to the first point in the subpath. When the last segment in a path is straight, use the **h** operator to draw the final segment and close the path.

Two inefficient ways of closing a path commonly occur. The first, shown in Example 12.3, uses the **l** operator to draw the final segment, followed by the **h** operator to close the path.

Example 12.3 *Using redundant* **l** *and* **h** *operators to close a path inefficiently*

```
100 100 m
100 200 l
200 200 l
200 100 l
```

```
100 100 l
h
```

The second, shown in Example 12.4, uses the **l** operator to draw the final segment of the path.

Example 12.4 *Using the* **l** *operator to close a path inefficiently*

```
100 100 m
100 200 l
200 200 l
200 100 l
100 100 l
```

Example 12.5 shows the correct way of closing a path with a straight segment, using the **h** operator.

Example 12.5 *Taking advantage of the* **h** *operator to close a path*

```
100 100 m
100 200 l
200 200 l
200 100 l
h
```

If the **h** operator is not used, the appropriate line join will not occur at the juncture of the path's initial and final point.

12.4 Don't close a path more than once

Close a path only once. Don't use the **h** operator before a path painting operator that implicitly closes the path: the **n**, **b**, **f**, **f*** and **s** operators. In addition, the **h** operator should not be used with the **re** operator, because the **re** operator produces a path that is already closed.

For example, do not use a sequence as in Example 12.6, because the **s** operator automatically closes the path before stroking it.

Example 12.6 *Improperly closing a path: multiple path closing operators*

```
150 240.7 m
253.2 200 l
180.4 150 l
75.4 134.5 l
h
s
```

Instead, use the sequence:

Example 12.7 *Properly closing a path: single path closing operator*

```
150 240.7 m
253.2 200 l
180.4 150 l
75.4 134.5 l
s
```

12.5 Don’t draw zero-length lines

When generating graphics from a computer program, it is not uncommon to produce line segments of zero length. Such line segments produce no useful output and should be eliminated before the PDF file is written.

Line segments of zero length may arise when straight line segments are used to approximate a curve. Generally, the programmer wants to make sure that the approximation is close to the actual curve, and so takes small steps in approximating the curve. Occasionally the steps are small enough that they produce segments of zero length after the coordinates have been converted to the format in which they are written to the file. (See Section 10.8, “Limit the precision of real numbers.”)

Zero-length line segments may also be generated when making a two-dimensional projection of a three-dimensional object. Lines in the three-dimensional object that go directly into the page have zero length in the two-dimensional projection.

12.6 Make sure drawing is needed

When generating graphics from a computer program, test before writing the graphics to a PDF file to ensure that the graphics actually make new marks on the page and do not simply draw over marks already made.

Redundant graphics typically arise when making a two–dimensional projection of a three–dimensional object. It is possible to end up with several images that lie on top of one another after being projected.

12.7 Take advantage of rectangle and curve operators

Use the **re** operator to draw a rectangle, instead of the corresponding sequence of **m** and **l** operators.

Curves can be drawn in one of two ways; either by approximating the curve with a sequence of straight segments or by using the curve operators present in PDF. Although approximating curves using straight segments is easy, it typically results in a very large amount of data. Use the curve operators (**c**, **v**, **y**) to represent curves in PDF files. Doing so results in a smaller file that can be rendered more quickly.

An algorithm for automatically fitting an arbitrary set of points with a cubic Bézier curve, like those used by PDF, can be found in the series of books called *Graphics Gems*. The algorithm described in *Graphics Gems* (see [10], [17], and [19] in the Bibliography) begins by assuming the points supplied can be fit by a single cubic Bézier curve, with the two endpoints of the Bézier curve being the first and last data points, and the Bézier control points calculated from the approximate tangents at the endpoints of the supplied data. The algorithm minimizes the sum of the squares of the distances between the data points and the curve being fit by moving the control points. If a satisfactory fit cannot be obtained, the data points are separated into two groups at the point with the greatest distance between the curve being fit and the actual data point, and two separate Bézier curves are fit to the two sets of points. This fitting and splitting is repeated until a satisfactory fit is obtained.

12.8 Coalesce operations

Graphics generated by a computer program occasionally contain a group of operations that can be combined into a single operation. These can arise, for example, when a curve is approximated by a series of short straight segments. Significant sections of the curve being approximated may be effectively straight, but the approximation program typically does not realize this and continues to approximate the curve as a sequence of small line segments, instead of combining the collinear segments.

For example, the sequence shown in Example 12.8 contains a number of segments that should be combined. Specifically, the first four **l** operators simply draw one straight line segment and should be combined.

Example 12.8 *Portion of a path before coalescing operations*

```
100 100 m
100 101 l
100 102 l
100 103 l
100 104 l
101 105 l
```

The entire sequence can be replaced by the equivalent and more efficient sequence in Example 12.9.

Example 12.9 *Portion of a path after coalescing operations*

```
100 100 m
100 104 l
101 105 l
```

CHAPTER 13

Optimizing Images

Sampled images typically require more memory and take more time to process and draw than any other graphics object element of a page. By carefully choosing an appropriate resolution, number of bits per color component, and compression filter, it is possible to significantly enhance image performance.

13.1 Preprocess images

PDF provides operators that transform and clip images. These operators should be used with care. For example, performance often improves if rotation and skewing of an image is performed *before* the image is placed in the PDF file, rather than *by* the PDF viewer application. Similarly, if an image is clipped, it is best to reduce the image to the smallest dimensions possible before placing the image in the PDF file, perhaps eliminating the need for clipping.

13.2 Match image resolution to target device resolution

If a grayscale or color image will primarily be viewed on computer screens (which typically have resolutions between 70–100 pixels per inch) or printed on typical color and monochrome printers (which have resolutions of 300 dpi and default halftone screens of approximately 60 lines per inch), there is no point in producing the image at 300 samples per inch. Most of the information in the higher resolution image will never be seen, the image will contain at least nine times as much data as it needs to (90,000 samples per square inch versus a maximum of 10,000 samples per square inch), and will draw more slowly.

Monochrome images can be stored at higher resolutions of 200 to 300 dpi. This resolution can be achieved on typical printers.

13.3 Use the minimum number of bits per color component

The amount of data needed to represent an image increases as the number of bits per color component increases. This is very important to consider when deciding how many bits per component to use for an image.

If an image requires continuous colors, it might very well need to use 8 bits per color component. However, many graphs, plots, and other types of drawings do not require continuous tone reproduction and are completely satisfactory with a small number of bits per color component.

13.4 Take advantage of indexed color spaces

If an image contains a relatively small number of colors, indexed color spaces can be used to reduce the amount of data needed to represent the image. In an indexed color space, the number of bits needed to represent each sample in an image is determined by the total number of colors in the image rather than by the precision needed to specify a single color.

Most computers currently have displays that support a limited number of colors. For example, it is very common for color displays on the Macintosh computer to provide no more than 256 colors, and many computers running the Microsoft Windows environment provide only 16 colors. On such devices, little loss of image quality will occur if 24-bit color images are replaced by 8-bit indexed color images.

As an example of the compression possible using indexed color spaces, suppose an image contains 256 different colors. Each pixel's color can then be encoded using only 8 bits, regardless of whether the colors in the image are 8-bit grayscale, 24-bit RGB, or 32-bit CMYK. If the colors are 24-bit RGB, using an indexed color space instead of the RGB values would reduce the amount of data needed to represent the image by approximately a factor of three: 24 bits per pixel using an RGB color space versus 8 bits per pixel using an indexed color space. The reduction is not exactly three because the use of an indexed color space requires that a lookup table, containing the list of colors used in the image, be written to the file. For a large image, the size of this lookup table is insignificant compared to the image and can be ignored. For a small image, the size of the lookup table must be included in the calculation. The size of the lookup table can be calculated from the description of indexed color spaces in Section 7.10.7 on page 173.

13.5 Use the DeviceGray color space for monochrome images

For a bitmap (monochrome) image, use the **DeviceGray** color space instead of **DeviceRGB**, **DeviceCMYK**, or **Indexed** color space. In addition, the **BitsPerComponent** attribute for bitmap images should be 1. These settings significantly reduce the amount of data used to represent the image.

Using a different color space or a larger **BitsPerComponent** greatly increases the amount of image data. As an extreme example, a bitmap image represented using a **DeviceCMYK** color space with 8 bits per component contains 32 times as much data as necessary: four color components with 8 bits per component, instead of a single color component with 1 bit per component.

13.6 Use in-line images appropriately

In-line images occupy less disk space and memory than image resources. However, image resources give viewer applications more flexibility in managing memory—the data of an image resource can be read on demand, while an in-line image must be kept in memory together with the rest of a page's contents.

Implementation note

PDF Writer and the Acrobat Distiller application represent images with less than 4K of data as in-line images until a total of 32K of in-line data are present on a page. Once this limit is reached, subsequent images on that page are represented in-line only if they are smaller than 1K.

13.7 Don't compress in-line images unnecessarily

In-line images should not always be compressed and converted to ASCII. Specifically, in-line images should not be compressed if the Contents stream of the page on which the in-line image appears is itself compressed.

Because an in-line image is located completely within the Contents stream of the page, it is automatically passed through the compression and ASCII conversion filters specified for the page's Contents stream. The specification of an additional compression or ASCII conversion filter in the in-line image itself under these circumstances results in the in-line image being compressed and converted to ASCII twice. This does not result in additional compression and slows down the decoding of the image.

13.8 Choose the appropriate filters

The selection of filters for image streams can be confusing, although a few relatively simple rules can greatly simplify the task.

The order of filters specified when data is decoded must be the opposite of the order in which the filters were applied when the data was encoded. For example, if data is encoded first using LZW and then by ASCII base-85, during decoding the ASCII base-85 filter must be used before the LZW decoding filter. In a stream object, the decoding filters and the order in which they are applied are specified by the **Filter** key. The example would appear as:

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

As discussed in Section 2.3.2 on page 30, it may be necessary to ensure that a PDF file contains only 7-bit ASCII characters, so the binary data in the PDF file must be encoded with one of the two binary-to-ASCII conversion filters supported by PDF: ASCII hexadecimal and ASCII base-85. Between these two, the ASCII base-85 encoding, which is decoded by the **ASCII85Decode** filter, is preferred because it produces a much smaller expansion in the amount of data than ASCII hexadecimal encoding does.

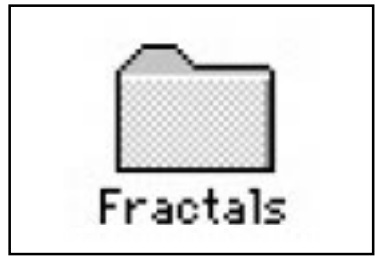
PDF supports several compression filters that reduce the size of data written into a PDF file. The compression filters can be broken down into two classes: lossless and lossy. A lossless filter is one in which the process of encoding and decoding results in no loss of information: the decoded data is bit-by-bit identical to the original data. For a lossy filter, the process of encoding and decoding results in a loss of information: the decoded data is not bit-by-bit identical to the original data. Lossy filters can be used when the resulting loss of information is not visually significant. The JPEG filter supported by PDF is a lossy filter.

JPEG encoding, which can be decoded by the **DCTDecode** filter, provides very significant compression of color and grayscale images, but because it is a lossy compression it is not appropriate in all circumstances. Screenshots, in particular, are often unacceptable when JPEG encoded. This happens because each pixel in a screenshot is usually significant, and the loss or alteration of just a few pixels can drastically alter the appearance of the screenshot.

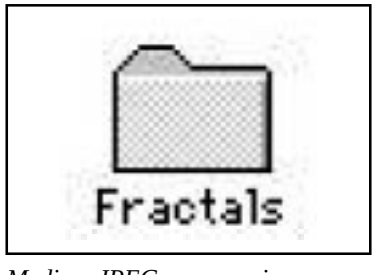
Figure 13.1 shows the effect of JPEG encoding on screenshots. The images shown in the figure are magnified by a factor of two to show the changes due to the compression. The 8×8 pixel blocks used in JPEG encoding appear clearly in the pattern inside the icon encoded using a high JPEG compression. The definitions of high, medium, and low JPEG compression are those used by the Acrobat Distiller program. The amount of data in the image from which the figure is taken is: 153,078 bytes with no JPEG encoding, 28,396 bytes with low compression JPEG encoding, 16,944 bytes with medium compression JPEG encoding, and 10,679 bytes with high compression JPEG encoding. All these sizes are for the data after it has been ASCII base-85 encoded.

Figure 13.1 *Effect of JPEG encoding on a screenshot*

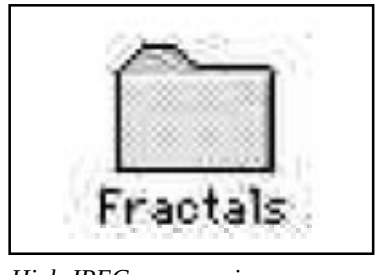
No JPEG compression



Low JPEG compression



Medium JPEG compression



High JPEG compression

Unlike screenshots, the effect of JPEG encoding on continuous-tone images is typically acceptable, particularly when high compression is not demanded. Figure 13.2 illustrates the effect. The image shown in the figure has been magnified by a

factor of two to make the effect of JPEG encoding more readily observable. The version obtained using high compression clearly shows the 8×8 pixel blocks used in JPEG encoding. As in the previous example, the definitions of high, medium, and low JPEG compression are those used by the Acrobat Distiller program, and the sizes shown are for the data after it has been ASCII base-85 encoded.

Figure 13.2 *Effect of JPEG encoding on a continuous-tone image*

No JPEG compression; 20,707 bytes

Low JPEG compression; 7,717 bytes

Medium JPEG compression; 3,470 bytes

High JPEG compression; 1,997 bytes

In addition to JPEG, PDF supports several lossless compression filters that may be used for images. Guidelines for selecting among them are summarized in Table 13.1.

Table 13.1 *Comparison of compression filters for images*

Use…	*When…*
DCTDecode	Image is grayscale or color Decompressed image doesn't need to be bit-by-bit identical to original image

CCITTFaxDecode	Image is monochrome (bitmap) Group 4 encoding should be used unless the application generating the file does not support Group 4 encoding
RunLengthDecode	Image contains many groups of identical bytes, such as an 8-bit grayscale image with many areas of same gray level. Should rarely be used
LZWDecode	Images that cannot use **DCTDecode** and that do not compress well using either CCITT or run length encoding

13.9 Use predefined spot functions

PDF 1.2

A Type-1 halftone screen (see page 192) includes a spot function. In PDF, this is represented with a Function resource; in PDF 1.2, there are *named* spot functions and *sampled* spot functions. The named spot functions are predefined, so an application can implement them directly in code. When printing a PDF file to a PostScript printer or a Postscript language file, an application can insert the predefined PostScript code for each named function.

Spot functions that do not correspond to any of the predefined set are sampled, and linear interpolation is used to approximate the function. For most functions, this is quite adequate, and the error introduced by sampling and interpolating is quite small.

Spot functions, however, are very sensitive to small errors. They are not used for their values, but rather for their *relative* values. In a 5-by-5 halftone cell, for example, the coordinates of each cell are passed to the spot function. The cell with the smallest spot value is darkened first, and the cell with the greatest value is darkened last. If the values for two cells are interchanged because of an error in the approximation, no matter how small, then they darken in reverse order. This can produce visible differences in the spot. This is an issue only for high-resolution, professional publishing applications.

Sampled functions require interpolation, which takes more time than executing a function directly. The sample values themselves consume space in the PDF file and the PostScript file that is produced by the viewer. The PostScript file will also include an interpolation function.

PDF 1.2 predefines several spot functions, intended to cover all of the common cases of spot functions found in desktop publishing and graphics applications. An application that uses a spot function that is very close to one of these may benefit from using the predefined function instead.

Implementation note

The Acrobat 3.0 Distiller samples spot functions over a 33-by-33 grid, producing about 1K of data (before compression). The Distiller attempts to match the PostScript language source code of a spot function with the code for the predefined functions. If it fails to find a match with the source code, it samples the function

and compares the results with sampled values of the predefined functions, so that a spot function that computes the same results as, say, InvertedDoubleDot, but uses different code to do so, will be treated as if the source code had matched.

CHAPTER 14

Clipping and Blends

Clipping restricts the areas on a page where marks can be made. It is similar to using a stencil when painting or airbrushing. A stencil with one or more holes in it is placed on a page. As long as the stencil remains in place, paint only reaches the page through the holes in the stencil. After the stencil is removed, paint can again be applied anywhere on the page. More than one stencil may be used in the production of a single page, and if a second stencil is added before the first one is removed, paint only reaches the page where there are holes in *both* stencils.

Similarly, in producing a PDF page, one or more clipping paths may be used. If a clipping path is not removed before a second clipping path is applied, the resulting clipping path is the intersection of the two paths.

Clipping paths may be specified in two distinct ways: paths and text. These provide clipping that affects all subsequent marking operations until the clipping path is explicitly changed. An example of each type of clipping is provided in the following sections.

Note *Whenever a clipping path is no longer needed, the default clipping path should be restored, as described in Section 10.4, "Remove unnecessary clipping paths."*

Image masks do not provide clipping as paths and text do, but they can be thought of as specifying a bitmap clipping template that is placed on the page, painted with a color, and then immediately removed. The differences between images and image masks are discussed below.

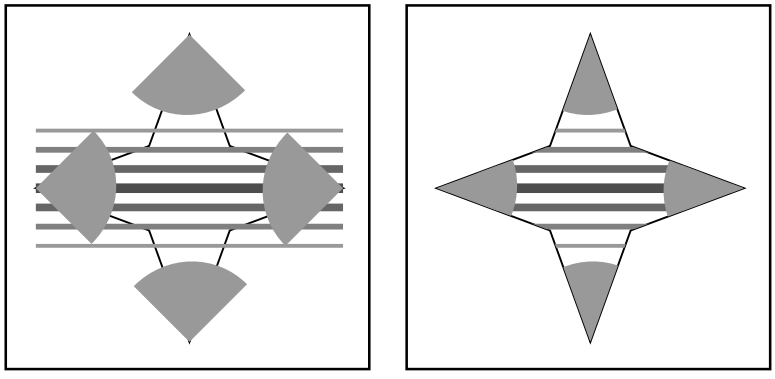
Often, page descriptions contain *blends*, smooth changes of color used as a background or to fill an object. Because blends typically fill objects, and clipping is needed in order to accomplish this, blends are also described in this chapter. A useful way to produce blends using images is provided.

14.1 Clipping to a path

As described in Section 8.6.3, "Path clipping operators," the **W** and **W*** operators can make any path a clipping path. To do this, insert the operator between the path segment operators and one of the path painting operators described in Section 8.6.2, "Path painting operators."

Figure 14.1 shows the effect of clipping to a region in the shape of a four-pointed star. In the figure, the graphics are shown with and without the star as a clipping path. To draw the figure, the star is first stroked and set to be the current clipping path. A series of lines is then drawn through the star, and the points of the star are filled using arcs.

Figure 14.1 *Clipping to a path*



Without clipping to star *With clipping to star*

Note *When a path is stroked and used as the current clipping path, remember that the stroke extends half the line width on each side of the path, while subsequent drawing is clipped to the path itself. Because of this, subsequent clipped drawing operations can draw over the "inner half" of the stroke.*

The PDF operations needed to produce this output are shown in Example 14.1. The star is first drawn using a series of **l** operators. It is set to be a clipping path using the **W** operator and stroked using the **s** operator. Next, a series of lines is drawn across the star using the **m** and **l** operators. The lines have different gray levels (set by the **G** operator) and line widths (set by the **w** operator). Because each line has a different width and color, each must be stroked (using the **S** operator) individually. To generate the non-clipped portion of the figure, the only change made to the PDF files was to remove the **W** operator.

Example 14.1 *Clipping to a path*

```
                                % Draw outline of star
391 392 m
370 450 l
311 472 l
370 494 l
391 552 l
412 494 l
471 472 l
412 450 l
W
s
                                                % Draw lines
.6 G 2 w 311 502 m 471 502 l S
.5 G 3 w 311 492 m 471 492 l S
```

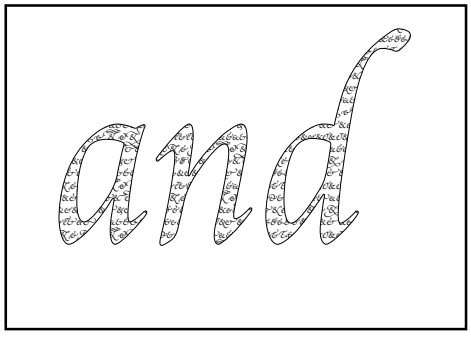
```
.4 G 4 w 311 482 m 471 482 l S
.3 G 5 w 311 472 m 471 472 l S
.4 G 4 w 311 462 m 471 462 l S
.5 G 3 w 311 452 m 471 452 l S
.6 G 2 w 311 442 m 471 442 l S
          % Draw and fill circles on endpoints
0.6 g
340 443 m
357 460 357 486 341 502 c
311 472 l
f
421 422 m
405 438 379 438 362 421 c
391 392 l
f
442 501 m
425 484 425 458 441 442 c
471 472 l
f
361 522 m
377 506 403 506 420 523 c
391 552 l
f
```

14.2 Clipping to text

Several of the text rendering modes described in Section 8.7.1.7, “Text rendering mode” allow text to be used as a clipping path. In particular, modes 4 through 7 can be used to clip subsequent drawing to the shapes of one or more characters.

Figure 14.2 shows the word “and” used as a clipping path. The word is first drawn as stroked and clipped text. Following this, a series of lines containing various ampersands is drawn on top of the word. Only those ampersands contained inside the clipping path defined by the word are visible. The font used for the word “and” is Poetica® Chancery III. The font used for the ampersands is Poetica Ampersands.

Figure 14.2 *Using text as a clipping path*



Example 14.2 shows the page description used to produce Figure 14.2. In the example, the font named F6 is Poetica Ampersands and the font named F24 is Poetica Chancery III.

Example 14.2 *Using text as a clipping path*

```
BT
100 500 Td
             % Draw the word "and", stroke it,
              % and use it as a clipping path.
/F24 144 Tf
0.25 w
5 Tr
(and) Tj
ET
BT
              % Select Poetica Ampersands font
/F6 6 Tf
100 615 Td
0 Tr
6 TL
                    % Draw lines of ampersands
(aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuU) '
(vVwWxXyYzZ123456aAbBcCdDeEfFgGhHiIjJkKlLmMnNoO) '
(pPqQrRsStTuUvVwWxXyYzZ123456aAbBcCdDeEfFgGhH)'
(jJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ123456) '
(aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuU) '
(vVwWxXyYzZ123456aAbBcCdDeEfFgGhHiIjJkKlLmMnNoO) '
(pPqQrRsStTuUvVwWxXyYzZ123456aAbBcCdDeEfFgGhH) '
(jJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ123456) '
(aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuU) '
(vVwWxXyYzZ123456aAbBcCdDeEfFgGhHiIjJkKlLmMnNoO) '
(pPqQrRsStTuUvVwWxXyYzZ123456aAbBcCdDeEfFgGhH) '
(jJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ123456) '
(aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuU) '
(vVwWxXyYzZ123456aAbBcCdDeEfFgGhHiIjJkKlLmMnNoO) '
(pPqQrRsStTuUvVwWxXyYzZ123456aAbBcCdDeEfFgGhH) '
(jJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ123456) '
(aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuU) '
(vVwWxXyYzZ123456aAbBcCdDeEfFgGhHiIjJkKlLmMnNoO) '
(pPqQrRsStTuUvVwWxXyYzZ123456aAbBcCdDeEfFgGhH) '
(jJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ123456) '
ET
```

After beginning a text object by using the **BT** operator, the point at which text will be drawn is set using the **Td** operator. Following this, the font (named **F24**) and the size (144 points) are set using the **Tf** operator, the linewidth for the stroke is set to 0.25 units using the **w** operator, and the stroked clipping text rendering mode (mode 5) is selected using the **Tr** operator. The word “and” is then drawn using the **Tj** operator. Next, the text object is ended using the **ET** operator. This is necessary

in order to draw text using a different rendering mode. Following this, another text object is started, the ampersand font (named F6) and the size (6 points) are set, the position where text will be drawn is moved, the filled text rendering mode (mode 0) is selected, and the line leading is set to 6 points using the **TL** operator. Finally, the ampersands are drawn by a series of **'** operators, and the text object ends.

14.3 Image masks

Although image masks do not provide clipping as described above, they can be thought of as operating as follows: a bitmap image defines the clipping path, where 1s and 0s are considered to be holes and masks. The rectangle containing the bitmap is painted with the current fill color. Immediately following this, the bitmap-derived clipping path is removed.

Image masks differ from images in two ways. First, when an image is drawn, all pixels in the rectangle of the image are painted. In an image mask, only the pixels under holes in the mask are painted; all other pixels are left unchanged. Second, the colors in which an image is painted are encoded inside the image itself, while an image mask is painted using the current fill color at the time the image mask is drawn. Because of this, an image mask may appear in different colors each time it is drawn.

As described in Section 7.11, “XObjects,” the structure of an image mask differs from that of an image in several ways. First, an image mask must have only one bit per color component. Second, an image mask must not contain a color space specification, while an image must. Third, the image mask dictionary must contain the **ImageMask** key with a value of *true*. For both images and image masks, the array specified as the value of the **Decode** key in the image can be used to choose whether bits containing 1s or bits containing 0s are considered to be set.

Figure 14.3 shows examples of images and image masks. The examples also illustrate how the decode array can be used to invert the image.

Example 14.3 shows the relevant sections from the PDF file used to produce the

Figure 14.3 *Images and image masks*

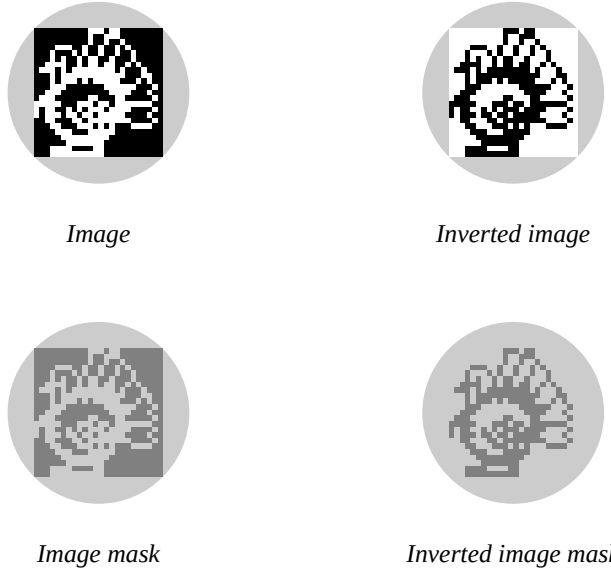


figure. Because the only difference between the PDF files used to draw each of the four examples is in the image object itself, all the drawing operations are common. The `0.6 g` operation appearing just before the image or image mask is drawn has an effect only when the object being drawn is an image mask, not an image. The example shows the operations used to draw the image mask portion of the figure. To produce the image portion of the figure, the line

```
/ImageMask true
```

was replaced with the line

```
/ColorSpace /DeviceGray
```

For the inverted image and inverted image mask, the line

```
/Decode [1 0]
```

was added to the dictionary of the image or image mask.

Example 14.3 *Images and image masks*

```
3 0 obj
<<
/Type /Page
/Parent 4 0 R
/MediaBox [53 470 198 616]
```

```
/Resources
<<
/XObject << /Im0 60 0 R >>
/ProcSet [/PDF /ImageC]
>>
/Contents 23 0 R
>>
endobj
23 0 obj
<< /Length 205 >>
stream
                  % Draw a circle and fill it
0.8 g
126 472 m
165 472 197 504 197 543 c
197 582 165 614 126 614 c
87 614 55 582 55 543 c
55 504 87 472 126 472 c
f
                         % Draw image or mask
q
100 0 0 100 76 493.2 cm
.6 g
/Im0 Do
Q
endstream
endobj
60 0 obj
<< /Type /XObject
/Subtype /Image
/Name /Im0
/Width 24
/Height 23
/BitsPerComponent 1
/Filter /ASCIIHexDecode
/Length 162
/ImageMask true
>>
stream
003b00 002700 002480 0e4940
114920 14b220 3cb650 75fe88
17ff8c 175f14 1c07e2 3803c4
703182 f8edfc b2bbc2 bb6f84
31bfc2 18ea3c 0e3e00 07fc00
03f800 1e1800 1ff800>
endstream
endobj
```

14.4 Blends

Several approaches may be used to produce blends. One alternative is to draw path segments such as rectangles, lines, and arcs adjacent to each other, each having a slightly different color. This method can result in large files and is slow to draw. Using images is often a much better method for producing blends.

Blends made using images usually occupy much less space in a PDF file. Images also have the advantage that they can be filled with arbitrary sequences of colors to provide arbitrary blends, and they can be easily stretched, rotated, and skewed in order to provide a variety of blend effects from a single image. In addition, the colors in an image can vary arbitrarily from sample to sample, allowing the production of effects that are difficult or impossible using path segment operators.

Using an image as a blend involves several steps:

1. Create the image containing the blend.
2. Draw the shape to be filled with the blend and make it the current clipping path.
3. Scale and translate the image using the **cm** operator so that it completely fills the shape.
4. Draw the image using the **Do** operator.
5. Remove the clipping path created in Step 2 so that any subsequent drawing is not restricted to the shape of the object that was filled with the blend.

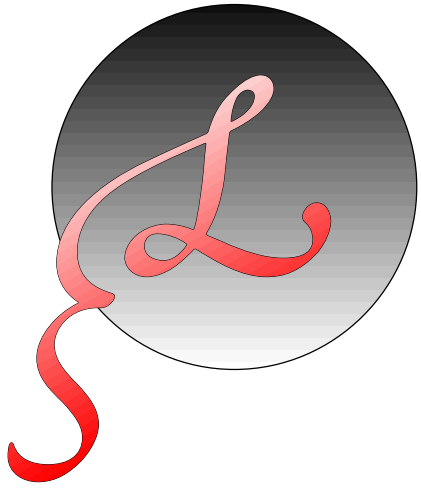
To create a linear blend in which the color inside an object varies smoothly from top to bottom, only a one-sample-wide image is needed, with as many rows in the image as there are to be steps in the blend. Each sample in the image is given the color of the corresponding band in the blend. For example, to create a four-step grayscale blend that goes from medium gray to black, create an image with a **Width** of 1, a **Height** of 4, and a **ColorSpace** of **DeviceGray**. Set **BitsPerComponent** as needed. Suppose you set it to 8. The image data contains the colors to be used in the blend. In this example, you might set them to 00, 20, 40, and 60 hexadecimal.

Now that this image has been created, it can be rotated to provide other blends. For example, to obtain a four-step horizontal blend instead of a vertical blend, the image need only be rotated by 90 degrees by setting the appropriate matrix (using the **cm** operator) before drawing the image.

Figure 14.4 illustrates the use of an image to produce a linear blend. The example consists of a circle, stroked and used as a clipping path for a 32-step vertical gray blend. A second blend is used inside the letter. This 27-step blend runs from light pink at the top to deep red at the bottom. The blend is tilted 30 degrees, so that the lines of constant color are approximately parallel to the stem coming off the left side of the letter L.

Note *The example blends in this chapter use a relatively small number of steps. This is done only to minimize the size of the examples. Blends of 256 steps, which generally provide smooth blends, can be used without a significant performance degradation.*

Figure 14.4 *Using an image to produce a linear blend*



The relevant sections from the PDF file used to produce the figure are shown in Example 14.4. The example is explained in the following paragraphs.

Example 14.4 *Using images as blends*

```
3 0 obj
<<
/Type /Page
/Parent 4 0 R
/MediaBox [0 0 612 792]
/Resources << /Font << /F39 7 0 R >>
/XObject << /Im0 10 0 R /Im1 11 0 R >>
/ProcSet [/PDF /Text /ImageC] >>
/Contents 6 0 R
>>
endobj
6 0 obj
<< /Length 383 >>
stream
      % Draw circle, use it as a clipping path
q
126 472 m
165 472 197 504 197 543 c
197 582 165 614 126 614 c
```

```
87 614 55 582 55 543 c
55 504 87 472 126 472 c
W
s
                    % Draw image inside circle
-150 0 0 -150 200 620 cm
/Im0 Do
Q
        % Draw character, stroke it and use it
                          % as a clipping path
q
BT
85 510 Td
/F39 144 Tf
0.25 w
5 Tr
(L) Tj
ET
                      % Draw image inside text
147 85 -50 86.7 45 420 cm
/Im1 Do
Q
endstream
endobj
10 0 obj
<< /Type /XObject
/Subtype /Image
/Name /Im0
/Width 1
/Height 32
/BitsPerComponent 8
/ColorSpace /DeviceGray
/Filter /ASCIIHexDecode
/Length 97 >>
stream
ff f8 ef e8 df d8 cf c8 bf b8 af a8 9f 98 8f 88 7f 78
6f 68 5f 58 4f 48 3f 38 2f 28 1f 18 0f 08>
endstream
endobj
11 0 obj
<< /Type /XObject
/Subtype /Image
/Name /Im1
/Width 1
/Height 27
/BitsPerComponent 8
/ColorSpace /DeviceRGB
/Filter /ASCIIHexDecode
```

```
/Length 190 >>
stream
ffd0d0 ffc8c8 ffc0c0 ffb8b8 ffb0b0 ffa8a8 ffa0a0
ff9898 ff9090 ff8888 ff8080 ff7878 ff7070 ff6868
ff6060 ff5858 ff5050 ff4848 ff4040 ff3838 ff3030
ff2828 ff2020 ff1818 ff1010 ff0808 ff0000>
endstream
endobj
```

Object number 3 is the Page object, and is included to show the Resources dictionary, containing the mapping between image and font names used in the page contents, and the objects which are the fonts and images. In addition, the dictionary contains a list of the procsets needed to print this page.

Object number 6 is the page contents. The graphics state is first saved using the **q** operator, in order to be able to restore the original clipping path after drawing the circle and filling it with a blend. Next, the circle is drawn using four Bézier curve segments (the **c** operators), set to be the clipping path using the **W** operator, and stroked using the **s** operator. Following this, the **cm** operator is used to translate and scale the image so that it fills the circle, and the gray blend (named Im0) is drawn using the **Do** operator. Next, the original clipping path is restored using the **Q** operator, and this state saved again, for restoration after using a clipping mode to fill the text.

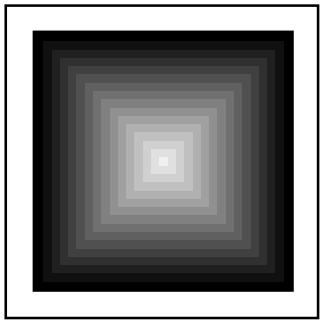
The text is positioned using the **Td** operator, and the font (named **F39**, which in the example is Poetica Initial Swash Capitals) and size (144 points) are set using the **Tf** operator. The font object and other related objects are not included in the section shown from the example file. The text rendering mode is set to stroke the text and use it as the clipping path (mode 5) using the **Tr** operator. The text is drawn using the **Tj** operator, and the text object ended. The transformation matrix is again set to scale the image that is to be used as the blend filling the letter. In addition to scaling the image, the matrix used produces a 30-degree rotation to provide a diagonal blend. The image used as the colored blend (named Im1) is drawn, and the initial graphics state restored.

Because the drawing and filling of the text are the last operations in the contents of this particular page, it is not necessary to save the graphics state before entering the text object and to restore the graphics state after drawing the blend. The saving and restoring is included in this example as a reminder that the graphics state must be restored before any subsequent drawing.

To produce smoother transitions, at the expense of possibly slower rendering, you can specify the **Interpolate** key in the image. If **Interpolate** is true, rendering devices that support image interpolation attempt to make a smooth transition between adjacent sample values.

Images may be used to produce other blends, such as the square blend shown in Figure 14.5. The blend shown in the figure is a 16-step grayscale blend. Radial blends, in which the bands of constant color are circles, and other arbitrarily complicated blends can also be produced using images.

Figure 14.5 *Using an image to produce a square blend*



The image used to produce the blend is shown in Example 14.5. It is a 31×31 sample grayscale image, with 8 bits per sample.

Example 14.5 *Image used to produce a grayscale square blend*

```
<< /Type /XObject
/Subtype /Image
/Name /Im0
/Width 31 /Height 31
/BitsPerComponent 8
/ColorSpace /DeviceGray
/Filter /ASCIIHexDecode
/Length 1954 >>
stream
000000000000000000000000000000000000000000000000000000000000000
001010101010101010101010101010101010101010101010101010101010101000
001020202020202020202020202020202020202020202020202020202020201000
001020303030303030303030303030303030303030303030303030303030201000
001020304040404040404040404040404040404040404040404040404030201000
001020304050505050505050505050505050505050505050505040304030201000
001020304050606060606060606060606060606060606060606060504030201000
001020304050607070707070707070707070707070707070706050403020100
001020304050607080808080808080808080808080808080807060504030201000
001020304050607080909090909090909090909090909080706050403020100
001020304050607080909090909090909090909090909080706050403020100
001020304050607080a0a0a0a0a0a0a0a0a0a0a0a0a0a0908070605040302010
001020304050607080a0b0b0b0b0b0b0b0b0b0b0b0a0908070605040302010
001020304050607080a0b0c0c0c0c0c0c0c0b0b0a0908070605040302010
001020304050607080a0b0c0d0d0d0d0d0c0b0b0a0908070605040302010
001020304050607080a0b0c0d0e0e0e0d0c0b0b0a0908070605040302010
001020304050607080a0b0c0d0e0f0e0d0c0b0b0a0908070605040302010
001020304050607080a0b0c0d0e0e0e0d0c0b0b0a0908070605040302010
001020304050607080a0b0c0d0d0d0d0d0c0b0b0a0908070605040302010
001020304050607080a0b0c0c0c0c0c0c0c0b0b0a0908070605040302010
001020304050607080a0b0b0b0b0b0b0b0b0b0b0a0908070605040302010
001020304050607080a0a0a0a0a0a0a0a0a0a0a0a0908070605040302010
001020304050607080909090909090909090909090909080706050403020100
```

```
00102030405060708080808080808080808080808070605040302010 00
00102030405060707070707070707070707070707070605040302010 00
00102030405060606060606060606060606060606060605040302010 00
00102030405050505050505050505050505050505050504030201000
00102030404040404040404040404040404040404040404030201000
00102030303030303030303030303030303030303030303030201000
00102020202020202020202020202020202020202020202020201000
00101010101010101010101010101010101010101010101010101000
00000000000000000000000000000000000000000000000000000000>
endstream
```

APPENDIX A

Example PDF Files

A.1 Minimal PDF file

Although the PDF file shown in this example does not draw anything, it is almost the minimum PDF file possible. It is not strictly the minimum acceptable file because it contains an Outlines object, a Contents object, and a Resources dictionary with a ProcSet resource. These objects were included to make this file useful as a starting point for developing test files. The objects present in this file are listed in Table A.1.

Note *When using this file as a starting point for creating other files, remember to update the ProcSet resource as needed (see Section 7.5, "ProcSets"). Also, remember that the cross-reference table entries may need to have a trailing blank (see Section 5.4, "Cross-reference table.")*

Table A.1 *Objects in empty example*

Object number	*Object type*
1	Catalog
2	Outlines
3	Pages
4	Page
5	Contents
6	ProcSet array

Example A.1 *Minimal PDF file*

```
%PDF-1.0
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
/Outlines 2 0 R
>>
endobj
2 0 obj
<<
/Type /Outlines
/Count 0
>>
endobj
3 0 obj
<<
/Type /Pages
/Count 1
/Kids [4 0 R]
>>
endobj
4 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /ProcSet 6 0 R >>
/MediaBox [0 0 612 792]
/Contents 5 0 R
>>
endobj
5 0 obj
<< /Length 35 >>
stream
%place page marking operators here
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
```

A.2 Simple text string

This PDF file is the classic “Hello World.” It displays a single line of text containing that string. The string is displayed in 24-point Helvetica. Because Helvetica is one of the base 14 fonts, no font descriptor is needed. This example illustrates the use of fonts and several text-related PDF operators. The objects contained in the file are listed in Table A.2.

Table A.2 *Objects in “Hello World” example*

Object number	*Object type*
1	Catalog
2	Outlines
3	Pages
4	Page
5	Contents
6	ProcSet array
7	Font (Type 1 font)

Example A.2 *PDF file for simple text example*

```
%PDF-1.0
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
/Outlines 2 0 R
>>
endobj
2 0 obj
<<
/Type /Outlines
/Count 0
>>
endobj
```

```
3 0 obj
<<
/Type /Pages
/Count 1
/Kids [4 0 R]
>>
endobj
4 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /Font << /F1 7 0 R >> /ProcSet 6 0 R >>
/MediaBox [0 0 612 792]
/Contents 5 0 R
>>
endobj
5 0 obj
<< /Length 44 >>
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
0000000322 00000 n
0000000415 00000 n
0000000445 00000 n
trailer
```

```
<<
/Size 8
/Root 1 0 R
>>
startxref
553
%%EOF
```

A.3 Simple graphics

This PDF file draws a thin black line segment, a thick black dashed line segment, a filled and stroked rectangle, and a filled and stroked Bézier curve. The file contains comments showing the various operations. The objects present in this file are listed in Table A.3.

Note *The byte-addresses in the cross-reference table are not necessarily accurate.*

Table A.3 *Objects in graphics example*

Object number	*Object type*
1	Catalog
2	Outlines
3	Pages
4	Page
5	Contents
6	ProcSets

Example A.3 *PDF file for simple graphics example*

```
%PDF-1.0
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
/Outlines 2 0 R
>>
endobj
2 0 obj
<<
/Type /Outlines
/Count 0
>>
endobj
3 0 obj
<<
/Type /Pages
```

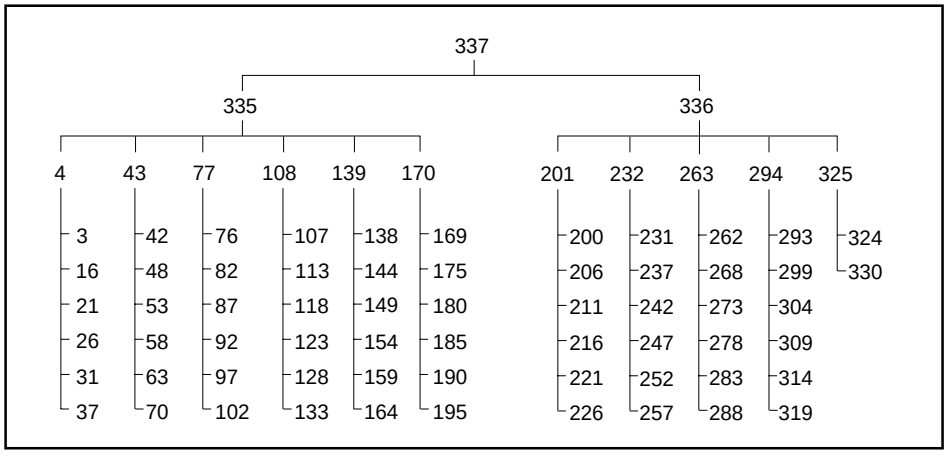
```
/Count 1
/Kids [ 4 0 R ]
>>
endobj
4 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /ProcSet 6 0 R >>
/MediaBox [ 0 0 612 792 ]
/Contents 5 0 R
>>
endobj
5 0 obj
<< /Length 612 >>
stream
% Draw a black line segment, using
% the default line width.
150 250 m
150 350 l
S
% Draw thicker, dashed line segment.
4 w  % Set a linewidth of 4 points.
[4 6] 0 d
% Set a dash pattern with 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 linewidth to 1 unit
% Draw a rectangle, 1 unit light blue border,
% filled with red
.5 .75 1 rg % light blue for fill color
1 0 0 RG  % red for stroke color
200 300 50 75 re
B
% Draw a curve using a Bézier curve,
% filled with gray and with a colored border.
.5 .1 .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
0000000016 00000 n
0000000086 00000 n
0000000136 00000 n
0000000200 00000 n
0000000328 00000 n
0000000992 00000 n
trailer
<<
/Size 8
/Root 1 0 R
>>
startxref
1110
%%EOF
```

A.4 Pages tree

This example is a fragment of a PDF file, illustrating the structure of the Pages tree for a large document. It contains the Pages objects for a 62-page file. The structure of the Pages tree for this example is shown in Figure A.1. In the figure, the numbers are object numbers corresponding to the objects in the PDF document fragment contained in Example A.4.

Figure A.1 *Pages tree for 62-page document example*



Example A.4 *Pages tree for a document containing 62 pages*

```
337 0 obj
```

```
<<
/Kids [335 0 R 336 0 R]
/Count 62
/Type /Pages
>>
endobj
335 0 obj
<<
/Kids [4 0 R 43 0 R 77 0 R 108 0 R 139 0 R 170 0 R]
/Count 36
/Type /Pages
/Parent 337 0 R
>>
endobj
336 0 obj
<<
/Kids [201 0 R 232 0 R 263 0 R 294 0 R 325 0 R]
/Count 26
/Type /Pages
/Parent 337 0 R
>>
endobj
4 0 obj
<<
/Kids [3 0 R 16 0 R 21 0 R 26 0 R 31 0 R 37 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
43 0 obj
<<
/Kids [42 0 R 48 0 R 53 0 R 58 0 R 63 0 R 70 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
77 0 obj
<<
/Kids [76 0 R 82 0 R 87 0 R 92 0 R 97 0 R 102 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
108 0 obj
<<
```

```
/Kids [107 0 R 113 0 R 118 0 R 123 0 R 128 0 R 133 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
139 0 obj
<<
/Kids [138 0 R 144 0 R 149 0 R 154 0 R 159 0 R 164 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
170 0 obj
<<
/Kids [169 0 R 175 0 R 180 0 R 185 0 R 190 0 R 195 0 R]
/Count 6
/Type /Pages
/Parent 335 0 R
>>
endobj
201 0 obj
<<
/Kids [200 0 R 206 0 R 211 0 R 216 0 R 221 0 R 226 0 R]
/Count 6
/Type /Pages
/Parent 336 0 R
>>
endobj
232 0 obj
<<
/Kids [231 0 R 237 0 R 242 0 R 247 0 R 252 0 R 257 0 R]
/Count 6
/Type /Pages
/Parent 336 0 R
>>
endobj
263 0 obj
<<
/Kids [262 0 R 268 0 R 273 0 R 278 0 R 283 0 R 288 0 R]
/Count 6
/Type /Pages
/Parent 336 0 R
>>
endobj
294 0 obj
<<
```

```
/Kids [293 0 R 299 0 R 304 0 R 309 0 R 314 0 R 319 0 R]
/Count 6
/Type /Pages
/Parent 336 0 R
>>
endobj
325 0 obj
<<
/Kids [324 0 R 330 0 R]
/Count 2
/Type /Pages
/Parent 336 0 R
>>
endobj
```

A.5 Outline

This section from a PDF file illustrates the structure of an outline tree with six entries. Example A.5 shows the outline with all entries open, as illustrated in Figure A.2.

Figure A.2 *Example of outline with six items, all open*

Onscreen 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

Example A.5 *Six entry outline, all items open*

```
21 0 obj
<<
/Count 6
/Type /Outlines
/First 22 0 R
/Last 29 0 R
>>
endobj
22 0 obj
<<
/Parent 21 0 R
```

```
/Dest [3 0 R /Top 0 792 0]
/Title (Document)
/Next 29 0 R
/First 25 0 R
/Last 28 0 R
/Count 4
>>
endobj
25 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Parent 22 0 R
/Title (Section 1)
/Next 26 0 R
>>
endobj
26 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Prev 25 0 R
/Next 28 0 R
/Parent 22 0 R
/Title (Section 2)
/First 27 0 R
/Last 27 0 R
/Count 1
>>
endobj
27 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Parent 26 0 R
/Title (Subsection 1)
>>
endobj
28 0 obj
<<
/Dest [3 0 R /FitR 3 255 622 792]
/Prev 26 0 R
/Parent 22 0 R
/Title (Section 3)
>>
endobj
29 0 obj
<<
/Prev 22 0 R
/Parent 21 0 R
/Dest [3 0 R /FitR 3 255 622 792]
```

```
/Title (Summary)
>>
endobj
```

Example A.6 is the same as Example A.5, except that one of the outline items has been closed. The outline appears as shown in Figure A.3.

Figure A.3 *Example of outline with six items, five of which are open*

Onscreen appearance	Object number	Count
	21	5
Document	22	3
Section 1	25	0
Section 2	26	–1
Section 3	28	0
Summary	29	0

Example A.6 *Six entry outline, five entries open*

```
21 0 obj
<<
/Count 5
/Type /Outlines
/First 22 0 R
/Last 29 0 R
>>
endobj
22 0 obj
<<
/Parent 21 0 R
/Dest [3 0 R /Top 0 792 0]
/Title (Document)
/Next 29 0 R
/First 25 0 R
/Last 28 0 R
/Count 3
>>
endobj
25 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Parent 22 0 R
/Title (Section 1)
/Next 26 0 R
>>
```

```
endobj
26 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Prev 25 0 R
/Next 28 0 R
/Parent 22 0 R
/Title (Section 2)
/First 27 0 R
/Last 27 0 R
/Count -1
>>
endobj
27 0 obj
<<
/Dest [3 0 R /FitR -38 255 650 792]
/Parent 26 0 R
/Title (Subsection 1)
>>
endobj
28 0 obj
<<
/Dest [3 0 R /FitR 3 255 622 792]
/Prev 26 0 R
/Parent 22 0 R
/Title (Section 3)
>>
endobj
29 0 obj
<<
/Prev 22 0 R
/Parent 21 0 R
/Dest [3 0 R /FitR 3 255 622 792]
/Title (Summary)
>>
endobj
```

A.6 Updated file

This example shows the structure of a PDF file as it is updated several times; multiple body sections, cross-reference sections, and trailers. In addition, it illustrates the fact that once an object ID has been assigned to an object, it keeps the ID until it is deleted, even if the object is altered. Finally, it illustrates the re-use 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 used in Section A.1, "Minimal PDF file." This file is not shown again here. First, four text annotations are added and the file saved. Next, the text of one of the annotations is altered, and the file saved. Following this, two of the text annotations are deleted, and the file saved again. Finally, three text annotations are added, and the file saved again.

The segments added to the file at each stage are shown separately. Throughout this example, objects are referred to by their object IDs, made up of the object number and generation number, rather than simply by the object number, as was done in earlier examples. This is necessary because objects are re-used in this example, so that the object number is not a unique identifier.

Note *The tables in this section show only the objects that are modified at some point during the updating process. Objects from the example file in Section A.1, "Minimal PDF file" that are never altered during the update are not shown.*

A.6.1 Add four text annotations

Four text annotations were added to the initial file and the file saved. Table A.4 lists the objects in this update.

Table A.4 *Object use after adding four text annotations*

Object ID	*Object type*
4 0	Page
7 0	Annots array
8 0	Text annotation
9 0	Text annotation
10 0	Text annotation
11 0	Text annotation

Example A.7 shows the lines added to the file by this update. The Page object is updated because an **Annots** key has been added. Note that the file's trailer now contains a **Prev** key, which points to the original cross-reference section in the file, while the **startxref** value at the end of the file points to the cross-reference section added by the update.

Example A.7 *Update section of PDF file when four text annotations are added*

```
4 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /ProcSet 6 0 R >>
/MediaBox [0 0 612 792]
/Contents 5 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
/Open true
/Rect [44 616 162 735]
/Contents (Text #1)
>>
endobj
9 0 obj
<<
/Type /Annot
/Subtype /Text
/Open false
/Rect [224 668 457 735]
/Contents (Text #2)
>>
endobj
10 0 obj
<<
/Type /Annot
/Subtype /Text
/Open true
/Rect [239 393 328 622]
/Contents (Text #3)
>>
endobj
11 0 obj
<<
/Type /Annot
/Subtype /Text
/Open false
/Rect [34 398 225 575]
/Contents (Text #4)
>>
endobj
xref
0 1
0000000000 65535 f
4 1
0000000612 00000 n
7 5
0000000747 00000 n
0000000792 00000 n
```

```
0000000897 00000 n
0000001004 00000 n
0000001111 00000 n
trailer
<<
/Size 12
/Root 1 0 R
/Prev 408
>>
startxref
1218
%%EOF
```

A.6.2 Modify text of one annotation

The lines shown in Example A.8 were added to the file when it was saved after modifying one text annotation. Note that the file now contains two copies of the object with ID 10 0 (the text annotation that was modified), and that the cross-reference section added points to the more recent version of the object. The cross-reference section added contains one subsection. The subsection contains an entry only for the object that was modified. In addition, the **Prev** key in the file's trailer has been updated to point to the cross-reference section added by the previous update, while the **startxref** value at the end of the file points to the newly added cross-reference section.

Example A.8 *Update section of PDF file when one text annotation is modified*

```
10 0 obj
<<
/Type /Annot
/Subtype /Text
/Open true
/Rect [239 393 328 622]
/Contents (Modified Text #3)
>>
endobj
xref
10 1
0000001444 00000 n
trailer
<<
/Size 12
/Root 1 0 R
/Prev 1218
>>
startxref
1560
%%EOF
```

A.6.3 Delete two annotations

Table A.5 lists the objects updated when two text annotations were deleted and the file saved.

Table A.5 *Object use after deleting two text annotations*

Object ID	*Object type*
7 0	Annots 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 fewer annotations.

Example A.9 shows the lines added when the file was saved. Note that objects with IDs 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 an **f**. The cross-reference section added in this step contains four entries, corresponding to 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 objects, and must now point to the newly freed entry for 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 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 re-use 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** key in the trailer dictionary has again been updated, so that it points to the cross-reference section added in the previous step, and the **startxref** value points to the newly added cross-reference section.

Example A.9 *Update section of PDF file when two text annotations are deleted*

```
7 0 obj
[10 0 R 11 0 R]
endobj
xref
0 1
0000000008 65535 f
7 3
0000001658 00000 n
0000000009 00001 f
0000000000 00001 f
trailer
<<
/Size 12
```

```
/Root 1 0 R
/Prev 1560
>>
startxref
1691
%%EOF
```

A.6.4 Add three annotations

Finally, three text annotations were added to the file. Table A.6 lists the objects involved in this update.

Table A.6 *Object use after adding three text annotations*

Object ID	*Object type*
7 0	Annots array
8 1	Text annotation
9 1	Text annotation
12 0	Text annotation

Object numbers 8 and 9, which were the object numbers used for the two annotations deleted in the previous step, have been re-used. The new objects have been given a generation number of 1, however. In addition, the third text annotation added was assigned the previously unused object ID of 12 0.

Example A.10 shows the lines added to the file by this update. The cross-reference section added in this step contains five entries, corresponding object number 0, the Annots array, and the three annotations added. The entry for object number zero is updated because the previously free entries for object numbers 8 and 9 have been re-used. The entry for object number zero 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 new text annotations.

As in previous updates, the trailer's Prev key and **startxref** value have been updated.

The annotation with object ID 12 0 illustrates the splitting of a long text string across multiple lines, as well as the technique for including non-standard characters in a string. In this case, the character is an ellipsis (…), which is character code 203 (octal) in the **PDFDocEncoding** used for text annotations.

Example A.10 *Update section of PDF file after three text annotations are added*

```
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
/Open true
/Rect [58 657 172 742]
/Contents (New Text #1)
>>
endobj
9 1 obj
<<
/Type /Annot
/Subtype /Text
/Open false
/Rect [389 459 570 537]
/Contents (New Text Annotation #2)
>>
endobj
12 0 obj
<<
/Type /Annot
/Subtype /Text
/Open true
/Rect [44 253 473 337]
/Contents (A longer annotation which we'll call, \
for lack of a better name\203New T\
ext #3)
>>
endobj
xref
0 1
0000000000 65535 f
7 3
0000001853 00000 n
0000001905 00001 n
0000002014 00001 n
12 1
0000002136 00000 n
trailer
<<
/Size 13
/Root 1 0 R
/Prev 1691
>>
startxref
2315
%%EOF
```

APPENDIX B

Summary of Page Marking Operators

Following is a list of all page marking operators used in PDF files, arranged alphabetically. For each operator, a brief description is given, along with a reference to the page in this document where the operator is discussed in detail. Words shown in boldface in the summary column are PostScript language operators.

Table B.1 *PDF page marking operators*

Operator	*Summary*	*Page*	
b	**closepath**, **fill**, and **stroke** path	227	
B	**fill** and **stroke** path	227	
b*	**closepath**, **eofill**, and **stroke** path	227	
B*	**eofill** and **stroke** path	227	
BDC	begin marked content, with a dictionary	240	PDF 1.2
BI	begin image	238	
BMC	begin marked content	240	PDF 1.2
BT	begin text object	233	
BX	begin section allowing undefined operators	239	PDF 1.1
c	**curveto**	223	
cm	**concat**. Concatenates the matrix to the current transformation matrix.	213	
cs	**setcolorspace** for fill	220	PDF 1.1
CS	**setcolorspace** for stroke	221	PDF 1.1
d	**setdash**	215	
d0	**setcharwidth** for Type 3 font	239	
d1	**setcachedevice** for Type 3 font	239	
Do	execute the named XObject	236	

Table B.1 *PDF page marking operators*

Operator	*Summary*	*Page*	
DP	mark a place in the content stream, with a dictionary	241	PDF 1.2
EI	end image	238	
EMC	end marked content	240	PDF 1.2
ET	end text object	233	
EX	end section that allows undefined operators	239	PDF 1.1
f	**fill** path	226	
F	**fill** path	226	
f*	**eofill** path	226	
g	**setgray** (fill)	219	
G	**setgray** (stroke)	220	
gs	set parameters in the extended graphics state	217	PDF 1.2
h	**closepath**	225	
i	**setflat**	213	
ID	begin image data	238	
j	**setlinejoin**	216	
J	**setlinecap**	214	
k	**setcmykcolor** (fill)	220	
K	**setcmykcolor** (stroke)	220	
l	**lineto**	223	
m	**moveto**	223	
M	**setmiterlimit**	217	
MP	mark a place in the content stream	241	PDF 1.2
n	end path without **fill** or **stroke**	226	
q	save graphics state	213	
Q	restore graphics state	213	
re	rectangle	224	
rg	**setrgbcolor** (fill)	220	
RG	**setrgbcolor** (stroke)	220	

Table B.1 *PDF page marking operators*

Operator	*Summary*	*Page*	
s	**closepath** and **stroke** path	226	
S	**stroke** path	226	
sc	**setcolor** (fill)	221	PDF 1.1
SC	**setcolor** (stroke)	221	PDF 1.1
scn	**setcolor** (fill, in pattern and separation color spaces)	221	PDF 1.2
SCN	**setcolor** (stroke, in pattern and separation color spaces)	221	PDF 1.2
Tc	set character spacing	229	
Td	move text current point	233	
TD	move text current point and set leading	233	
Tf	set font name and size	230	
Tj	show text	234	
TJ	show text, allowing individual character positioning	235	
TL	set leading	230	
Tm	set text matrix	233	
Tr	set text rendering mode	232	
Ts	set super/subscripting text rise	232	
Tw	set word spacing	229	
Tz	set horizontal scaling	230	
T*	move to start of next line	233	
v	**curveto**	223	
w	**setlinewidth**	216	
W	**clip**	227	
W*	**eoclip**	227	
y	**curveto**	224	
'	move to next line and show text	234	
"	move to next line and show text	235	

APPENDIX C

Predefined Font Encodings

PDF provides several predefined font encodings:

- **MacRomanEncoding**, **MacExpertEncoding**, and **WinAnsiEncoding** may be used in Font and Encoding objects.
- **PDFDocEncoding** is the encoding used in outline entries, text annotations, and strings in the Info dictionary.
- **StandardEncoding** is the built-in encoding for many fonts.

This appendix contains three tables describing these encodings. The first table shows all encodings except **MacExpertEncoding** and is arranged alphabetically by character name. The second table is similar, except that it is arranged numerically by character code. The third table shows the encoding for **MacExpertEncoding**, which is shown in a separate table because it has a substantially different character set than the other encodings.

C.1 Predefined encodings sorted by character name

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
A	A	65	101	65	101	65	101	65	101
Æ	AE	225	341	174	256	198	306	198	306
Á	Aacute	—	—	231	347	193	301	193	301
Â	Acircumflex	—	—	229	345	194	302	194	302
Ä	Adieresis	—	—	128	200	196	304	196	304
À	Agrave	—	—	203	313	192	300	192	300
Å	Aring	—	—	129	201	197	305	197	305
Ã	Atilde	—	—	204	314	195	303	195	303
B	B	66	102	66	102	66	102	66	102
C	C	67	103	67	103	67	103	67	103
Ç	Ccedilla	—	—	130	202	199	307	199	307
D	D	68	104	68	104	68	104	68	104
E	E	69	105	69	105	69	105	69	105
É	Eacute	—	—	131	203	201	311	201	311
Ê	Ecircumflex	—	—	230	346	202	312	202	312
Ë	Edieresis	—	—	232	350	203	313	203	313
È	Egrave	—	—	233	351	200	310	200	310
Ð	Eth	—	—	—	—	208	320	208	320
F	F	70	106	70	106	70	106	70	106
G	G	71	107	71	107	71	107	71	107
H	H	72	110	72	110	72	110	72	110
I	I	73	111	73	111	73	111	73	111
Í	Iacute	—	—	234	352	205	315	205	315
Î	Icircumflex	—	—	235	353	206	316	206	316
Ï	Idieresis	—	—	236	354	207	317	207	317
Ì	Igrave	—	—	237	355	204	314	204	314
J	J	74	112	74	112	74	112	74	112
K	K	75	113	75	113	75	113	75	113
L	L	76	114	76	114	76	114	76	114
Ł	Lslash	232	350	—	—	—	—	149	225
M	M	77	115	77	115	77	115	77	115
N	N	78	116	78	116	78	116	78	116
Ñ	Ntilde	—	—	132	204	209	321	209	321
O	O	79	117	79	117	79	117	79	117
Œ	OE	234	352	206	316	140	214	150	226
Ó	Oacute	—	—	238	356	211	323	211	323
Ô	Ocircumflex	—	—	239	357	212	324	212	324

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
Ö	Odieresis	—	—	133	205	214	326	214	326
Ò	Ograve	—	—	241	361	210	322	210	322
Ø	Oslash	233	351	175	257	216	330	216	330
Õ	Otilde	—	—	205	315	213	325	213	325
P	P	80	120	80	120	80	120	80	120
Q	Q	81	121	81	121	81	121	81	121
R	R	82	122	82	122	82	122	82	122
S	S	83	123	83	123	83	123	83	123
Š	Scaron	—	—	—	—	138	212	151	227
T	T	84	124	84	124	84	124	84	124
Þ	Thorn	—	—	—	—	222	336	222	336
U	U	85	125	85	125	85	125	85	125
Ú	Uacute	—	—	242	362	218	332	218	332
Û	Ucircumflex	—	—	243	363	219	333	219	333
Ü	Udieresis	—	—	134	206	220	334	220	334
Ù	Ugrave	—	—	244	364	217	331	217	331
V	V	86	126	86	126	86	126	86	126
W	W	87	127	87	127	87	127	87	127
X	X	88	130	88	130	88	130	88	130
Y	Y	89	131	89	131	89	131	89	131
Ý	Yacute	—	—	—	—	221	335	221	335
Ÿ	Ydieresis	—	—	217	331	159	237	152	230
Z	Z	90	132	90	132	90	132	90	132
Ž	Zcaron	—	—	—	—	—	—	153	231
a	a	97	141	97	141	97	141	97	141
á	aacute	—	—	135	207	225	341	225	341
â	acircumflex	—	—	137	211	226	342	226	342
´	acute	194	302	171	253	180	264	180	264
ä	adieresis	—	—	138	212	228	344	228	344
æ	ae	241	361	190	276	230	346	230	346
à	agrave	—	—	136	210	224	340	224	340
&	ampersand	38	46	38	46	38	46	38	46
å	aring	—	—	140	214	229	345	229	345
^	asciicircum	94	136	94	136	94	136	94	136
~	asciitilde	126	176	126	176	126	176	126	176
*	asterisk	42	52	42	52	42	52	42	52
@	at	64	100	64	100	64	100	64	100
ã	atilde	—	—	139	213	227	343	227	343
b	b	98	142	98	142	98	142	98	142

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
\	backslash	92	134	92	134	92	134	92	134
|	bar	124	174	124	174	124	174	124	174
{	braceleft	123	173	123	173	123	173	123	173
}	braceright	125	175	125	175	125	175	125	175
[	bracketleft	91	133	91	133	91	133	91	133
]	bracketright	93	135	93	135	93	135	93	135
˘	breve	198	306	249	371	—	—	24	30
¦	brokenbar	—	—	—	—	166	246	166	246
•	bullet	183	267	165	245	149	225	128	200
c	c	99	143	99	143	99	143	99	143
ˇ	caron	207	317	255	377	—	—	25	31
ç	ccedilla	—	—	141	215	231	347	231	347
¸	cedilla	203	313	252	374	184	270	184	270
¢	cent	162	242	162	242	162	242	162	242
ˆ	circumflex	195	303	246	366	136	210	26	32
:	colon	58	72	58	72	58	72	58	72
,	comma	44	54	44	54	44	54	44	54
©	copyright	—	—	169	251	169	251	169	251
¤	currency	168	250	219	333	164	244	164	244
d	d	100	144	100	144	100	144	100	144
†	dagger	178	262	160	240	134	206	129	201
‡	daggerdbl	179	263	224	340	135	207	130	202
°	degree	—	—	161	241	176	260	176	260
¨	dieresis	200	310	172	254	168	250	168	250
÷	divide	—	—	214	326	247	367	247	367
$	dollar	36	44	36	44	36	44	36	44
˙	dotaccent	199	307	250	372	—	—	27	33
ı	dotlessi	245	365	245	365	—	—	154	232
e	e	101	145	101	145	101	145	101	145
é	eacute	—	—	142	216	233	351	233	351
ê	ecircumflex	—	—	144	220	234	352	234	352
ë	edieresis	—	—	145	221	235	353	235	353
è	egrave	—	—	143	217	232	350	232	350
8	eight	56	70	56	70	56	70	56	70
…	ellipsis	188	274	201	311	133	205	131	203
—	emdash	208	320	209	321	151	227	132	204
–	endash	177	261	208	320	150	226	133	205
=	equal	61	75	61	75	61	75	61	75
ð	eth	—	—	—	—	240	360	240	360

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
!	exclam	33	41	33	41	33	41	33	41
¡	exclamdown	161	241	193	301	161	241	161	241
f	f	102	146	102	146	102	146	102	146
fi	fi	174	256	222	336	—	—	147	223
5	five	53	65	53	65	53	65	53	65
fl	fl	175	257	223	337	—	—	148	224
ƒ	florin	166	246	196	304	131	203	134	206
4	four	52	64	52	64	52	64	52	64
⁄	fraction	164	244	218	332	—	—	135	207
g	g	103	147	103	147	103	147	103	147
ß	germandbls	251	373	167	247	223	337	223	337
`	grave	193	301	96	140	96	140	96	140
>	greater	62	76	62	76	62	76	62	76
«	guillemotleft	171	253	199	307	171	253	171	253
»	guillemotright	187	273	200	310	187	273	187	273
‹	guilsinglleft	172	254	220	334	139	213	136	210
›	guilsinglright	173	255	221	335	155	233	137	211
h	h	104	150	104	150	104	150	104	150
˝	hungarumlaut	205	315	253	375	—	—	28	34
-	hyphen	45	55	45	55	45	55	45	55
i	i	105	151	105	151	105	151	105	151
í	iacute	—	—	146	222	237	355	237	355
î	icircumflex	—	—	148	224	238	356	238	356
ï	idieresis	—	—	149	225	239	357	239	357
ì	igrave	—	—	147	223	236	354	236	354
j	j	106	152	106	152	106	152	106	152
k	k	107	153	107	153	107	153	107	153
l	l	108	154	108	154	108	154	108	154
<	less	60	74	60	74	60	74	60	74
¬	logicalnot	—	—	194	302	172	254	172	254
ł	lslash	248	370	—	—	—	—	155	233
m	m	109	155	109	155	109	155	109	155
¯	macron	197	305	248	370	175	257	175	257
−	minus	—	—	—	—	—	—	138	212
µ	mu	—	—	181	265	181	265	181	265
×	multiply	—	—	—	—	215	327	215	327
n	n	110	156	110	156	110	156	110	156
9	nine	57	71	57	71	57	71	57	71
ñ	ntilde	—	—	150	226	241	361	241	361

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
#	numbersign	35	43	35	43	35	43	35	43
o	o	111	157	111	157	111	157	111	157
ó	oacute	—	—	151	227	243	363	243	363
ô	ocircumflex	—	—	153	231	244	364	244	364
ö	odieresis	—	—	154	232	246	366	246	366
œ	oe	250	372	207	317	156	234	156	234
˛	ogonek	206	316	254	376	—	—	29	35
ò	ograve	—	—	152	230	242	362	242	362
1	one	49	61	49	61	49	61	49	61
½	onehalf	—	—	—	—	189	275	189	275
¼	onequarter	—	—	—	—	188	274	188	274
¹	onesuperior	—	—	—	—	185	271	185	271
ª	ordfeminine	227	343	187	273	170	252	170	252
º	ordmasculine	235	353	188	274	186	272	186	272
ø	oslash	249	371	191	277	248	370	248	370
õ	otilde	—	—	155	233	245	365	245	365
p	p	112	160	112	160	112	160	112	160
¶	paragraph	182	266	166	246	182	266	182	266
(	parenleft	40	50	40	50	40	50	40	50
)	parenright	41	51	41	51	41	51	41	51
%	percent	37	45	37	45	37	45	37	45
.	period	46	56	46	56	46	56	46	56
·	periodcentered	180	264	225	341	183	267	183	267
‰	perthousand	189	275	228	344	137	211	139	213
+	plus	43	53	43	53	43	53	43	53
±	plusminus	—	—	177	261	177	261	177	261
q	q	113	161	113	161	113	161	113	161
?	question	63	77	63	77	63	77	63	77
¿	questiondown	191	277	192	300	191	277	191	277
"	quotedbl	34	42	34	42	34	42	34	42
„	quotedblbase	185	271	227	343	132	204	140	214
“	quotedblleft	170	252	210	322	147	223	141	215
”	quotedblright	186	272	211	323	148	224	142	216
‘	quoteleft	96	140	212	324	145	221	143	217
’	quoteright	39	47	213	325	146	222	144	220
‚	quotesinglbase	184	270	226	342	130	202	145	221
'	quotesingle	169	251	39	47	39	47	39	47
r	r	114	162	114	162	114	162	114	162
®	registered	—	—	168	250	174	256	174	256

Char	Name	StandardEncoding		MacRomanEncoding		WinAnsiEncoding		PDFDocEncoding	
		Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal
˚	ring	202	312	251	373	176	260	30	36
s	s	115	163	115	163	115	163	115	163
š	scaron	—	—	—	—	154	232	157	235
§	section	167	247	164	244	167	247	167	247
;	semicolon	59	73	59	73	59	73	59	73
7	seven	55	67	55	67	55	67	55	67
6	six	54	66	54	66	54	66	54	66
/	slash	47	57	47	57	47	57	47	57
	space	32	40	32, 202	40,312	32	40	32	40
£	sterling	163	243	163	243	163	243	163	243
t	t	116	164	116	164	116	164	116	164
þ	thorn	—	—	—	—	254	376	254	376
3	three	51	63	51	63	51	63	51	63
¾	threequarters	—	—	—	—	190	276	190	276
³	threesuperior	—	—	—	—	179	263	179	263
˜	tilde	196	304	247	367	152	230	31	37
™	trademark	—	—	170	252	153	231	146	222
2	two	50	62	50	62	50	62	50	62
²	twosuperior	—	—	—	—	178	262	178	262
u	u	117	165	117	165	117	165	117	165
ú	uacute	—	—	156	234	250	372	250	372
û	ucircumflex	—	—	158	236	251	373	251	373
ü	udieresis	—	—	159	237	252	374	252	374
ù	ugrave	—	—	157	235	249	371	249	371
_	underscore	95	137	95	137	95	137	95	137
v	v	118	166	118	166	118	166	118	166
w	w	119	167	119	167	119	167	119	167
x	x	120	170	120	170	120	170	120	170
y	y	121	171	121	171	121	171	121	171
ý	yacute	—	—	—	—	253	375	253	375
ÿ	ydieresis	—	—	216	330	255	377	255	377
¥	yen	165	245	180	264	165	245	165	245
z	z	122	172	122	172	122	172	122	172
ž	zcaron	—	—	—	—	—	—	158	236
0	zero	48	60	48	60	48	60	48	60

Note *In the* **WinAnsiEncoding**, *the hyphen character can also be accessed using a character code of 173, the space using 160, and bullets are used for the otherwise unused character codes 127, 128, 129, 141, 142, 143, 144, 157, and 158.*

C.2 Predefined encodings sorted by character code

Note *Character codes 0 through 23 are not used in any of the predefined encodings.*

Code		*StandardEncoding*	*MacRomanEncoding*	*WinAnsiEncoding*	*PDFDocEncoding*
Decimal	*Octal*				
24	30	—	—	—	breve
25	31	—	—	—	caron
26	32	—	—	—	circumflex
27	33	—	—	—	dotaccent
28	34	—	—	—	hungarumlaut
29	35	—	—	—	ogonek
30	36	—	—	—	ring
31	37	—	—	—	tilde
32	40	space	space	space	space
33	41	exclam	exclam	exclam	exclam
34	42	quotedbl	quotedbl	quotedbl	quotedbl
35	43	numbersign	numbersign	numbersign	numbersign
36	44	dollar	dollar	dollar	dollar
37	45	percent	percent	percent	percent
38	46	ampersand	ampersand	ampersand	ampersand
39	47	quoteright	quotesingle	quotesingle	quotesingle
40	50	parenleft	parenleft	parenleft	parenleft
41	51	parenright	parenright	parenright	parenright
42	52	asterisk	asterisk	asterisk	asterisk
43	53	plus	plus	plus	plus
44	54	comma	comma	comma	comma
45	55	hyphen	hyphen	hyphen	hyphen
46	56	period	period	period	period
47	57	slash	slash	slash	slash
48	60	zero	zero	zero	zero
49	61	one	one	one	one
50	62	two	two	two	two
51	63	three	three	three	three
52	64	four	four	four	four
53	65	five	five	five	five
54	66	six	six	six	six
55	67	seven	seven	seven	seven
56	70	eight	eight	eight	eight
57	71	nine	nine	nine	nine
58	72	colon	colon	colon	colon
59	73	semicolon	semicolon	semicolon	semicolon
60	74	less	less	less	less
61	75	equal	equal	equal	equal

Code		StandardEncoding	MacRomanEncoding	WinAnsiEncoding	PDFDocEncoding
Decimal	Octal				
62	76	greater	greater	greater	greater
63	77	question	question	question	question
64	100	at	at	at	at
65	101	A	A	A	A
66	102	B	B	B	B
67	103	C	C	C	C
68	104	D	D	D	D
69	105	E	E	E	E
70	106	F	F	F	F
71	107	G	G	G	G
72	110	H	H	H	H
73	111	I	I	I	I
74	112	J	J	J	J
75	113	K	K	K	K
76	114	L	L	L	L
77	115	M	M	M	M
78	116	N	N	N	N
79	117	O	O	O	O
80	120	P	P	P	P
81	121	Q	Q	Q	Q
82	122	R	R	R	R
83	123	S	S	S	S
84	124	T	T	T	T
85	125	U	U	U	U
86	126	V	V	V	V
87	127	W	W	W	W
88	130	X	X	X	X
89	131	Y	Y	Y	Y
90	132	Z	Z	Z	Z
91	133	bracketleft	bracketleft	bracketleft	bracketleft
92	134	backslash	backslash	backslash	backslash
93	135	bracketright	bracketright	bracketright	bracketright
94	136	asciicircum	asciicircum	asciicircum	asciicircum
95	137	underscore	underscore	underscore	underscore
96	140	quoteleft	grave	grave	grave
97	141	a	a	a	a
98	142	b	b	b	b
99	143	c	c	c	c
100	144	d	d	d	d
101	145	e	e	e	e
102	146	f	f	f	f
103	147	g	g	g	g

Code		StandardEncoding	MacRomanEncoding	WinAnsiEncoding	PDFDocEncoding
Decimal	Octal				
104	150	h	h	h	h
105	151	i	i	i	i
106	152	j	j	j	j
107	153	k	k	k	k
108	154	l	l	l	l
109	155	m	m	m	m
110	156	n	n	n	n
111	157	o	o	o	o
112	160	p	p	p	p
113	161	q	q	q	q
114	162	r	r	r	r
115	163	s	s	s	s
116	164	t	t	t	t
117	165	u	u	u	u
118	166	v	v	v	v
119	167	w	w	w	w
120	170	x	x	x	x
121	171	y	y	y	y
122	172	z	z	z	z
123	173	braceleft	braceleft	braceleft	braceleft
124	174	bar	bar	bar	bar
125	175	braceright	braceright	braceright	braceright
126	176	asciitilde	asciitilde	asciitilde	asciitilde
127	177	—	—	bullet	—
128	200	—	Adieresis	bullet	bullet
129	201	—	Aring	bullet	dagger
130	202	—	Ccedilla	quotesinglbase	daggerdbl
131	203	—	Eacute	florin	ellipsis
132	204	—	Ntilde	quotedblbase	emdash
133	205	—	Odieresis	ellipsis	endash
134	206	—	Udieresis	dagger	florin
135	207	—	aacute	daggerdbl	fraction
136	210	—	agrave	circumflex	guilsinglleft
137	211	—	acircumflex	perthousand	guilsinglright
138	212	—	adieresis	Scaron	minus
139	213	—	atilde	guilsinglleft	perthousand
140	214	—	aring	OE	quotedblbase
141	215	—	ccedilla	bullet	quotedblleft
142	216	—	eacute	bullet	quotedblright
143	217	—	egrave	bullet	quoteleft
144	220	—	ecircumflex	bullet	quoteright
145	221	—	edieresis	quoteleft	quotesinglbase

Code		StandardEncoding	MacRomanEncoding	WinAnsiEncoding	PDFDocEncoding
Decimal	Octal				
146	222	—	iacute	quoteright	trademark
147	223	—	igrave	quotedblleft	fi
148	224	—	icircumflex	quotedblright	fl
149	225	—	idieresis	bullet	Lslash
150	226	—	ntilde	endash	OE
151	227	—	oacute	emdash	Scaron
152	230	—	ograve	tilde	Ydieresis
153	231	—	ocircumflex	trademark	Zcaron
154	232	—	odieresis	scaron	dotlessi
155	233	—	otilde	guilsinglright	lslash
156	234	—	uacute	oe	oe
157	235	—	ugrave	bullet	scaron
158	236	—	ucircumflex	bullet	zcaron
159	237	—	udieresis	Ydieresis	—
160	240	—	dagger	space	—
161	241	exclamdown	degree	exclamdown	exclamdown
162	242	cent	cent	cent	cent
163	243	sterling	sterling	sterling	sterling
164	244	fraction	section	currency	currency
165	245	yen	bullet	yen	yen
166	246	florin	paragraph	brokenbar	brokenbar
167	247	section	germandbls	section	section
168	250	currency	registered	dieresis	dieresis
169	251	quotesingle	copyright	copyright	copyright
170	252	quotedblleft	trademark	ordfeminine	ordfeminine
171	253	guillemotleft	acute	guillemotleft	guillemotleft
172	254	guilsinglleft	dieresis	logicalnot	logicalnot
173	255	guilsinglright	—	hyphen	—
174	256	fi	AE	registered	registered
175	257	fl	Oslash	macron	macron
176	260	—	—	degree	degree
177	261	endash	plusminus	plusminus	plusminus
178	262	dagger	—	twosuperior	twosuperior
179	263	daggerdbl	—	threesuperior	threesuperior
180	264	periodcentered	yen	acute	acute
181	265	—	mu	mu	mu
182	266	paragraph	—	paragraph	paragraph
183	267	bullet	—	periodcentered	periodcentered
184	270	quotesinglbase	—	cedilla	cedilla
185	271	quotedblbase	—	onesuperior	onesuperior
186	272	quotedblright	—	ordmasculine	ordmasculine
187	273	guillemotright	ordfeminine	guillemotright	guillemotright

Code		StandardEncoding	MacRomanEncoding	WinAnsiEncoding	PDFDocEncoding
Decimal	Octal				
188	274	ellipsis	ordmasculine	onequarter	onequarter
189	275	perthousand	—	onehalf	onehalf
190	276	—	ae	threequarters	threequarters
191	277	questiondown	oslash	questiondown	questiondown
192	300	—	questiondown	Agrave	Agrave
193	301	grave	exclamdown	Aacute	Aacute
194	302	acute	logicalnot	Acircumflex	Acircumflex
195	303	circumflex	—	Atilde	Atilde
196	304	tilde	florin	Adieresis	Adieresis
197	305	macron	—	Aring	Aring
198	306	breve	—	AE	AE
199	307	dotaccent	guillemotleft	Ccedilla	Ccedilla
200	310	dieresis	guillemotright	Egrave	Egrave
201	311	—	ellipsis	Eacute	Eacute
202	312	ring	space	Ecircumflex	Ecircumflex
203	313	cedilla	Agrave	Edieresis	Edieresis
204	314	—	Atilde	Igrave	Igrave
205	315	hungarumlaut	Otilde	Iacute	Iacute
206	316	ogonek	OE	Icircumflex	Icircumflex
207	317	caron	oe	Idieresis	Idieresis
208	320	emdash	endash	Eth	Eth
209	321	—	emdash	Ntilde	Ntilde
210	322	—	quotedblleft	Ograve	Ograve
211	323	—	quotedblright	Oacute	Oacute
212	324	—	quoteleft	Ocircumflex	Ocircumflex
213	325	—	quoteright	Otilde	Otilde
214	326	—	divide	Odieresis	Odieresis
215	327	—	—	multiply	multiply
216	330	—	ydieresis	Oslash	Oslash
217	331	—	Ydieresis	Ugrave	Ugrave
218	332	—	fraction	Uacute	Uacute
219	333	—	currency	Ucircumflex	Ucircumflex
220	334	—	guilsinglleft	Udieresis	Udieresis
221	335	—	guilsinglright	Yacute	Yacute
222	336	—	fi	Thorn	Thorn
223	337	—	fl	germandbls	germandbls
224	340	—	daggerdbl	agrave	agrave
225	341	AE	periodcentered	aacute	aacute
226	342	—	quotesinglbase	acircumflex	acircumflex
227	343	ordfeminine	quotedblbase	atilde	atilde
228	344	—	perthousand	adieresis	adieresis
229	345	—	Acircumflex	aring	aring

Code		StandardEncoding	MacRomanEncoding	WinAnsiEncoding	PDFDocEncoding
Decimal	Octal				
230	346	—	Ecircumflex	ae	ae
231	347	—	Aacute	ccedilla	ccedilla
232	350	Lslash	Edieresis	egrave	egrave
233	351	Oslash	Egrave	eacute	eacute
234	352	OE	Iacute	ecircumflex	ecircumflex
235	353	ordmasculine	Icircumflex	edieresis	edieresis
236	354	—	Idieresis	igrave	igrave
237	355	—	Igrave	iacute	iacute
238	356	—	Oacute	icircumflex	icircumflex
239	357	—	Ocircumflex	idieresis	idieresis
240	360	—	—	eth	eth
241	361	ae	Ograve	ntilde	ntilde
242	362	—	Uacute	ograve	ograve
243	363	—	Ucircumflex	oacute	oacute
244	364	—	Ugrave	ocircumflex	ocircumflex
245	365	dotlessi	dotlessi	otilde	otilde
246	366	—	circumflex	odieresis	odieresis
247	367	—	tilde	divide	divide
248	370	lslash	macron	oslash	oslash
249	371	oslash	breve	ugrave	ugrave
250	372	oe	dotaccent	uacute	uacute
251	373	germandbls	ring	ucircumflex	ucircumflex
252	374	—	cedilla	udieresis	udieresis
253	375	—	hungarumlaut	yacute	yacute
254	376	—	ogonek	thorn	thorn
255	377	—	caron	ydieresis	ydieresis

C.3 MacExpert encoding

Char	Name	Code		Char	Name	Code	
		Decimal	Octal			Decimal	Octal
Æ	AEsmall	190	276	Ł	Lslashsmall	194	302
Á	Aacutesmall	135	207	L	Lsmall	108	154
Â	Acircumflexsmall	137	211	¯	Macronsmall	244	364
´	Acutesmall	39	47	M	Msmall	109	155
Ä	Adieresissmall	138	212	N	Nsmall	110	156
À	Agravesmall	136	210	Ñ	Ntildesmall	150	226
Å	Aringsmall	140	214	Œ	OEsmall	207	317
A	Asmall	97	141	Ó	Oacutesmall	151	227
Ã	Atildesmall	139	213	Ô	Ocircumflexsmall	153	231
˘	Brevesmall	243	363	Ö	Odieresissmall	154	232
B	Bsmall	98	142	˛	Ogoneksmall	242	362
ˇ	Caronsmall	174	256	Ò	Ogravesmall	152	230
Ç	Ccedillasmall	141	215	Ø	Oslashsmall	191	277
¸	Cedillasmall	201	311	O	Osmall	111	157
ˆ	Circumflexsmall	94	136	Õ	Otildesmall	155	233
C	Csmall	99	143	P	Psmall	112	160
¨	Dieresissmall	172	254	Q	Qsmall	113	161
˙	Dotaccentsmall	250	372	°	Ringsmall	251	373
D	Dsmall	100	144	R	Rsmall	114	162
É	Eacutesmall	142	216	š	Scaronsmall	167	247
Ê	Ecircumflexsmall	144	220	S	Ssmall	115	163
Ë	Edieresissmall	145	221	Þ	Thornsmall	185	271
È	Egravesmall	143	217	˜	Tildesmall	126	176
E	Esmall	101	145	T	Tsmall	116	164
Ð	Ethsmall	68	104	Ú	Uacutesmall	156	234
F	Fsmall	102	146	Û	Ucircumflexsmall	158	236
`	Gravesmall	96	140	Ü	Udieresissmall	159	237
G	Gsmall	103	147	Ù	Ugravesmall	157	235
H	Hsmall	104	150	U	Usmall	117	165
˝	Hungarumlautsmall	34	42	V	Vsmall	118	166
Í	Iacutesmall	146	222	W	Wsmall	119	167
Î	Icircumflexsmall	148	224	X	Xsmall	120	170
Ï	Idieresissmall	149	225	Ý	Yacutesmall	180	264
Ì	Igravesmall	147	223	Ÿ	Ydieresissmall	216	330
I	Ismall	105	151	Y	Ysmall	121	171
J	Jsmall	106	152	ž	Zcaronsmall	189	275
K	Ksmall	107	153	Z	Zsmall	122	172

Char	Name	Code		Char	Name	Code	
		Decimal	Octal			Decimal	Octal
&	ampersandsmall	38	46	l	lsuperior	241	361
a	asuperior	129	201	m	msuperior	247	367
b	bsuperior	245	365	9	nineinferior	187	273
¢	centinferior	169	251	9	nineoldstyle	57	71
¢	centoldstyle	35	43	9	ninesuperior	225	341
¢	centsuperior	130	202	n	nsuperior	246	366
:	colon	58	72	.	onedotenleader	43	53
₡	colonmonetary	123	173	⅛	oneeighth	74	112
,	comma	44	54	l	onefitted	124	174
,	commainferior	178	262	½	onehalf	72	110
,	commasuperior	248	370	1	oneinferior	193	301
$	dollarinferior	182	266	1	oneoldstyle	49	61
$	dollaroldstyle	36	44	¼	onequarter	71	107
$	dollarsuperior	37	45	1	onesuperior	218	332
d	dsuperior	235	353	⅓	onethird	78	116
8	eightinferior	165	245	o	osuperior	175	257
8	eightoldstyle	56	70	(	parenleftinferior	91	133
8	eightsuperior	161	241	(	parenleftsuperior	40	50
e	esuperior	228	344	)	parenrightinferior	93	135
¡	exclamdownsmall	214	326	)	parenrightsuperior	41	51
!	exclamsmall	33	41	.	period	46	56
ff	ff	86	126	.	periodinferior	179	263
ffi	ffi	89	131	·	periodsuperior	249	371
ffl	ffl	90	132	¿	questiondownsmall	192	300
fi	fi	87	127	?	questionsmall	63	77
–	figuredash	208	320	r	rsuperior	229	345
⅝	fiveeighths	76	114	Rp	rupiah	125	175
5	fiveinferior	176	260	;	semicolon	59	73
5	fiveoldstyle	53	65	⅞	seveneighths	77	115
5	fivesuperior	222	336	7	seveninferior	166	246
fl	fl	88	130	7	sevenoldstyle	55	67
4	fourinferior	162	242	7	sevensuperior	224	340
4	fouroldstyle	52	64	6	sixinferior	164	244
4	foursuperior	221	335	6	sixoldstyle	54	66
/	fraction	47	57	6	sixsuperior	223	337
-	hyphen	45	55		space	32	40
-	hypheninferior	95	137	s	ssuperior	234	352
-	hyphensuperior	209	321	⅜	threeeighths	75	113
i	isuperior	233	351	3	threeinferior	163	243

Char	Name	Code		Char	Name	Code	
		Decimal	Octal			Decimal	Octal
3	threeoldstyle	51	63				
¾	threequarters	73	111				
—	threequartersemdash	61	75				
³	threesuperior	220	334				
ᵗ	tsuperior	230	346				
..	twodotenleader	42	52				
₂	twoinferior	170	252				
2	twooldstyle	50	62				
²	twosuperior	219	333				
⅔	twothirds	79	117				
₀	zeroinferior	188	274				
0	zerooldstyle	48	60				
⁰	zerosuperior	226	342				

APPENDIX D

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 Manual, Second 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.

All limits are sufficiently large that most PDF files should never approach them. However, using the techniques described in Chapters 10 through 14 of this book will further reduce the chance of reaching these limits.

This appendix describes typical limits for Acrobat Exchange and Acrobat Reader. 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 of 65,535 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 10GB).

Table D.1 describes the architectural limits for most PDF viewer applications running on 32-bit machines. These limits are likely to remain constant across a wide variety of implementations. However, memory limits will often be exceeded before architectural limits, such as the limit on the number of PDF objects, are reached.

Table D.1 *Architectural limits*

Quantity	*Limit*	*Explanation*
integer	2,147,483,647	Largest positive value, $2^{31} - 1$.
	–2,147,483,648	Largest negative value, -2^{31}.
real	±32,767	Approximate range of values.
	±1/65,536	Approximate smallest non-zero value.
	5	Approximate number of decimal digits of precision in fractional part.
array	65,535	Maximum number of elements in an array.
dictionary	65,535	Maximum number of key–value pairs in a dictionary.
string	65,535	Maximum number of characters in a string.
name	127	Maximum number of characters in a name.
indirect object	250,000	Maximum number of indirect objects in a PDF file.

Memory limits cannot be characterized so precisely, because the amount of available memory and the way in which it is allocated vary from one implementation 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 non-essential (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, 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.

Acrobat Exchange and Acrobat Reader have some additional architectural limits:

- Thumbnails may be no larger than 106×106 samples, and should be created at one-eighth scale for 8.5×11 inch and A4 size pages. Thumbnails should use either the **DeviceGray** or direct or indexed **DeviceRGB** color space.
- The minimum allowed page size is 1×1 inch (72×72 units in the default user space coordinate system), and the maximum allowed page size is 45×45 inches (3240×3240 units in the default user space coordinate system).
- The zoom factor of a view is constrained to be between 12% and 800%, regardless of the zoom factor specified in the PDF file.

- When Acrobat Exchange or Acrobat Reader 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 occur 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.

APPENDIX E

Obtaining XUIDs and Technical Notes

Creators of widely distributed forms who wish to use the XUID mechanism must obtain an organization ID from Adobe Systems Incorporated at the addresses listed below.

Technical notes, technical support, and periodic mailings are available to members of the Adobe Developers Association. In particular, the PostScript language software development kit (SDK) contains all the technical notes mentioned in this book. The Adobe Developers Association can be contacted at the addresses listed below:

Europe:
Adobe Developers Association
Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z-O
The Netherlands
Telephone: +44-131-458-6800
Fax: +44-131-458-6801

U.S. and the rest of the world:
Adobe Developers Association
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110
Telephone: (408) 536-9000

In addition, some technical notes and other information may be available from Adobe's World Wide Web server

```
http://www.adobe.com
```

and from an anonymous ftp site

```
ftp.adobe.com
```

When accessing the anonymous ftp site, use "anonymous" as the user name, and provide your E-mail address as the password (for example, smith@adobe.com).

APPENDIX F

PDF Name Registry

With the introduction of Adobe Acrobat 2.0, it has become easy for third parties to add private data to PDF documents and to add plug-ins 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. PDF 1.1

A PDF producer or Acrobat viewer plug-in may define new action, destination, annotation, and security handler types. If a user opens a PDF document and the plug-in that implements the new type of object is unavailable, the viewers will behave as described in Appendix , “Compatibility and Implementation Notes."

A PDF producer or Acrobat plug-in may also add keys to any PDF object that is implemented as a dictionary except the trailer dictionary. In addition, a PDF producer or Acrobat plug-in may create tags that indicate the role of Marked Content operators, as described in Section 8.10.3 on page 239. The names of such tags have additional requirements beyond those specified below. PDF 1.2

To avoid conflicts with third-party names and with future versions of PDF, Adobe maintains a registry, similar to the registry it maintains for Document Structuring Conventions. Third-party developers must only add private data that conforms to the registry rules. The registry includes three classes:

- *First-class* — Names and data of value to a wide range of developers. All the names defined in PDF 1.0 and 1.1 are first-class names. Plug-ins 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 data name and format for consideration as first-class, contact Adobe’s Developer Support group, as described later in this section.

- *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, ensuring that different developers’ second-class names will not conflict. It is up to each developer to ensure that they do not use the same name in conflicting ways themselves. To request a prefix for second-class names, contact Adobe’s Developer Support group, as described later in this section.

- *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-ins; this prefix is **XX**. This prefix must be used as the first characters in the names of all private data added by the developer. It it not necessary to contact Adobe to register third-class names.

Note *New keys for the Info dictionary in the Catalog and in Threads need not be registered.*

To register either first- or second-class names, contact Adobe's Developer Support group at (408) 536-9000, or send e-mail to

devsup-person@adobe.com

APPENDIX G

Compatibility and Implementation Notes

PDF 1.1

The goal of the Adobe Acrobat family of products is to enable people to exchange and view electronic documents easily and reliably. Ideally, that means that any Acrobat viewer 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. Of course, new versions of viewers are introduced to provide additional capabilities not present before. Furthermore, beginning with Acrobat 2.0, viewers may accept plug-in extensions, making some Acrobat 2.0 viewers more capable than others depending on what extensions are present. Both the viewers and PDF itself have been designed to enable users to view everything in the document that the viewer 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 specify how a viewer should behave when it reads a file that does not conform to the specification. This appendix provides this information. The PDF version number associated with a file determines how it should be treated when a viewer 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.

G.1 Version numbers

The PDF version number consists of a major and minor version. The version number is part of the PDF header, the first line of the file. This header takes the form:

%PDF-*M.m*

where *M* is the major number and *m* is the minor number.

If PDF changes in a way that current viewers will be unlikely to read a document without a serious error, the major version number will be incremented. A serious error is an error that prevents pages from being viewed.

If PDF changes in a way that a viewer will display an error message but continue its work, the minor version number will change. Adding new page description operators would require a change in the minor version number.

If PDF changes in a way that current viewers are unlikely to detect, the version number need not change.This includes the addition of private data that can be gracefully ignored by consumers that do not understand that data. An example is adding a key to a dictionary object such as the Catalog.

An Acrobat viewer will try to read any file with a valid PDF header, even if the version number is newer than the viewer itself. It will read without errors any file that does not require a plug-in, even if the version number is older than the viewer. 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. However, a plug-in is never required to display the contents of a page.

If a viewer opens a document with a newer major version number 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 notification that a command failed.) Processing continues if possible. Acrobat Exchange does not permit a document with a newer major version number to be inserted into another document.

If a viewer opens a document with a newer minor version number than it expects, it silently remembers the version number. Only if it encounters an error does it alert the user. At this point it notifies the user that the document is newer than expected, that an error has occurred, and that no further errors will be reported. The document may not be incrementally saved but can be saved to a new file. The saved file will continue to have the new version number. A user may insert a document with a newer minor version into another document. The resulting document can be saved. Its version number will be the maximum of the version number of the original document and the documents inserted into the original.

When opening a file, the Acrobat viewers are very liberal in their check for a valid PDF header. All viewers allow the header to appear anywhere in the first 1,000 bytes of the file. The 1.0 viewers require only that "**%PDF-**" appear in the header, but ignore the rest of the header. Subsequent viewers search for a header of the form described above. However, they also accept a header of the form:

%!PS-Adobe-*N.n* PDF-*M.m*

where *N.n* is an Adobe Document Structuring Conventions version number and *M.m* is a PDF version number. (The *PostScript Language Reference Manual* describes the Document Structuring Conventions).

G.2 Dictionary keys

Adding key-value pairs not described in the PDF specification to dictionary objects usually does not affect the behavior of 1.0 viewers and never affects the behavior of subsequent viewers. These keys are ignored. If a dictionary object such as an annotation is copied into another document during a page insertion (or in Acrobat

2.0 and 3.0 viewers during a page extraction), all key-value pairs are copied. If a value is an indirect reference to another object, that object may be copied as well, depending on the key.

In some cases a 1.0 viewer displays an error if it finds an unknown key in a dictionary. These cases are keys in image dictionaries (both XObjects and in-line images) and keys in DecodeParms dictionaries for filters.

See Appendix F for information on how to choose key names that are compatible with future versions of PDF.

G.3 Implementation notes

The following sections give details of the implementation of Adobe Acrobat. They are presented in the same order in which they appear in the main text.

Section 4.4, “Strings”

All Acrobat viewers can read strings that include non-printable ASCII.

Section 7.2, “Date”

Acrobat 1.0 viewers report date strings as ordinary strings. Later viewers report date strings as dates when used as the value of the **CreationDate** or **ModDate** in the Info dictionary or as the value of the **Date** key in annotations. The viewers ignore the GMT information.

Section 4.5, “Names”

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)`.

Section 4.8, “Streams” (Filters)

PDF uses stream objects to encapsulate the data in images, indexed color spaces, thumbnails, and many other objects. These streams usually use filters to compress their data. The legal PDF 1.0 filters are the same as those available in PostScript Level 2. The 1.0 viewer behavior when encountering an unknown filter depends on its context, as described in Table G.1.

Table G.1 *Acrobat 1.0 Viewer behavior with unknown filters*

Context	*Behavior*
Image resource	The image does not appear but no error is reported.

In-line image	(An in-line image is specified directly in a page description, while an image resource is specified outside of a page and referenced from the page.) An error is reported, and page processing stops.
Indexed color space	An error is reported, but page processing continues.
Thumbnail	An error is reported, no more thumbnails are displayed, but the thumbnails can be deleted and created again.
Embedded font	An error is reported, and the viewer behaves as if the font is not embedded.
Page description	An error is reported, and page processing stops.
Form description	An error is reported, and page processing stops.
Type 3 character description	An error is reported, and page processing stops.

Subsequent viewers do not allow plug-ins to provide additional filters. If an unrecognized filter is encountered, these viewers specify the context in which the filter was found. If an error occurs while displaying a page, only the first error is reported. Subsequent behavior depends on the context, as described in Table G.2.

Table G.2 *Acrobat 2.0 Viewer behavior with unknown filters*

Context	*Behavior*
Image resource	The image does not appear but page processing continues.
In-line image	Page processing stops.
Indexed color space	The image does not appear but page processing continues.
Thumbnail	An error is reported, no more thumbnails are displayed, but the thumbnails can be deleted and created again.
Embedded font	The viewer behaves as if the font had not been embedded.
Page description	Page processing stops.
Form description	The form does not appear but page processing continues.
Type 3 character description	The character does not appear but page processing continues. The current point is adjusted based on the character's width.

Operations that process pages, such as Find and Create Thumbnails, stop as soon as an error occurs.

Versions of the Acrobat viewer prior to 3.0 do not understand the **FlateDecode** filter. They display an error message indicating that they are unable to process a page.

When a stream specifies an external file, PDF 1.0 and PDF 1.1 parsers ignore the file and always use the characters between **stream** and **endstream**.

Section 5.1, “PDF files”

The Acrobat 1.0 viewers successfully read files that contain binary data. The restriction on line length is not enforced by any Acrobat viewer.

The Acrobat 1.0 products on the Apple® Macintosh® computer create files with type **'TEXT'**. Later Acrobat products create files with type **'PDF '**. A user can open these documents from a 1.0 viewer but not from the Finder.

Section 5.7, “Encryption”

On opening a protected document, a version 1.0 Acrobat viewer displays a blank page or reports that an error was found while processing a page. Subsequent viewers report that a plug-in is required to open the document if the security handler for the document is not available.

Section 6.4, “Page objects”

Acrobat viewers rebuild the Beads array for all pages of a document containing beads if the first page with a bead does not have a Beads array.

Section 6.6, “Annotations”

An annotation is a dictionary element of a page’s **Annots** array. Its **Subtype** specifies the type of annotation. Only **Text** and **Link** are defined by PDF 1.0. If a 1.0 viewer reads a page with an annotation whose **Subtype** is not **Text** or **Link**, it displays an error. It displays one error per page no matter how many annotations are present.

Subsequent viewers display unknown annotations in a closed form similar to text annotations, with an icon containing a question mark. If the user opens the annotation, an alert appears with a message giving the annotation type and explaining that an unavailable plug-in is required to open it. An unknown annotation can be selected, moved, and deleted. Every annotation type must specify its position and size using the **Rect** key.

Acrobat viewers ignore the first two numbers in the **Border** array of an annotation.

Acrobat viewers support a maximum of 10 entries in the dash array (the fourth element of the **Border** array).

Acrobat viewers update the **ModDate** string only for text annotations.

If an Acrobat viewer encounters an annotation type it does not understand (i.e., for which there is no annotation handler), the viewer displays it as an unknown annotation unless the annotation's **F** (Flags) key specifies that the "invisible" flag is set. The **C**, **T**, **M**, and **F** keys are ignored by Acrobat 1.0 viewers. The **H**, **BS**, **AP**, and **AS** keys are ignored by Acrobat 2.0 and 2.1 viewers.

The use of the *Hidden* and *Print* flags (bits 2 and 3) in an annotation has no effect on Acrobat viewers prior to 3.0. Annotations that should be hidden are shown; annotations that should be printed are not printed.

In version 3.0 of Acrobat, neither text annotations nor link annotations respect the *Print* flag.

Section 6.6.1, "Annotation borders"

Acrobat viewers prior to 3.0 ignore the **BS** key.

If an Acrobat 3.0 viewer encounters a border subtype it doesn't recognize, the border defaults to **S** (solid).

Section 6.6.2, "Annotation highlighting"

Acrobat viewers prior to 3.0 ignore highlighting modes. The Acrobat 3.0 viewer uses highlighting modes only for **Link** and **Widget** annotations.

Section 6.6.3, "Annotation appearances"

The presence of an appearance dictionary has no effect in versions of Acrobat prior to 3.0. The additional functionality provided by this construct is not available in older viewers.

Section 6.6.5, "Link annotations"

Acrobat 1.0 viewers do not report an error when a user activates a link or outline entry that has an unknown destination type or is missing a destination. Links and outline entries with an **A** key will appear to have no destination. Subsequent viewers report an error when the destination or action type is unknown.

Section 6.6.6, "Movie Player annotations"

Acrobat 1.0 viewers report the following error when they encounter an annotation of type **Movie**: "An error occurred while reading a note or link. Unknown annotation type." The annotation does not appear on the document. Subsequent viewers report the following error when they encounter an annotation of type **Movie**: "The Plug-in required by this 'Movie' annotation is unavailable." The annotation is displayed as a gray rectangle with a question mark.

Section 6.8, "Destinations"

A link or a bookmark in PDF 1.0 is a dictionary that contains a **Dest** key whose value specifies a view of the document that is displayed when the link or bookmark is activated. A destination is an array. Its first element is a name that serves as destination type; it determines the interpretation of subsequent array elements. If a

1.0 viewer encounters an unknown destination type, no action is performed and no error is reported when the user activates the link or bookmark. Subsequent viewers display a message when they find an unknown destination type.

An Acrobat 1.0 viewer does nothing if it does not find a **Dest** key in a link or bookmark.

Section 6.9, “Actions”

Action have superseded destinations in PDF 1.1. An Acrobat 1.0 viewer ignores actions.

The existence of the **Next** key in an action dictionary has no effect in Acrobat viewers prior to version 3.0.

Section 6.9.1, “Action Trigger Points”

The existence of an Additional Actions dictionary in an annotation, page, outline, or document has no effect in Acrobat viewers prior to 3.0.

In Acrobat 3.0, the **Open** and **Close** actions are disabled if the display is not in a page-oriented mode (e.g., if it is in multi-column mode). This prevents actions for multiple pages from being executed simultaneously, which would be confusing to the user.

Section 6.9.3, “GoToR action”

The **NewWindow** attribute is ignored by Acrobat viewers prior to 3.0.

Section 6.9.4, “Launch action”

Some implementations of Acrobat viewers may check for alternative keys whose values provide platform-specific parameters for the **Launch** action. For example, the Acrobat viewer for Windows uses the dictionary corresponding to the **Win** key to determine its launch parameters.

The Acrobat viewer for Windows use the Windows function `ShellExecute` to launch an application. The **Win** dictionary entries correspond to the parameters of `ShellExecute`.

The **NewWindow** attribute is ignored by Acrobat viewers prior to 3.0.

Section 6.9.6, “URI action”

Acrobat 1.0 viewers report no error when a link annotation that uses the URI action is invoked. The link inverts its color and performs no action. Subsequent viewers report the following error when a link annotation that uses the URI action is invoked: “The plug-in required by this URI action is not available.”

When resolving the fragment identifier, the WebLink plug-in checks all named destinations defined for the document. If one is found whose name matches the fragment identifier, that destination is invoked.

Section 6.9.8, “Movie Player actions”

Acrobat viewers prior to version 3.0 report an error when they encounter an action of type **Movie**.

Section 6.9.9, “SetState action”

Acrobat viewers prior to 3.0 report the following error when encountering an action of type **SetState**: “The plug-in needed for this SetState action is not available.”

In Acrobat Viewer 3.0 the effect of a **SetState** action is temporary in nature in the same manner that opening or closing text annotations is temporary. If the user saves the document, the changes become permanent. Otherwise, the user is not prompted to save the document and the change in state is not permanent.

Section 6.9.10, “Hide action”

Acrobat viewers prior to 3.0 report the following error when encountering an action of type **Hide**: “The plug-in needed for this Hide action is not available.”

In Acrobat 3.0, the effect of the Hide action is temporary in nature in the same manner that opening or closing text annotations is temporary. If the user saves the document, the changes become permanent. Otherwise, the user is not prompted to save the document and the hiding/showing of the annotation is not permanent.

Section 6.9.11, “Named actions”

Acrobat viewers prior to 3.0 report the following error when encountering an action of type **Named**: “The plug-in needed for this Named action is not available.”

The Acrobat 3.0 viewer extends the list of Named Actions in Table 6.36 on page 106 by allowing most viewer menu item names to be specified. For further details, see the listing of menu item names in the Acrobat Plug-In Developer’s SDK.

Section 6.9.12, “NOP action”

Acrobat 1.0 viewers ignore all actions, including NOP. If an Acrobat 2.0 viewer attempts to perform a NOP action, it displays a warning that says that the plug-in required for this action is not present. It is unlikely, however, that such a warning would occur, as this type of action is defined only in places where behavior may be inherited, and there are no such places in Acrobat 2.0.

Section 6.12, “Articles”

The thread array and dictionary objects are invisible to 1.0 viewers. Consequently, operations that insert or delete pages do not carry along any threads.

Section 6.13, “File ID”

Although the **ID** key is not required, all Adobe applications that produce PDF include this key. Acrobat Exchange adds this key when saving a file if it is not present.

Section 6.14, “Encryption dictionary”

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 6.4.

Section 6.15.4, “Field dictionaries”

In Acrobat 3.0, partial field names may not contain a period.

Section 6.15.11, “Choice”

In Acrobat 3.0, the **Opt** array in a Choice field (see Section 6.15.11 on page 126) must be homogenous: the elements must be either all strings or all arrays.

Section 6.15.13.1, “SubmitForm Action”

In Acrobat 3.0, if the response to a **SubmitForm** action uses Forms Data Format, then the URL must end in `#FDF` so that it is recognized as such by the Acrobat software and handled properly. Conversely, if the response is anything else, then the URL should not end in `#FDF`.

Section 6.15.13.3, “ImportData Action”

Acrobat 3.0 puts a relative file specification (of the FDF with respect to the Form) as the value of **F**. When performing the action, if the FDF is not found, then Acrobat 3.0 tries to locate the file in a few “well-known” platform-dependent locations. For example, on the Windows platform, it looks in the directory from which Acrobat loaded, the current directory, the System directory, the Windows directory, and the directories that are listed in the PATH environment variable; on the Macintosh, it looks in the Preferences folder, and in the Acrobat folder.

When executing the **ImportData** action, Acrobat 3.0 imports the FDF into the current Form, ignoring the **F** and **ID** keys inside the FDF.

Section 7.5, “ProcSets”

Each page includes a ProcSet resource that describe the PostScript procedure sets required to print the page. A 1.0 viewer ignores requests for unknown procedure sets. An Acrobat 2.0 viewer warns the user that a procedure set is unavailable and cancels printing.

Section 7.6, “Fonts”

All Acrobat viewers ignore the **Name** key in a Font resource.

Section 7.6.6, “Type 3 fonts”

For compatibility with Acrobat 1.0, 2.0, and 2.1, the names of resources in a Type 3 font’s Resources dictionary must match the names of resources in the Page’s Resources dictionary. If backwards compatibility is not required, then any valid names may be used.

Section 7.9, “Font descriptors”

Acrobat viewers prior to version 3.0 ignore the **FontFile3** value. If a font uses the Adobe Standard Roman Character CharSet, then 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.

Section 7.9.1, “Font files”

Embedded TrueType fonts are ignored by Acrobat 1.0 viewers.

Section 7.10, “Color spaces”

An image has a **ColorSpace** key. A 1.0 viewer displays an error each time it finds an image with a color space that is not one of the PDF 1.0 color spaces. Color spaces may not be added by plug-ins. If an Acrobat 2.0 viewer encounters an unknown color space, such as the special color spaces defined in PDF 1.2 (**Pattern**, **Separation**, and some uses of **Indexed**), it will be in a document with a PDF version number greater than 1.1. The viewer displays an error specifying the type of color space but it reports no further errors.

PDF 1.1 defines three additional color spaces: **CalGray**, **CalRGB**, and **Lab**. To be more compatible with 1.0 viewers, PDF 1.1 allows an image color space to be specified indirectly through the Resources dictionary. When an Acrobat 2.0 viewer processes an image and the image’s **ColorSpace** key specifies **DeviceRGB**, the viewer looks in the page’s resources for a color space called **DefaultRGB**. If this key is present, the color space associated with it is used instead of **DeviceRGB**. Similarly, if an image’s **ColorSpace** key specifies **DeviceGray**, the viewer looks for **DefaultGray**. The 1.0 viewer ignores **DefaultRGB** and **DefaultGray**.

See page 169 for an explanation of the use of color spaces in page descriptions. The presence of **DefaultRGB** or **DefaultGray** change the interpretation of some color operators.

Acrobat viewers allow a user to approximate device-independent colors on screen with device-dependent colors with no transformation. **CalGray** colors are viewed as **DeviceGray**, and **CalRGB** colors are viewed as **DeviceRGB**.

Section 7.10.9, “Separation color spaces”

The Acrobat 3.0 viewer applies the *tintTransform* function, as specified in this section, for displaying graphics that use separation color spaces.

Section 7.11, “XObjects”

An XObject is a stream or dictionary that is referred to by name from a page description by the **Do** operator. The effect of the operator is determined by the type of the XObject. A 1.0 viewer displays an error for each XObject of a different type, no matter how many are on a page.

Plug-ins may not add XObject types, since they are considered part of the page and a viewer without plug-ins should always be able to display a page. If an Acrobat 2.0 viewer encounters an unknown XObject type, it will be in a document with a PDF version number greater than 1.1. The viewer displays an error specifying the type of XObject but it reports no further errors.

Section 7.11.1, “Images”

The **Name** key in an Image resource is ignored by all Acrobat viewers.

Section 7.11.3, “Color rendering intent”

The Acrobat 1.0 viewers display an error if an image specifies an **Intent**.

Because of the large gamut of most displays, Acrobat viewers ignore the **Intent** key when displaying a PDF file and always use **RelativeColorimetric**. When printing to a PostScript printer, the Acrobat viewers do not specify an intent unless one was explicitly specified.

Section 7.11.6, “OPI dictionary”

The Acrobat 3.0 Distiller program converts OPI comments into OPI dictionaries, and when the Acrobat 3.0 Viewer prints a PDF file to a PostScript file or printer, it converts the OPI dictionary to OPI comments. However, the OPI information has no effect on the displayed XObject (form or image).

In Acrobat 3.0, the value of the **F** key in an OPI dictionary must be a string.

The Acrobat 3.0 Distiller application and the Acrobat 3.0 Viewer do not support OPI 2.0.

Section 7.12.1, “Sampled functions (Function Type 0)”

Acrobat 3.0 supports only linear interpolation (**Order** 1).

Section 7.14.2, “Predefined spot functions”

If the Acrobat 3.0 Distiller encounters a call to **setscreen** or **sethalftone**, and if that call includes a spot function, the Distiller examines the code for the spot function. If the code matches the PostScript shown in Table 7.35 on page 193, then the Distiller puts the corresponding name in the halftone dictionary, and Acrobat uses that PostScript code when printing the PDF file to a PostScript printer. Otherwise, the Distiller samples the spot function and generate a function for the halftone dictionary; when printing to a PostScript printer, Acrobat generates a spot function that interpolates values from that function.

Section 7.15, “Patterns”

The Acrobat 3.0 does not display patterns on the screen, but it does print them to PostScript.

Chapter 8, “Page Descriptions”

A 1.0 viewer reports an error the first time it finds an unknown operator or an operator with too few operands, but it continues processing the marking context (e.g., page or form). If it finds ten errors on a page, it reports back to the user and asks whether to continue processing. No further errors are reported. Each time an error occurs, the operand stack is cleared. Later Acrobat viewers behave the same, although there is no additional warning if ten errors are encountered.

PDF 1.1 provides new page description operators for specifying device-independent color and pass-through PostScript fragments. Since these operators are incompatible with 1.0 viewers, PDF 1.1 provides alternative compatible methods as well.

PDF 1.2 defines new operators for setting parameters in the graphics state (gs) and for setting the color in **Pattern** and **Separation** color spaces. There is no compatible mechanism for these operators in viewers prior to Acrobat 3.0.

Section 8.5.2, “Color operators”

For compatibility with PDF 1.0 viewers, it is recommended that device-dependent colors be specified using the 1.0 operators and that device-independent colors be specified using the color space substitution method defined in Section 7.10.10 on page 175.

If an Acrobat 1.0 viewer reads a page containing any of the setcolorspace, setcolor, or intent operators (**cs**, **CS**, **sc**, **SC**, **scn**, **SCN**, or **ri**), it reports an error. Errors can be ignored by the user and objects are displayed, but colors will most likely be black (the default).

The **scn** and **SCN** operators are not compatible with versions of Acrobat prior to 3.0.

Section 8.7.5, “Text string operators”

In versions of Acrobat prior to 3.0, when using the **TJ** operator, the x-coordinate of the current point after drawing a character and moving by any specified offset must not be less than the x-coordinate of the current point before the character was drawn.

Section 8.8.1, “XObject operators”

If an Acrobat 1.0 viewer reads a page containing the **PS** operator, it reports an error. The operator is otherwise ignored.

Section 8.10.2, “Compatibility operators”

If an Acrobat 1.0 viewer reads a page containing the **BX** or **EX** operators, it reports an error. The operators are otherwise ignored.

Section 9.3.2, “Header and linearization information”

In the Acrobat 3.0 viewer, linearization is an option that is available when a PDF file is saved. It rewrites the entire file (a “full save”) and always uses version 1.2 in the header: `%PDF-1.2`.

APPENDIX H

Forms Data Format

PDF 1.2

This appendix describes FDF, the file format used for Acrobat Forms data, a new feature of PDF 1.2. FDF is used when submitting Form data to a server, receiving the response, and incorporating it into the Form. It can also be used to generate (i.e. "export") stand-alone files containing Form data that can be stored, transmitted electronically (e.g., via e-mail), and imported back into the corresponding Form.

FDF is based on PDF, and uses the same syntax and set of basic object types as PDF. It also has the same file structure, except for the fact that the cross-reference is optional. The document structure is much simpler than PDF, since the body of an FDF document consists of only one required object. Objects in FDF can only be of generation 0; no two objects can have the same object number, and FDF files cannot have updates appended to them. The value of the **Length** attribute in the dictionary of any stream object appearing inside an FDF document must be a direct object.

FDF uses the MIME type `application/vnd.fdf`. On Windows and Unix it uses the `*.fdf` extension, and on the Mac it has the `'FDF '` file type.

H.1 File Structure

An FDF file consists of a one-line header, a body, and a trailer. It can optionally contain a cross-reference table. In other words, FDF is structured in the same way as PDF, but need only contain those elements required for Acrobat Forms data export and import, which are described below.

H.1.1 Header

The first line of an FDF file specifies the version number of the PDF specification that FDF is a part of. The current version of PDF is 1.2; therefore the first line of an FDF file is

```
%FDF-1.2
```

H.1.2 Body

The body consists of one Catalog object and any additional indirect objects that it may reference. The Catalog object is a dictionary with only one (required) key in it, **FDF**. Its value is a dictionary, whose entries are described in Section H.2, "The FDF Catalog Object".

It is legal for the body to contain additional objects, and for the Catalog object to contain additional key/value pairs. Comments can appear anywhere in the body section of an FDF file.

Just as in PDF, objects in FDF can be direct or indirect.

H.1.3 Trailer

The trailer consists of a trailer dictionary, followed by the last line of the FDF file, containing the end-of-file marker, **%%EOF**. The trailer dictionary consists of the keyword **trailer**, followed by at least one key/value pair enclosed in double angle brackets. The only required key is **Root**, and its value is an indirect reference to the Catalog object in the FDF body.

It is legal for the trailer dictionary to contain the additional key/value pairs described in the PDF specification.

H.2 The FDF Catalog Object

The value of the **FDF** key in the Catalog object is a dictionary, whose entries are described in the following table:

Table H.1 *FDF attributes*

Key	*Type*	*Semantics*
Fields	array	*(Optional)* This array contains the root fields being exported or imported. A root field is one with no parent (i.e. it is not in the **Kids** array of another field). The attributes of the fields are described in Section 6.15.4 on page 119.
Status	string	*(Optional)* A status to be displayed indicating the result of an action, typically a SubmitForm action (see Section 6.15.13.1 on page 129). This string is encoded with **PDFDocEncoding**.
Implementation note		*The Acrobat 3.0 implementation of Forms displays the* **Status***, if any, in an Alert Note, when importing an FDF.*
F	File specification	*(Optional)* File specification for the Acrobat Form that this FDF was exported from, or is meant to be imported into.
ID	array	*(Optional)* The value of the **ID** field in the trailer dictionary of the Acrobat Form that this FDF was exported from, or is meant to be imported into.

Table H.2 describes the attributes of each field in the FDF. The majority of the attributes described in this table represent the same information and have the same semantics as the attributes of the same name described in Section 6.15.4 on page 119.

Table H.2 *Field attributes*

Key	*Type*	*Semantics*
T	string	*(Required)* The partial field name.
Kids	array	*(Optional)* Contains the child field dictionaries.
V	*various*	*(Optional)* Field value.
Opt	array	*(Optional)* Options.
Ff	integer	*(Optional)* Field flags. When imported into an Acrobat Form, it replaces the current value of the **Ff** key in the corresponding field inside the Form. If **SetFf** and/or **ClrFf** are also present, they are ignored.
SetFf	integer	*(Optional)* Field flags. When imported into an Acrobat Form, it is OR'ed with the current value of the **Ff** key in the corresponding field inside the Form.
ClrFf	integer	*(Optional)* Field flags. When imported into an Acrobat Form, for each bit that is set to one in this value, sets the corresponding bit in the Form field's **Ff** flags to zero. If **SetFf** is also present, **ClrFf** is applied after **SetFf**.
F	integer	*(Optional)* Widget annotation flags. When imported into an Acrobat Form, it replaces the current value of the **F** key in the corresponding field inside the Form. If **SetF** and/or **ClrF** are also present, they are ignored.
SetF	integer	*(Optional)* Widget annotation flags. When imported into an Acrobat Form, it is OR'ed with the current value of the **F** key in the corresponding field inside the Form.
ClrF	integer	*(Optional)* Widget annotation flags. When imported into an Acrobat Form, for each bit that is set to one in this value, sets the corresponding bit in the Form's **F** flags to zero. If **SetF** is also present, **ClrF** is applied after **SetF**.
AP	dictionary	*(Optional)* Appearance of the Widget annotation.
AS	name	*(Optional)* Appearance state.
A	dictionary	*(Optional)* Action to be performed on activation of this Widget annotation.
AA	dictionary	*(Optional)* Additional actions.

H.3 Use of FDF

For most of the keys, unless otherwise indicated in Table H.2, importing consists of taking the value of each key as received in the FDF, and using it to replace the value of the corresponding key in the field inside the Form with the same fully qualified name.

Implementation note *Of all the possible keys shown in Table H.2, the 3.0 version of Acrobat will only export the* **V** *key of a field when generating FDF. It will, however, import FDF files containing fields using any of the described keys.*

Implementation note *Acrobat 3.0, when importing an FDF that came back as a result of a SubmitForm action, if the Form currently being displayed is not the one specified in the* **F** *key of the* **FDF** *dictionary (which, as stated in Table H.1, is an optional key), then that Form is first fetched, and then the FDF gets imported.*

Implementation note *When exporting FDF, Acrobat 3.0 computes a relative path from the location the FDF is being stored, to the location the Form is in, and uses that as the value of the* **F** *key in the FDF dictionary.*

Implementation note *Under Acrobat 3.0, if an FDF being imported contains fields whose fully qualified names are not present in the Form, those fields will be discarded. This feature can be useful, among other cases, if an FDF containing commonly used fields (such as name, address, etc.) is used to populate various types of Acrobat Forms, each of which does not necessarily include all the fields available in the FDF.*

Note *As shown in Table H.2, the only required key in the field dictionary is* **T**. *One possible use for exporting FDF with fields that only contain* **T** *but no* **V**, *is as an indication to a server of which fields are desired in the FDF coming back as a response. For example, a server accessing a database might choose between sending all fields in a record, vs. just some selected ones, based on the use of this feature in the request FDF. The implementation of Acrobat Forms will ignore, during import, fields in the FDF that do not exist in the Form.*

Implementation note *The Acrobat 3.0 implementation of Forms allows the choice for a SubmitForm action to send the data using HTML 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 as a response will need to be modified to generate FDF instead.*

H.4 Sample FDF

```
%FDF-1.2
1 0 obj <<
/FDF <<
/Fields
[
<<
/T (My Children)
/V (Tali)
/Opt [(Maya) (Adam) (Tali)]
```

```
>>
]
/F (Dependents.pdf)
>>
>>
endobj
trailer
<</Root 1 0 R>>
%%EOF
```

APPENDIX I

ISO 639 Language Codes

code	language
aa	Afar
ab	Abkhazian
af	Afrikaans
am	Amharic
ar	Arabic
as	Assamese
ay	Aymara
az	Azerbaijani
ba	Bashkir
be	Byelorussian
bg	Bulgarian
bh	Bihari
bi	Bislama
bn	Bengali; Bangla
bo	Tibetan
br	Breton
ca	Catalan
co	Corsican
cs	Czech
cy	Welsh
da	Danish
de	German
dz	Bhutani
el	Greek
en	English
eo	Esperanto
es	Spanish
et	Estonian
eu	Basque
fa	Persian
fi	Finnish
fj	Fiji
fo	Faeroese
fr	French
fy	Frisian
ga	Irish
gd	Scots Gaelic
gl	Galician
gn	Guarani
gu	Gujarati
ha	Hausa
hi	Hindi
hr	Croatian
hu	Hungarian
hy	Armenian
ia	Interlingua
ie	Interlingue
ik	Inupiak
in	Indonesian
is	Icelandic
it	Italian
iw	Hebrew
ja	Japanese
ji	Yiddish
jw	Javanese

code	language
ka	Georgian
kk	Kazakh
kl	Greenlandic
km	Cambodian
kn	Kannada
ko	Korean
ks	Kashmiri
ku	Kurdish
ky	Kirghiz
la	Latin
ln	Lingala
lo	Laothian
lt	Lithuanian
lv	Latvian, Lettish
mg	Malagasy
mi	Maori
mk	Macedonian
ml	Malayalam
mn	Mongolian
mo	Moldavian
mr	Marathi
ms	Malay
mt	Maltese
my	Burmese
na	Nauru
ne	Nepali

code	language
nl	Dutch
no	Norwegian
oc	Occitan
om	(Afan) Oromo
or	Oriya
pa	Punjabi
pl	Polish
ps	Pashto, Pushto
pt	Portuguese
qu	Quechua
rm	Rhaeto-Romance
rn	Kirundi
ro	Romanian
ru	Russian
rw	Kinyarwanda
sa	Sanskrit
sd	Sindhi
sg	Sangro
sh	Serbo-Croatian
si	Singhalese
sk	Slovak
sl	Slovenian
sm	Samoan
sn	Shona

code	language
so	Somali
sq	Albanian
sr	Serbian
ss	Siswati
st	Sesotho
su	Sundanese
sv	Swedish
sw	Swahili
ta	Tamil
te	Tegulu
tg	Tajik
th	Thai
ti	Tigrinya
tk	Turkmen
tl	Tagalog
tn	Setswanato Tonga
tr	Turkish
ts	Tsonga
tt	Tatar
tw	Twi
uk	Ukrainian
ur	Urdu
uz	Uzbek
vi	Vietnamese
vo	Volapuk

code	language
wo	Wolof
xh	Xhosa
yo	Yoruba
zh	Chinese
zu	Zulu

APPENDIX J

ISO 3166 Country Codes

country	code
AFGHANISTAN	AF
ALBANIA	AL
ALGERIA	DZ
AMERICAN SAMOA	AS
ANDORRA	AD
ANGOLA	AO
ANGUILLA	AI
ANTARCTICA	AQ
ANTIGUA AND BARBUDA	AG
ARGENTINA	AR
ARMENIA	AM
ARUBA	AW
AUSTRALIA	AU
AUSTRIA	AT
AZERBAIJAN	AZ
BAHAMAS	BS
BAHRAIN	BH
BANGLADESH	BD
BARBADOS	BB
BELARUS	BY
BELGIUM	BE
BELIZE	BZ
BENIN	BJ
BERMUDA	BM
BHUTAN	BT
BOLIVIA	BO
BOSNIA AND HERZEGOWINA	BA
BOTSWANA	BW
BOUVET ISLAND	BV
BRAZIL	BR
BRITISH INDIAN OCEAN TERRITORY	IO
BRUNEI DARUSSALAM	BN
BULGARIA	BG
BURKINA FASO	BF
BURUNDI	BI
CAMBODIA	KH
CAMEROON	CM
CANADA	CA
CAPE VERDE	CV
CAYMAN ISLANDS	KY
CENTRAL AFRICAN REPUBLIC	CF
CHAD	TD
CHILE	CL

country	code
CHINA	CN
CHRISTMAS ISLAND	CX
COCOS (KEELING) ISLANDS	CC
COLOMBIA	CO
COMOROS	KM
CONGO	CG
COOK ISLANDS	CK
COSTA RICA	CR
COTE D'IVOIRE	CI
CROATIA (local name: Hrvatska)	HR
CUBA	CU
CYPRUS	CY
CZECH REPUBLIC	CZ
DENMARK	DK
DJIBOUTI	DJ
DOMINICA	DM
DOMINICAN REPUBLIC	DO
EAST TIMOR	TP
ECUADOR	EC
EGYPT	EG
EL SALVADOR	SV
EQUATORIAL GUINEA	GQ
ERITREA	ER
ESTONIA	EE
ETHIOPIA	ET
FALKLAND ISLANDS (MALVINAS)	FK
FAROE ISLANDS	FO
FIJI	FJ
FINLAND	FI
FRANCE	FR

country	code
FRANCE, METROPOLITAN	FX
FRENCH GUIANA	GF
FRENCH POLYNESIA	PF
FRENCH SOUTHERN TERRITORIES	TF
GABON	GA
GAMBIA	GM
GEORGIA	GE
GERMANY	DE
GHANA	GH
GIBRALTAR	GI
GREECE	GR
GREENLAND	GL
GRENADA	GD
GUADELOUPE	GP
GUAM	GU
GUATEMALA	GT
GUINEA	GN
GUINEA-BISSAU	GW
GUYANA	GY
HAITI	HT
HEARD AND MC DONALD ISLANDS	HM
HONDURAS	HN
HONG KONG	HK
HUNGARY	HU
ICELAND	IS
INDIA	IN
INDONESIA	ID
IRAN (ISLAMIC REPUBLIC OF)	IR
IRAQ	IQ
IRELAND	IE

country	code
ISRAEL	IL
ITALY	IT
JAMAICA	JM
JAPAN	JP
JORDAN	JO
KAZAKHSTAN	KZ
KENYA	KE
KIRIBATI	KI
KOREA, DEMOCRATIC PEOPLE'S REPUBLIC OF	KP
KOREA, REPUBLIC OF	KR
KUWAIT	KW
KYRGYZSTAN	KG
LAO PEOPLE'S DEMOCRATIC REPUBLIC	LA
LATVIA	LV
LEBANON	LB
LESOTHO	LS
LIBERIA	LR
LIBYAN ARAB JAMAHIRIYA	LY
LIECHTENSTEIN	LI
LITHUANIA	LT
LUXEMBOURG	LU
MACAU	MO
MACEDONIA, THE FORMER YUGO-SLAV REPUBLIC OF	MK
MADAGASCAR	MG
MALAWI	MW
MALAYSIA	MY
MALDIVES	MV
MALI	ML

country	code
MALTA	MT
MARSHALL ISLANDS	MH
MARTINIQUE	MQ
MAURITANIA	MR
MAURITIUS	MU
MAYOTTE	YT
MEXICO	MX
MICRONESIA, FEDERATED STATES OF	FM
MOLDOVA, REPUBLIC OF	MD
MONACO	MC
MONGOLIA	MN
MONTSERRAT	MS
MOROCCO	MA
MOZAMBIQUE	MZ
MYANMAR	MM
NAMIBIA	NA
NAURU	NR
NEPAL	NP
NETHERLANDS	NL
NETHERLANDS ANTILLES	AN
NEW CALEDONIA	NC
NEW ZEALAND	NZ
NICARAGUA	NI
NIGER	NE
NIGERIA	NG
NIUE	NU
NORFOLK ISLAND	NF
NORTHERN MARIANA ISLANDS	MP
NORWAY	NO

country	code
OMAN	OM
PAKISTAN	PK
PALAU	PW
PANAMA	PA
PAPUA NEW GUINEA	PG
PARAGUAY	PY
PERU	PE
PHILIPPINES	PH
PITCAIRN	PN
POLAND	PL
PORTUGAL	PT
PUERTO RICO	PR
QATAR	QA
REUNION	RE
ROMANIA	RO
RUSSIAN FEDERATION	RU
RWANDA	RW
SAINT KITTS AND NEVIS	KN
SAINT LUCIA	LC
SAINT VINCENT AND THE GRENA-DINES	VC
SAMOA	WS
SAN MARINO	SM
SAO TOME AND PRINCIPE	ST
SAUDI ARABIA	SA
SENEGAL	SN
SEYCHELLES	SC
SIERRA LEONE	SL
SINGAPORE	SG
SLOVAKIA (Slovak Republic)	SK

country	code
SLOVENIA	SI
SOLOMON ISLANDS	SB
SOMALIA	SO
SOUTH AFRICA	ZA
SPAIN	ES
SRI LANKA	LK
ST. HELENA	SH
ST. PIERRE AND MIQUELON	PM
SUDAN	SD
SURINAME	SR
SVALBARD AND JAN MAYEN ISLANDS	SJ
SWAZILAND	SZ
SWEDEN	SE
SWITZERLAND	CH
SYRIAN ARAB REPUBLIC	SY
TAIWAN, PROVINCE OF CHINA	TW
TAJIKISTAN	TJ
TANZANIA, UNITED REPUBLIC OF	TZ
THAILAND	TH
TOGO	TG
TOKELAU	TK
TONGA	TO
TRINIDAD AND TOBAGO	TT
TUNISIA	TN
TURKEY	TR
TURKMENISTAN	TM
TURKS AND CAICOS ISLANDS	TC
TUVALU	TV
UGANDA	UG

country	code
UKRAINE	UA
UNITED ARAB EMIRATES	AE
UNITED KINGDOM	GB
UNITED STATES	US
UNITED STATES MINOR OUTLYING ISLANDS	UM
URUGUAY	UY
UZBEKISTAN	UZ
VANUATU	VU
VATICAN CITY STATE (HOLY SEE)	VA
VENEZUELA	VE
VIET NAM	VN
VIRGIN ISLANDS (BRITISH)	VG
VIRGIN ISLANDS (U.S.)	VI
WALLIS AND FUTUNA ISLANDS	WF
WESTERN SAHARA	EH
YEMEN	YE
YUGOSLAVIA	YU
ZAIRE	ZR
ZAMBIA	ZM
ZIMBABWE	ZW

Bibliography

Note *Adobe technical notes are available online at the following URL: http://www.adobe.com/supportservice/devrelations/technotes.html*

[1] Adobe Systems Incorporated, *PostScript Language Reference Manual, Second Edition*, Addison-Wesley, 1990, ISBN 0-201-10174-2. Reference manual describing the imaging model used in the PostScript language and the language itself.

[2] Adobe Systems Incorporated, *PostScript Language Reference Manual Supplement For Version 2016*, 7 July 1995. This describes changes to the PostScript language as of version 2016.

[3] Adobe Systems Incorporated, *Supporting Data Compression in PostScript Level 2 and the Filter Operator*, Adobe Developer Support Technical Note 5115.

[4] Adobe Systems Incorporated, *Supporting the DCT Filters in PostScript Level 2*, Adobe Developer Support Technical Note 5116. Contains errata for the JPEG discussion in the *PostScript Language Reference Manual, Second Edition*. Also describes the compatibility of the JPEG implementation with various versions of the JPEG standard.

[5] Adobe Systems Incorporated, *Adobe CMap and CIDfont File Specification, version 1,* Adobe Developer Support Technical Note 5014.

[6] Adobe Systems Incorporated, *Adobe Type 1 Font Format*, Addison-Wesley, 1990, ISBN 0-201-57044-0. Explains the internal organization of a PostScript language Type 1 font program.

[7] Adobe Systems Incorporated, *Adobe Type 1 Font Format: Multiple Master Extensions*, Adobe Developer Support Technical Note 5086. Describes the additions made to the Type 1 font format to support multiple master fonts.

[8] Adobe Systems Incorporated, *The Compact Font Format Specification*, Adobe Developer Support Technical Note 5176. Describes the additions made to the Type 1 font format to support Type1C fonts.

[9] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, ISBN 0-201-00023-7. Includes a discussion of balanced trees.

[10] Arvo, James (ed.), *Graphics Gems II*, Academic Press, 1991, ISBN 0-12-064480-0. The section “Geometrically Continuous Cubic Bézier Curves” by Hans-Peter Seidel describes the mathematics used to smoothly join two cubic Bézier curves.

[11] Berners-Lee, T., and D. Connolly. Internet RFC 1866, *Hypertext Markup Language 2.0 Proposed Standard*. November 1995. For updates, see http://www.w3.org/pub/WWW/MarkUp/html-spec.

[12] Berners-Lee, T., Masinter, McCahill, and the Network Working Group. Internet RFC 1738, *Uniform Resource Locators*.

```
<URL:ftp://ds.internic.net/rfc/rfc1738.txt;type=a>
```

[13] CCITT, *Blue Book*, Volume VII.3, 1988. ISBN 92-61-03611-2. Recommendations T.4 and T.6 are the CCITT standards for Group 3 and Group 4 facsimile encoding. This document may be purchased from Global Engineering Documents, P.O. Box 19539, Irvine, California 92713.

[14] CCITT, *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*, 1988.

[15] Fielding, Network Working Group. Internet RFC 1808, *Relative Uniform Resource Locators*.

```
<URL:ftp://ds.internic.net/rfc/rfc1808.txt;type=a>
```

[16] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, *Second Edition*, Addison-Wesley, 1990, ISBN 0-201-12110-7. Section 11.2, “Parametric Cubic Curves”, contains a description of the mathematics of cubic Bézier curves and a comparison of various types of parametric cubic curves.

[17] Glassner, Andrew S. (ed.), *Graphics Gems*, Academic Press, 1990, ISBN 0-12-286165-5. 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” by Philip J. Schneider, “Some Properties of Bézier Curves” by Ronald Goldman, and “A Bézier Curve-Based Root-Finder” by Philip J. Schneider. The source code appearing in the appendix is available via anonymous ftp, as described in the preface to *Graphics Gems III* [19].

[18] Joint Photographic Experts Group (JPEG) “Revision 8 of the JPEG Technical Specification,” ISO/IEC JTC1/SC2/WG8, CCITT SGVIII, August 14, 1990. Defines a set of still-picture grayscale and color image data compression algorithms.

[19] Kirk, David (ed.), *Graphics Gems III*, Academic Press, 1992, ISBN 0-12-409670-0 (with IBM Disk) or ISBN 0-12-409671-9 (with Macintosh disk). 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 utilizes 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. All of the source code appearing in the appendix is available via anonymous ftp, as described in the preface.

[20] Microsoft Corp., *TrueType 1.0 Font Files*, Revision 1.00, May 1992.

[21] Pennebaker, W. B. and Joan L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993, ISBN 0-442-01272-1.

[22] Ron Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992.

[23] Warnock, John and D. Wyatt, “A Device Independent Graphics Imaging Model for Use with Raster Devices,” *Computer Graphics* (ACM SIGGRAPH), Volume 16, Number 3, July 1982. Technical background for the imaging model used in the PostScript language.

Colophon

This book was produced electronically using Adobe FrameMaker® on the Macintosh® and Sun™ SPARCstation® computers. Art was produced using Adobe Photoshop®, Adobe Illustrator, and Adobe FrameMaker on the Macintosh.

Authors—Tim Bienz, Richard Cohn, and Jim Meehan

Key Contributors—Alan Wootton, Nabeel Al-Shamma, Ed Taft

Language Supervisor—Ed Taft

Editors—Gary Staas, Diana Wynne

Illustrations and Book Production—Lauren Buchholz

Reviewers—Nabeel Al-Shamma, David Gelphman, Sherri Nichols, Paul Rovner, Alan Wootton, Jim Pravetz, and numerous others at Adobe Systems; and L. Peter Deutsch of Aladdin Enterprises.

Publication Management—Patrick Ames

Project Management—Rob Babcock, Bob Wulff