

First Edition
2008-7-1

Document management — Portable document format — Part 1: PDF 1.7

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing.

Copyright Notice

This document has been derived directly from the copyright ISO 32000-1 standard document available for purchase from the ISO web site at http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502. It is being made available from the web site of Adobe Systems Incorporated (http://www.adobe.com/devnet/pdf/pdf_reference.html) under agreement with ISO for those that do not need the official version containing the ISO logo and copyright notices. This version of the ISO 32000-1 standard is copyright by Adobe Systems Incorporated through an agreement with ISO who is the copyright owner of the official ISO 32000-1 document of which this is an authorized copy.

The technical material is identical between this version and the ISO Standard; the page and sections numbers are also preserved. Requests for permission to reproduce this document for any purpose should be arranged with ISO.

Contents

Page

Foreword	vi
Introduction	vii
1 Scope	1
2 Conformance	1
2.1 General	1
2.2 Conforming readers	1
2.3 Conforming writers	1
2.4 Conforming products	2
3 Normative references	2
4 Terms and definitions	6
5 Notation	10
6 Version Designations	10
7 Syntax	11
7.1 General	11
7.2 Lexical Conventions	11
7.3 Objects	13
7.4 Filters	22
7.5 File Structure	38
7.6 Encryption	55
7.7 Document Structure	70
7.8 Content Streams and Resources	81
7.9 Common Data Structures	84
7.10 Functions	92
7.11 File Specifications	99
7.12 Extensions Dictionary	108
8 Graphics	110
8.1 General	110
8.2 Graphics Objects	110
8.3 Coordinate Systems	114
8.4 Graphics State	121
8.5 Path Construction and Painting	131
8.6 Colour Spaces	138
8.7 Patterns	173
8.8 External Objects	201
8.9 Images	203
8.10 Form XObjects	217
8.11 Optional Content	222
9 Text	237
9.1 General	237
9.2 Organization and Use of Fonts	237
9.3 Text State Parameters and Operators	243
9.4 Text Objects	248
9.5 Introduction to Font Data Structures	253
9.6 Simple Fonts	254
9.7 Composite Fonts	267
9.8 Font Descriptors	281
9.9 Embedded Font Programs	288
9.10 Extraction of Text Content	292
10 Rendering	296

10.1	General	296
10.2	CIE-Based Colour to Device Colour	297
10.3	Conversions among Device Colour Spaces	297
10.4	Transfer Functions	300
10.5	Halftones	301
10.6	Scan Conversion Details	316
11	Transparency	320
11.1	General	320
11.2	Overview of Transparency	320
11.3	Basic Compositing Computations	322
11.4	Transparency Groups	332
11.5	Soft Masks	342
11.6	Specifying Transparency in PDF	344
11.7	Colour Space and Rendering Issues	353
12	Interactive Features	362
12.1	General	362
12.2	Viewer Preferences	362
12.3	Document-Level Navigation	365
12.4	Page-Level Navigation	374
12.5	Annotations	381
12.6	Actions	414
12.7	Interactive Forms	430
12.8	Digital Signatures	466
12.9	Measurement Properties	479
12.10	Document Requirements	484
13	Multimedia Features	486
13.1	General	486
13.2	Multimedia	486
13.3	Sounds	506
13.4	Movies	507
13.5	Alternate Presentations	509
13.6	3D Artwork	511
14	Document Interchange	547
14.1	General	547
14.2	Procedure Sets	547
14.3	Metadata	548
14.4	File Identifiers	551
14.5	Page-Piece Dictionaries	551
14.6	Marked Content	552
14.7	Logical Structure	556
14.8	Tagged PDF	573
14.9	Accessibility Support	610
14.10	Web Capture	616
14.11	Prepress Support	627
Annex A (informative)		
Operator Summary		643
Annex B (normative)		
Operators in Type 4 Functions		647
Annex C		

(normative)

Implementation Limits 649

Annex D
(normative)

Character Sets and Encodings 651

Annex E
(normative)

PDF Name Registry 673

Annex F
(normative)

Linearized PDF 675

Annex G
(informative)

Linearized PDF Access Strategies 695

Annex H
(informative)

Example PDF Files 699

Annex I
(normative)

PDF Versions and Compatibility 727

Annex J
(informative)

FDF Rename Flag Implementation Example 729

Annex K
(informative)

PostScript Compatibility — Transparent Imaging Model 731

Annex L
(informative)

Colour Plates 733

Bibliography 745

Foreword

On January 29, 2007, Adobe Systems Incorporated announced its intention to release the full Portable Document Format (PDF) 1.7 specification to the American National Standard Institute (ANSI) and the Enterprise Content Management Association (AIIM), for the purpose of publication by the International Organization for Standardization (ISO).

PDF has become a de facto global standard for more secure and dependable information exchange since Adobe published the complete PDF specification in 1993. Both government and private industry have come to rely on PDF for the volumes of electronic records that need to be more securely and reliably shared, managed, and in some cases preserved for generations. Since 1995 Adobe has participated in various working groups that develop technical specifications for publication by ISO and worked within the ISO process to deliver specialized subsets of PDF as standards for specific industries and functions. Today, PDF for Archive (PDF/A) and PDF for Exchange (PDF/X) are ISO standards, and PDF for Engineering (PDF/E) and PDF for Universal Access (PDF/UA) are proposed standards. Additionally, PDF for Healthcare (PDF/H) is an AIIM proposed Best Practice Guide. AIIM serves as the administrator for PDF/A, PDF/E, PDF/UA and PDF/H.

In the spring of 2008 the ISO 32000 document was prepared by Adobe Systems Incorporated (based upon PDF Reference, sixth edition, Adobe Portable Document Format version 1.7, November 2006) and was reviewed, edited and adopted, under a special “fast-track procedure”, by Technical Committee ISO/TC 171, *Document management application*, Subcommittee SC 2, *Application issues*, in parallel with its approval by the ISO member bodies.

In January 2008, this ISO technical committee approved the final revised documentation for PDF 1.7 as the international standard ISO 32000-1. In July 2008 the ISO document was placed for sale on the ISO web site (<http://www.iso.org>).

This document you are now reading is a copy of the ISO 32000-1 standard. By agreement with ISO, Adobe Systems is allowed to offer this version of the ISO standard as a free PDF file on its web site. It is not an official ISO document but the technical content is identical including the section numbering and page numbering.

Introduction

ISO 32000 specifies a digital form for representing documents called the Portable Document Format or usually referred to as PDF. PDF was developed and specified by Adobe Systems Incorporated beginning in 1993 and continuing until 2007 when this ISO standard was prepared. The Adobe Systems version PDF 1.7 is the basis for this ISO 32000 edition. The specifications for PDF are backward inclusive, meaning that PDF 1.7 includes all of the functionality previously documented in the Adobe PDF Specifications for versions 1.0 through 1.6. It should be noted that where Adobe removed certain features of PDF from their standard, they too are not contained herein.

The goal of PDF is to enable users to exchange and view electronic documents easily and reliably, independent of the environment in which they were created or the environment in which they are viewed or printed. At the core of PDF is an advanced imaging model derived from the PostScript® page description language. This PDF Imaging Model enables the description of text and graphics in a device-independent and resolution-independent manner. To improve performance for interactive viewing, PDF defines a more structured format than that used by most PostScript language programs. Unlike Postscript, which is a programming language, PDF is based on a structured binary file format that is optimized for high performance in interactive viewing. PDF also includes objects, such as annotations and hypertext links, that are not part of the page content itself but are useful for interactive viewing and document interchange.

PDF files may be created natively in PDF form, converted from other electronic formats or digitized from paper, microform, or other hard copy format. Businesses, governments, libraries, archives and other institutions and individuals around the world use PDF to represent considerable bodies of important information.

Over the past fourteen years, aided by the explosive growth of the Internet, PDF has become widely used for the electronic exchange of documents. There are several specific applications of PDF that have evolved where limiting the use of some features of PDF and requiring the use of others, enhances the usefulness of PDF. ISO 32000 is an ISO standard for the full function PDF; the following standards are for more specialized uses. PDF/X (ISO 15930) is now the industry standard for the intermediate representation of printed material in electronic prepress systems for conventional printing applications. PDF/A (ISO 19005) is now the industry standard for the archiving of digital documents. PDF/E (ISO 24517) provides a mechanism for representing engineering documents and exchange of engineering data. As major corporations, government agencies, and educational institutions streamline their operations by replacing paper-based workflow with electronic exchange of information, the impact and opportunity for the application of PDF will continue to grow at a rapid pace.

PDF, together with software for creating, viewing, printing and processing PDF files in a variety of ways, fulfils a set of requirements for electronic documents including:

- preservation of document fidelity independent of the device, platform, and software,
- merging of content from diverse sources—Web sites, word processing and spreadsheet programs, scanned documents, photos, and graphics—into one self-contained document while maintaining the integrity of all original source documents,
- collaborative editing of documents from multiple locations or platforms,
- digital signatures to certify authenticity,
- security and permissions to allow the creator to retain control of the document and associated rights,
- accessibility of content to those with disabilities,
- extraction and reuse of content for use with other file formats and applications, and
- electronic forms to gather data and integrate it with business systems.

The International Organization for Standardization draws attention to the fact that it is claimed that compliance with this document may involve the use of patents concerning the creation, modification, display and processing of PDF files which are owned by the following parties:

- Adobe Systems Incorporated, 345 Park Avenue, San Jose, California, 95110-2704, USA

ISO takes no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights has assured the ISO that they are willing to negotiate licenses under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO. Information may be obtained from those parties listed above.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO shall not be held responsible for identifying any or all such patent rights.

A repository of referenced documents has been established by AIIM (<http://www.aiim.org/pdfrefdocs>). Not all referenced documents can be found there because of copyright restrictions.

Document management — Portable document format —

Part 1: PDF 1.7

IMPORTANT — The electronic file of this document contains colours which are considered to be useful for the correct understanding of the document. Users should therefore consider printing this document using a colour printer.

1 Scope

This International Standard specifies a digital form for representing electronic documents to enable users to exchange and view electronic documents independent of the environment in which they were created or the environment in which they are viewed or printed. It is intended for the developer of software that creates PDF files (conforming writers), software that reads existing PDF files and interprets their contents for display and interaction (conforming readers) and PDF products that read and/or write PDF files for a variety of other purposes (conforming products).

This standard does not specify the following:

- specific processes for converting paper or electronic documents to the PDF format;
- specific technical design, user interface or implementation or operational details of rendering;
- specific physical methods of storing these documents such as media and storage conditions;
- methods for validating the conformance of PDF files or readers;
- required computer hardware and/or operating system.

2 Conformance

2.1 General

Conforming PDF files shall adhere to all requirements of the ISO 32000-1 specification and a conforming file is not obligated to use any feature other than those explicitly required by ISO 32000-1.

NOTE 1 The proper mechanism by which a file can presumptively identify itself as being a PDF file of a given version level is described in 7.5.2, "File Header".

2.2 Conforming readers

A conforming reader shall comply with all requirements regarding reader functional behaviour specified in ISO 32000-1. The requirements of ISO 32000-1 with respect to reader behaviour are stated in terms of general functional requirements applicable to all conforming readers. ISO 32000-1 does not prescribe any specific technical design, user interface or implementation details of conforming readers. The rendering of conforming files shall be performed as defined by ISO 32000-1.

2.3 Conforming writers

A conforming writer shall comply with all requirements regarding the creation of PDF files as specified in ISO 32000-1. The requirements of ISO 32000-1 with respect to writer behaviour are stated in terms of general functional requirements applicable to all conforming writers and focus on the creation of conforming files. ISO 32000-1 does not prescribe any specific technical design, user interface or implementation details of conforming writers.

2.4 Conforming products

A conforming product shall comply with all requirements regarding the creation of PDF files as specified in ISO 32000-1 as well as comply with all requirements regarding reader functional behavior specified in ISO 32000-1.

3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 639-1:2002, *Codes for the representation of names of languages -- Part 1: Alpha-2 code*.

ISO 639-2:1998, *Codes for the representation of names of languages -- Part 2: Alpha-3 code*.

ISO 3166-1:2006, *Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes*.

ISO 3166-2:1998, *Codes for the representation of names of countries and their subdivisions -- Part 2: Country subdivision code*.

ISO/IEC 8824-1:2002, *Abstract Syntax Notation One (ASN.1): Specification of basic notation*.

ISO/IEC 10918-1:1994, *Digital Compression and Coding of Continuous-Tone Still Images* (informally known as the JPEG standard, for the Joint Photographic Experts Group, the ISO group that developed the standard).

ISO/IEC 15444-2:2004, *Information Technology—JPEG 2000 Image Coding System: Extensions*.

ISO/IEC 11544:1993/Cor 2:2001, *Information technology—Coded representation of picture and audio information—Progressive bi-level image compression (JBIG2)*.

IEC/3WD 61966-2.1:1999, *Colour Measurement and Management in Multimedia Systems and Equipment, Part 2.1: Default RGB Colour Space—sRGB*.

ISO 15076-1:2005, *Image technology colour management - Architecture, profile format and data structure - Part 1: Based on ICC.1:2004-10*.

ISO 10646:2003, *Information technology -- Universal Multiple-Octet Coded Character Set (UCS)*.

ISO/IEC 9541-1:1991, *Information technology -- Font information interchange -- Part 1: Architecture*.

ANSI X3.4-1986, *Information Systems - Coded Sets 7-Bit American National Standard Code for Information Interchange (7-bit ASCII)*.

NOTE 1 The following documents can be found at AIIM at <http://www.aiim.org/pdfrefdocs> as well as at the Adobe Systems Incorporated Web Site http://www.adobe.com/go/pdf_ref_bibliography.

PDF Reference, Version 1.7, – 5th ed., (ISBN 0-321-30474-8), Adobe Systems Incorporated.

JavaScript for Acrobat API Reference, Version 8.0, (April 2007), Adobe Systems Incorporated.

Acrobat 3D JavaScript Reference, (April 2007), Adobe Systems Incorporated.

Adobe Glyph List, Version 2.0, (September 2002), Adobe Systems Incorporated.

OPI: Open Prepress Interface Specification 1.3, (September 1993), Adobe Systems Incorporated.

PDF Signature Build Dictionary Specification v.1.4, (March 2008), Adobe Systems Incorporated.

Adobe XML Architecture, Forms Architecture (XFA) Specification, version 2.5, (June 2007), Adobe Systems Incorporated.

Adobe XML Architecture, Forms Architecture (XFA) Specification, version 2.4, (September 2006), Adobe Systems Incorporated.

Adobe XML Architecture, Forms Architecture (XFA) Specification, version 2.2, (June 2005), Adobe Systems Incorporated.

Adobe XML Architecture, Forms Architecture (XFA) Specification, version 2.0, (October 2003), Adobe Systems Incorporated.

NOTE 2 Beginning with XFA 2.2, the XFA specification includes the Template Specification, the Config Specification, the XDP Specification, and all other XML specifications unique to the XML Forms Architecture (XFA).

Adobe XML Architecture, XML Data Package (XDP) Specification, version 2.0, (October 2003), Adobe Systems Incorporated.

Adobe XML Architecture, Template Specification, version 2.0, (October 2003), Adobe Systems Incorporated.

XML Forms Data Format Specification, version 2.0, (September 2007), Adobe Systems Incorporated.

XMP: Extensible Metadata Platform, (September 2005), Adobe Systems Incorporated.

TIFF Revision 6.0, Final, (June 1992), Adobe Systems Incorporated.

NOTE 3 The following Adobe Technical Notes can be found at the AIIM website at <http://www.aiim.org/pdfnotes> as well as at the Adobe Systems Incorporated Web Site (<http://www.adobe.com>) using the general search facility, entering the Technical Note number.

Technical Note #5004, Adobe Font Metrics File Format Specification, Version 4.1, (October 1998), Adobe Systems Incorporated.

NOTE 4 Adobe font metrics (AFM) files are available through the Type section of the ASN Web site.

Technical Note #5014, Adobe CMap and CID Font Files Specification, Version 1.0, (June 1993), Adobe Systems Incorporated.

Technical Note #5015, Type 1 Font Format Supplement, (May 1994), Adobe Systems Incorporated.

Technical Note #5078, Adobe-Japan1-4 Character Collection for CID-Keyed Fonts, (June 2004), Adobe Systems Incorporated.

Technical Note #5079, Adobe-GB1-4 Character Collection for CID-Keyed Fonts, (November 2000), Adobe Systems Incorporated.

Technical Note #5080, Adobe-CNS1-4 Character Collection for CID-Keyed Fonts, (May 2003), Adobe Systems Incorporated.

Technical Note #5087, Multiple Master Font Programs for the Macintosh, (February 1992), Adobe Systems Incorporated.

Technical Note #5088, Font Naming Issues, (April 1993), Adobe Systems Incorporated.

Technical Note #5092, CID-Keyed Font Technology Overview, (September 1994), Adobe Systems Incorporated.

Technical Note #5093, Adobe-Korea1-2 Character Collection for CID-Keyed Fonts, (May 2003), Adobe Systems Incorporated.

Technical Note #5094, Adobe CJKV Character Collections and CMaps for CID-Keyed Fonts, (June 2004), Adobe Systems Incorporated.

Technical Note #5097, Adobe-Japan2-0 Character Collection for CID-Keyed Fonts, (May 2003), Adobe Systems Incorporated.

Technical Note #5116, Supporting the DCT Filters in PostScript Level 2, (November 1992), Adobe Systems Incorporated.

Technical Note #5176, The Compact Font Format Specification, version 1.0, (December 2003), Adobe Systems Incorporated.

Technical Note #5177, The Type 2 Charstring Format, (December 2003), Adobe Systems Incorporated.

Technical Note #5411, ToUnicode Mapping File Tutorial, (May 2003), Adobe Systems Incorporated.

Technical Note #5620, Portable Job Ticket Format, Version 1.1, (April 1999), Adobe Systems Incorporated.

Technical Note #5660, Open Prepress Interface (OPI) Specification, Version 2.0, (January 2000), Adobe Systems Incorporated.

NOTE 5 The following documents are available as Federal Information Processing Standards Publications.

FIPS PUB 186-2, Digital Signature Standard, describes DSA signatures, (January 2000), Federal Information Processing Standards.

FIPS PUB 197, Advanced Encryption Standard (AES), (November 2001), Federal Information Processing Standards.

NOTE 6 The following documents are available as Internet Engineering Task Force RFCs.

RFC 1321, The MD5 Message-Digest Algorithm, (April 1992), Internet Engineering Task Force (IETF).

RFC 1738, Uniform Resource Locators, (December 1994), Internet Engineering Task Force (IETF).

RFC 1808, Relative Uniform Resource Locators, (June 1995), Internet Engineering Task Force (IETF).

RFC 1950, ZLIB Compressed Data Format Specification, Version 3.3, (May 1996), Internet Engineering Task Force (IETF).

RFC 1951, DEFLATE Compressed Data Format Specification, Version 1.3, (May 1996), Internet Engineering Task Force (IETF).

RFC 2045, Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, (November 1996), Internet Engineering Task Force (IETF).

RFC 2046, Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, (November 1996), Internet Engineering Task Force (IETF).

RFC 2083, PNG (Portable Network Graphics) Specification, Version 1.0, (March 1997), Internet Engineering Task Force (IETF).

RFC 2315, PKCS #7: Cryptographic Message Syntax, Version 1.5, (March 1998), Internet Engineering Task Force (IETF).

RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax, (August 1998), Internet Engineering Task Force (IETF).

RFC 2560, X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP, (June 1999), Internet Engineering Task Force (IETF).

RFC 2616, Hypertext Transfer Protocol—HTTP/1.1, (June 1999), Internet Engineering Task Force (IETF).

RFC 2898, PKCS #5: Password-Based Cryptography Specification Version 2.0, (September 2000), Internet Engineering Task Force (IETF).

RFC 3066, Tags for the Identification of Languages, (January 2001), Internet Engineering Task Force (IETF).

RFC 3161, Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP), (August 2001), Internet Engineering Task Force (IETF).

RFC 3174, US Secure Hash Algorithm 1 (SHA1), (September 2001), Internet Engineering Task Force (IETF).

RFC 3280, Internet X.509 Public Key Infrastructure, Certificate and Certificate Revocation List (CRL) Profile, (April 2002), Internet Engineering Task Force (IETF).

NOTE 7 The following documents are available from other sources.

Adobe Type 1 Font Format., Version 1.1, (February 1993), Addison-Wesley, ISBN 0-201-57044-0.

OpenType Font Specification 1.4, December 2004, Microsoft.

TrueType Reference Manual, (December 2002), Apple Computer, Inc.

Standard ECMA-363, Universal 3D File Format, 1st Edition (U3D), (December 2004), Ecma International.

PANOSE Classification Metrics Guide, (February 1997), Hewlett-Packard Corporation.

ICC Characterization Data Registry, International Color Consortium (ICC).

Recommendations T.4 and T.6, Group 3 and Group 4 facsimile encoding, International Telecommunication Union (ITU).

TrueType 1.0 Font Files Technical Specification, Microsoft Corporation.

Client-Side JavaScript Reference, (May 1999), Mozilla Foundation.

The Unicode Standard, Version 4.0, Addison-Wesley, Boston, MA, 2003, Unicode Consortium.

Unicode Standard Annex #9, The Bidirectional Algorithm, Version 4.0.0, (April 2003), Unicode Consortium.

Unicode Standard Annex #14, Line Breaking Properties, Version 4.0.0, (April 2003), Unicode Consortium.

Unicode Standard Annex #29, Text Boundaries, Version 4.0.0, (March 2005), Unicode Consortium.

Extensible Markup Language (XML) 1.1, World Wide Web Consortium (W3C).

4 Terms and definitions

For the purposes of this document, these terms and definitions apply.

4.1

... (ellipsis)

An ellipsis is used within PDF examples to indicate omitted detail. Pairs of ellipses are also used to bracket comments, in *italic*, about such omitted detail.

4.2

8-bit value

(see byte)

4.3

array object

a one-dimensional collection of objects arranged sequentially and implicitly numbered starting at 0

4.4

ASCII

the American Standard Code for Information Interchange, a widely used convention for encoding a specific set of 128 characters as binary numbers defined in ANSI X3.4-1986

4.5

binary data

an ordered sequence of bytes

4.6

boolean objects

either the keyword **true** or the keyword **false**

4.7

byte

a group of 8 binary digits which collectively can be configured to represent one of 256 different values and various realizations of the 8 binary digits are widely used in today's electronic equipment

4.8

catalog

the primary dictionary object containing references directly or indirectly to all other objects in the document with the exception that there may be objects in the **trailer** that are not referred to by the **catalog**

4.9

character

numeric code representing an abstract symbol according to some defined character encoding rule

NOTE 1 There are three manifestations of characters in PDF, depending on context:

- A PDF file is represented as a sequence of 8-bit bytes, some of which are interpreted as character codes in the ASCII character set and some of which are treated as arbitrary binary data depending upon the context.
- The contents (data) of a string or stream object in some contexts are interpreted as character codes in the PDFDocEncoding or UTF-16 character set.
- The contents of a string within a PDF content stream in some situations are interpreted as character codes that select glyphs to be drawn on the page according to a character encoding that is associated with the text font.

4.10

character set

a defined set of symbols each assigned a unique character value

4.11**conforming reader**

software application that is able to read and process PDF files that have been made in conformance with this specification and that itself conforms to requirements of conforming readers specified here [ISO 32000-1]

4.12**conforming product**

software application that is both a conforming reader and a conforming writer

4.13**conforming writer**

software application that is able to write PDF files that conform to this specification [ISO 32000-1]

4.14**content stream**

stream object whose data consists of a sequence of instructions describing the graphical elements to be painted on a page

4.15**cross reference table**

data structure that contains the byte offset start for each of the indirect objects within the file

4.16**developer**

Any entity, including individuals, companies, non-profits, standards bodies, open source groups, etc., who are developing standards or software to use and extend ISO 32000-1.

4.17**dictionary object**

an associative table containing pairs of objects, the first object being a name object serving as the key and the second object serving as the value and may be any kind of object including another dictionary

4.18**direct object**

any object that has not been made into an indirect object

4.19**electronic document**

electronic representation of a page-oriented aggregation of text, image and graphic data, and metadata useful to identify, understand and render that data, that can be reproduced on paper or displayed without significant loss of its information content

4.20**end-of-line marker (EOL marker)**

one or two character sequence marking the end of a line of text, consisting of a CARRIAGE RETURN character (0Dh) or a LINE FEED character (0Ah) or a CARRIAGE RETURN followed immediately by a LINE FEED

4.21**FDF file**

File conforming to the Forms Data Format containing form data or annotations that may be imported into a PDF file (see 12.7.7, "Forms Data Format")

4.22**filter**

an optional part of the specification of a stream object, indicating how the data in the stream should be decoded before it is used

4.23

font

identified collection of graphics that may be glyphs or other graphic elements [ISO 15930-4]

4.24

function

a special type of object that represents parameterized classes, including mathematical formulas and sampled representations with arbitrary resolution

4.25

glyph

recognizable abstract graphic symbol that is independent of any specific design [ISO/IEC 9541-1]

4.26

graphic state

the top of a push down stack of the graphics control parameters that define the current global framework within which the graphics operators execute

4.27

ICC profile

colour profile conforming to the ICC specification [ISO 15076-1:2005]

4.28

indirect object

an object that is labeled with a positive integer object number followed by a non-negative integer generation number followed by **obj** and having **endobj** after it

4.29

integer object

mathematical integers with an implementation specified interval centered at 0 and written as one or more decimal digits optionally preceded by a sign

4.30

name object

an atomic symbol uniquely defined by a sequence of characters introduced by a SOLIDUS (/), (2Fh) but the SOLIDUS is not considered to be part of the name

4.31

name tree

similar to a dictionary that associates keys and values but the keys in a name tree are strings and are ordered

4.32

null object

a single object of type null, denoted by the keyword **null**, and having a type and value that are unequal to those of any other object

4.33

number tree

similar to a dictionary that associates keys and values but the keys in a number tree are integers and are ordered

4.34

numeric object

either an integer object or a real object

4.35

object

a basic data structure from which PDF files are constructed and includes these types: array, Boolean, dictionary, integer, name, null, real, stream and string

4.36**object reference**

an object value used to allow one object to refer to another; that has the form “<n> <m> R” where <n> is an indirect object number, <m> is its version number and R is the uppercase letter R

4.37**object stream**

a stream that contains a sequence of PDF objects

4.38**PDF**

Portable Document Format file format defined by this specification [ISO 32000-1]

4.39**real object**

approximate mathematical real numbers, but with limited range and precision and written as one or more decimal digits with an optional sign and a leading, trailing, or embedded PERIOD (2Eh) (decimal point)

4.40**rectangle**

a specific array object used to describe locations on a page and bounding boxes for a variety of objects and written as an array of four numbers giving the coordinates of a pair of diagonally opposite corners, typically in the form [ll_x ll_y ur_x ur_y] specifying the lower-left x, lower-left y, upper-right x, and upper-right y coordinates of the rectangle, in that order

4.41**resource dictionary**

associates resource names, used in content streams, with the resource objects themselves and organized into various categories (e.g., Font, ColorSpace, Pattern)

4.42**space character**

text string character used to represent orthographic white space in text strings

NOTE 2

space characters include HORIZONTAL TAB (U+0009), LINE FEED (U+000A), VERTICAL TAB (U+000B), FORM FEED (U+000C), CARRIAGE RETURN (U+000D), SPACE (U+0020), NOBREAK SPACE (U+00A0), EN SPACE (U+2002), EM SPACE (U+2003), FIGURE SPACE (U+2007), PUNCTUATION SPACE (U+2008), THIN SPACE (U+2009), HAIR SPACE (U+200A), ZERO WIDTH SPACE (U+200B), and IDEOGRAPHIC SPACE (U+3000)

4.43**stream object**

consists of a dictionary followed by zero or more bytes bracketed between the keywords stream and endstream

4.44**string object**

consists of a series of bytes (unsigned integer values in the range 0 to 255) and the bytes are not integer objects, but are stored in a more compact form

4.45**web capture**

refers to the process of creating PDF content by importing and possibly converting internet-based or locally-resident files. The files being imported may be any arbitrary format, such as HTML, GIF, JPEG, text, and PDF

4.46**white-space character**

characters that separate PDF syntactic constructs such as names and numbers from each other; white space characters are HORIZONTAL TAB (09h), LINE FEED (0Ah), FORM FEED (0Ch), CARRIAGE RETURN (0Dh), SPACE (20h); (see Table 1 in 7.2.2, “Character Set”)

4.47

XFDF file

file conforming to the XML Forms Data Format 2.0 specification, which is an XML transliteration of Forms Data Format (FDF)

4.48

XMP packet

structured wrapper for serialized XML metadata that can be embedded in a wide variety of file formats

5 Notation

PDF operators, PDF keywords, the names of keys in PDF dictionaries, and other predefined names are written in bold sans serif font; words that denote operands of PDF operators or values of dictionary keys are written in italic sans serif font.

Token characters used to delimit objects and describe the structure of PDF files, as defined in 7.2, "Lexical Conventions", may be identified by their ANSI X3.4-1986 (ASCII 7-bit USA codes) character name written in upper case in bold sans serif font followed by a parenthetic two digit hexadecimal character value with the suffix "h".

Characters in text streams, as defined by 7.9.2, "String Object Types", may be identified by their ANSI X3.4-1986 (ASCII 7-bit USA codes) character name written in uppercase in sans serif font followed by a parenthetic four digit hexadecimal character code value with the prefix "U+" as shown in EXAMPLE 1 in this clause.

EXAMPLE 1 **EN SPACE** (U+2002).

6 Version Designations

For the convenience of the reader, the PDF versions in which various features were introduced are provided informatively within this document. The first version of PDF was designated PDF 1.0 and was specified by Adobe Systems Incorporated in the PDF Reference 1.0 document published by Adobe and Addison Wesley. Since then, PDF has gone through seven revisions designated as: PDF 1.1, PDF 1.2, PDF 1.3, PDF 1.4, PDF 1.5, PDF 1.6 and PDF 1.7. All non-deprecated features defined in a previous PDF version were also included in the subsequent PDF version. Since ISO 32000-1 is a PDF version matching PDF 1.7, it is also suitable for interpretation of files made to conform with any of the PDF specifications 1.0 through 1.7. Throughout this specification in order to indicate at which point in the sequence of versions a feature was introduced, a notation with a PDF version number in parenthesis (e.g., (*PDF 1.3*)) is used. Thus if a feature is labelled with (*PDF 1.3*) it means that PDF 1.0, PDF 1.1 and PDF 1.2 were not specified to support this feature whereas all versions of PDF 1.3 and greater were defined to support it.

7 Syntax

7.1 General

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

PDF syntax is best understood by considering it as four parts, as shown in Figure 1:

- *Objects*. A PDF document is a data structure composed from a small set of basic types of data objects. Sub-clause 7.2, "Lexical Conventions," describes the character set used to write objects and other syntactic elements. Sub-clause 7.3, "Objects," describes the syntax and essential properties of the objects. Sub-clause 7.3.8, "Stream Objects," provides complete details of the most complex data type, the stream object.
- *File structure*. The PDF file structure determines how objects are stored in a PDF file, how they are accessed, and how they are updated. This structure is independent of the semantics of the objects. Sub-clause 7.5, "File Structure," describes the file structure. Sub-clause 7.6, "Encryption," describes a file-level mechanism for protecting a document's contents from unauthorized access.
- *Document structure*. The PDF document structure specifies how the basic object types are used to represent components of a PDF document: pages, fonts, annotations, and so forth. Sub-clause 7.7, "Document Structure," describes the overall document structure; later clauses address the detailed semantics of the components.
- *Content streams*. A PDF *content stream* contains a sequence of instructions describing the appearance of a page or other graphical entity. These instructions, while also represented as objects, are conceptually distinct from the objects that represent the document structure and are described separately. Sub-clause 7.8, "Content Streams and Resources," discusses PDF content streams and their associated resources.

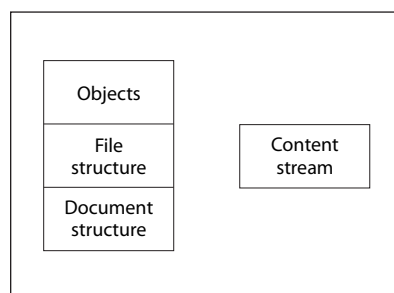


Figure 1 – PDF Components

In addition, this clause describes some data structures, built from basic objects, that are so widely used that they can almost be considered basic object types in their own right. These objects are covered in: 7.9, "Common Data Structures"; 7.10, "Functions"; and 7.11, "File Specifications."

NOTE Variants of PDF's object and file syntax are also used as the basis for other file formats. These include the Forms Data Format (FDF), described in 12.7.7, "Forms Data Format", and the Portable Job Ticket Format (PJTF), described in Adobe Technical Note #5620, *Portable Job Ticket Format*.

7.2 Lexical Conventions

7.2.1 General

At the most fundamental level, a PDF file is a sequence of bytes. These bytes can be grouped into *tokens* according to the syntax rules described in this sub-clause. One or more tokens are assembled to form higher-

level syntactic entities, principally *objects*, which are the basic data values from which a PDF document is constructed.

A non-encrypted PDF can be entirely represented using byte values corresponding to the visible printable subset of the character set defined in ANSI X3.4-1986, plus white space characters. However, a PDF file is not restricted to the ASCII character set; it may contain arbitrary bytes, subject to the following considerations:

- The tokens that delimit objects and that describe the structure of a PDF file shall use the ASCII character set. In addition all the reserved words and the names used as keys in PDF standard dictionaries and certain types of arrays shall be defined using the ASCII character set.
- The data values of strings and streams objects may be written either entirely using the ASCII character set or entirely in binary data. In actual practice, data that is naturally binary, such as sampled images, is usually represented in binary for compactness and efficiency.
- A PDF file containing binary data shall be transported as a binary file rather than as a text file to insure that all bytes of the file are faithfully preserved.

NOTE 1 A binary file is not portable to environments that impose reserved character codes, maximum line lengths, end-of-line conventions, or other restrictions

NOTE 2 In this clause, the usage of the term character is entirely independent of any logical meaning that the value may have when it is treated as data in specific contexts, such as representing human-readable text or selecting a glyph from a font.

7.2.2 Character Set

The PDF character set is divided into three classes, called *regular*, *delimiter*, and *white-space* characters. This classification determines the grouping of characters into tokens. The rules defined in this sub-clause apply to all characters in the file except within strings, streams, and comments.

The *White-space characters* shown in Table 1 separate syntactic constructs such as names and numbers from each other. All white-space characters are equivalent, except in comments, strings, and streams. In all other contexts, PDF treats any sequence of consecutive white-space characters as one character.

Table 1 – White-space characters

Decimal	Hexadecimal	Octal	Name
0	00	000	Null (NUL)
9	09	011	HORIZONTAL TAB (HT)
10	0A	012	LINE FEED (LF)
12	0C	014	FORM FEED (FF)
13	0D	015	CARRIAGE RETURN (CR)
32	20	040	SPACE (SP)

The CARRIAGE RETURN (0Dh) and LINE FEED (0Ah) characters, also called *newline characters*, shall be treated as *end-of-line* (EOL) markers. The combination of a CARRIAGE RETURN followed immediately by a LINE FEED shall be treated as one EOL marker. EOL markers may be treated the same as any other white-space characters. However, sometimes an EOL marker is required or recommended—that is, preceding a token that must appear at the beginning of a line.

NOTE The examples in this standard use a convention that arranges tokens into lines. However, the examples' use of white space for indentation is purely for clarity of exposition and need not be included in practical use.

The *delimiter characters* (,), <, >, [,], {, }, /, and % are special (LEFT PARENTHESIS (28h), RIGHT PARENTHESIS (29h), LESS-THAN SIGN (3Ch), GREATER-THAN SIGN (3Eh), LEFT SQUARE BRACKET (5Bh), RIGHT SQUARE BRACKET (5Dh), LEFT CURLY BRACE (7Bh), RIGHT CURLY BRACE (07Dh), SOLIDUS (2Fh) and PERCENT SIGN (25h), respectively). They delimit syntactic entities such as arrays, names, and comments. Any of these characters terminates the entity preceding it and is not included in the entity. Delimiter characters are allowed within the scope of a string when following the rules for composing strings; see 7.3.4.2, "Literal Strings". The leading (of a string does delimit a preceding entity and the closing) of a string delimits the string's end.

Table 2 – Delimiter characters

Glyph	Decimal	Hexadecimal	Octal	Name
(40	28	50	LEFT PARENTHESIS
)	41	29	51	RIGHT PARENTHESIS
<	60	3C	60	LESS-THAN SIGN
>	62	3E	62	GREATER-THAN SIGN
[91	5B	133	LEFT SQUARE BRACKET
]	93	5D	135	RIGHT SQUARE BRACKET
{	123	7B	173	LEFT CURLY BRACKET
}	125	7D	175	RIGHT CURLY BRACKET
/	47	2F	57	SOLIDUS
%	37	25	45	PERCENT SIGN

All characters except the white-space characters and delimiters are referred to as *regular characters*. These characters include bytes that are outside the ASCII character set. A sequence of consecutive regular characters comprises a single token. PDF is case-sensitive; corresponding uppercase and lowercase letters shall be considered distinct.

7.2.3 Comments

Any occurrence of the PERCENT SIGN (25h) outside a string or stream introduces a *comment*. The comment consists of all characters after the PERCENT SIGN and up to but not including the end of the line, including regular, delimiter, SPACE (20h), and HORIZONTAL TAB characters (09h). A conforming reader shall ignore comments, and treat them as single white-space characters. That is, a comment separates the token preceding it from the one following it.

EXAMPLE The PDF fragment in this example is syntactically equivalent to just the tokens abc and 123.

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

Comments (other than the %PDF-*n.m* and %%EOF comments described in 7.5, "File Structure") have no semantics. They are not necessarily preserved by applications that edit PDF files.

7.3 Objects

7.3.1 General

PDF includes eight basic types of objects: Boolean values, Integer and Real numbers, Strings, Names, Arrays, Dictionaries, Streams, and the null object.

Objects may be labelled so that they can be referred to by other objects. A labelled object is called an indirect object (see 7.3.10, "Indirect Objects").

Each object type, their method of creation and their proper referencing as indirect objects is described in 7.3.2, "Boolean Objects" through 7.3.10, "Indirect Objects."

7.3.2 Boolean Objects

Boolean objects represent the logical values of true and false. They appear in PDF files using the keywords **true** and **false**.

7.3.3 Numeric Objects

PDF provides two types of numeric objects: integer and real. *Integer objects* represent mathematical integers. *Real objects* represent mathematical real numbers. The range and precision of numbers may be limited by the internal representations used in the computer on which the conforming reader is running; Annex C gives these limits for typical implementations.

An integer shall be written as one or more decimal digits optionally preceded by a sign. The value shall be interpreted as a signed decimal integer and shall be converted to an integer object.

EXAMPLE 1 Integer objects

123 43445 +17 -98 0

A real value shall be written as one or more decimal digits with an optional sign and a leading, trailing, or embedded PERIOD (2Eh) (decimal point). The value shall be interpreted as a real number and shall be converted to a real object.

EXAMPLE 2 Real objects

34.5 -3.62 +123.6 4. -.002 0.0

NOTE 1 A conforming writer shall not use the PostScript syntax for numbers with non-decimal radices (such as 16#FFFE) or in exponential format (such as 6.02E23).

NOTE 2 Throughout this standard, the term *number* refers to an object whose type may be either integer or real. Wherever a real number is expected, an integer may be used instead. For example, it is not necessary to write the number 1.0 in real format; the integer 1 is sufficient.

7.3.4 String Objects

7.3.4.1 General

A *string object* shall consist of a series of zero or more bytes. String objects are not integer objects, but are stored in a more compact format. The length of a string may be subject to implementation limits; see Annex C.

String objects shall be written in one of the following two ways:

- As a sequence of literal characters enclosed in parentheses () (using LEFT PARENTHESIS (28h) and RIGHT PARENTHESIS (29h)); see 7.3.4.2, "Literal Strings."
- As hexadecimal data enclosed in angle brackets < > (using LESS-THAN SIGN (3Ch) and GREATER-THAN SIGN (3Eh)); see 7.3.4.3, "Hexadecimal Strings."

NOTE In many contexts, conventions exist for the interpretation of the contents of a string value. This sub-clause defines only the basic syntax for writing a string as a sequence of bytes; conventions or rules governing the contents of strings in particular contexts are described with the definition of those particular contexts.

7.9.2, "String Object Types," describes the encoding schemes used for the contents of string objects.

7.3.4.2 Literal Strings

A *literal string* shall be written as an arbitrary number of characters enclosed in parentheses. Any characters may appear in a string except unbalanced parentheses (LEFT PARENTHESIS (28h) and RIGHT PARENTHESIS (29h)) and the backslash (REVERSE SOLIDUS (5Ch)), which shall be treated specially as described in this sub-clause. Balanced pairs of parentheses within a string require no special treatment.

EXAMPLE 1 The following are valid literal strings:
 (This is a string)
 (Strings may contain newlines
 and such.)
 (Strings may contain balanced parentheses () and
 special characters (*!&}^% and so on).)
 (The following is an empty string.)
 ()
 (It has zero (0) length.)

Within a literal string, the REVERSE SOLIDUS is used as an escape character. The character immediately following the REVERSE SOLIDUS determines its precise interpretation as shown in Table 3. If the character following the REVERSE SOLIDUS is not one of those shown in Table 3, the REVERSE SOLIDUS shall be ignored.

Table 3 – Escape sequences in literal strings

Sequence	Meaning
\n	LINE FEED (0Ah) (LF)
\r	CARRIAGE RETURN (0Dh) (CR)
\t	HORIZONTAL TAB (09h) (HT)
\b	BACKSPACE (08h) (BS)
\f	FORM FEED (FF)
\(LEFT PARENTHESIS (28h)
\)	RIGHT PARENTHESIS (29h)
\\	REVERSE SOLIDUS (5Ch) (Backslash)
\ddd	Character code ddd (octal)

A conforming writer may split a literal string across multiple lines. The REVERSE SOLIDUS (5Ch) (backslash character) at the end of a line shall be used to indicate that the string continues on the following line. A conforming reader shall disregard the REVERSE SOLIDUS and the end-of-line marker following it when reading the string; the resulting string value shall be identical to that which would be read if the string were not split.

EXAMPLE 2 (These \
 two strings \
 are the same.)
 (These two strings are the same.)

An end-of-line marker appearing within a literal string without a preceding REVERSE SOLIDUS shall be treated as a byte value of (0Ah), irrespective of whether the end-of-line marker was a CARRIAGE RETURN (0Dh), a LINE FEED (0Ah), or both.

EXAMPLE 3 (This string has an end-of-line at the end of it.
)
 (So does this one.\n)

The `\ddd` escape sequence provides a way to represent characters outside the printable ASCII character set.

EXAMPLE 4 (This string contains \245two octal characters\307.)

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

EXAMPLE 5 the literal
 (\0053)
 denotes a string containing two characters, \005 (Control-E) followed by the digit 3, whereas both
 (\053)
 and
 (\53)
 denote strings containing the single character \053, a plus sign (+).

Since any 8-bit value may appear in a string (with proper escaping for REVERSE SOLIDUS (backslash) and unbalanced PARENTHESES) this `\ddd` notation provides a way to specify characters outside the ASCII character set by using ASCII characters only. However, any 8-bit value may appear in a string, represented either as itself or with the `\ddd` notation described.

When a document is encrypted (see 7.6, “Encryption”), all of its strings are encrypted; the encrypted string values contain arbitrary 8-bit values. When writing encrypted strings using the literal string form, the conforming writer shall follow the rules described. That is, the REVERSE SOLIDUS character shall be used as an escape to specify unbalanced PARENTHESES or the REVERSE SOLIDUS character itself. The REVERSE SOLIDUS may, but is not required, to be used to specify other, arbitrary 8-bit values.

7.3.4.3 Hexadecimal Strings

Strings may also be written in hexadecimal form, which is useful for including arbitrary binary data in a PDF file. A hexadecimal string shall be written as a sequence of hexadecimal digits (0–9 and either A–F or a–f) encoded as ASCII characters and enclosed within angle brackets (using LESS-THAN SIGN (3Ch) and GREATER-THAN SIGN (3Eh)).

EXAMPLE 1 <4E6F762073686D6F7A206B6120706F702E>

Each pair of hexadecimal digits defines one byte of the string. White-space characters (such as SPACE (20h), HORIZONTAL TAB (09h), CARRIAGE RETURN (0Dh), LINE FEED (0Ah), and FORM FEED (0Ch)) shall be ignored.

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

EXAMPLE 2 <901FA3>
 is a 3-byte string consisting of the characters whose hexadecimal codes are 90, 1F, and A3, but
 <901FA>
 is a 3-byte string containing the characters whose hexadecimal codes are 90, 1F, and A0.

7.3.5 Name Objects

Beginning with PDF 1.2 a *name object* is an atomic symbol uniquely defined by a sequence of any characters (8-bit values) except null (character code 0). *Uniquely defined* means that any two name objects made up of the same sequence of characters denote the same object. *Atomic* means that a name has no internal structure; although it is defined by a sequence of characters, those characters are not considered elements of the name.

When writing a name in a PDF file, a SOLIDUS (2Fh) (/) shall be used to introduce a name. The SOLIDUS is not part of the name but is a prefix indicating that what follows is a sequence of characters representing the name in the PDF file and shall follow these rules:

- a) A NUMBER SIGN (23h) (#) in a name shall be written by using its 2-digit hexadecimal code (23), preceded by the NUMBER SIGN.
- b) Any character in a name that is a regular character (other than NUMBER SIGN) shall be written as itself or by using its 2-digit hexadecimal code, preceded by the NUMBER SIGN.
- c) Any character that is not a regular character shall be written using its 2-digit hexadecimal code, preceded by the NUMBER SIGN only.

NOTE 1 There is not a unique encoding of names into the PDF file because regular characters may be coded in either of two ways.

White space used as part of a name shall always be coded using the 2-digit hexadecimal notation and no white space may intervene between the SOLIDUS and the encoded name.

Regular characters that are outside the range EXCLAMATION MARK(21h) (!) to TILDE (7Eh) (~) should be written using the hexadecimal notation.

The token SOLIDUS (a slash followed by no regular characters) introduces a unique valid name defined by the empty sequence of characters.

NOTE 2 The examples shown in Table 4 and containing # are not valid literal names in PDF 1.0 or 1.1.

Table 4 – Examples of literal names

Syntax for Literal name	Resulting Name
/Name1	Name1
/ASomewhatLongerName	ASomewhatLongerName
/A;Name_With-Variou***Characters?	A;Name_With-Variou***Characters?
/1.2	1.2
/\$\$	\$\$
/@pattern	@pattern
/.notdef	.notdef
/lime#20Green	Lime Green
/paired#28#29parentheses	paired()parentheses
/The_Key_of_F#23_Minor	The_Key_of_F#_Minor
/A#42	AB

In PDF, literal names shall always be introduced by the SOLIDUS character (/), unlike keywords such as **true**, **false**, and **obj**.

NOTE 3 This standard follows a typographic convention of writing names without the leading SOLIDUS when they appear in running text and tables. For example, **Type** and **FullScreen** denote names that would actually be written in a PDF file (and in code examples in this standard) as **/Type** and **/FullScreen**.

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

As stated above, name objects shall be treated as atomic within a PDF file. Ordinarily, the bytes making up the name are never treated as text to be presented to a human user or to an application external to a conforming reader. However, occasionally the need arises to treat a name object as text, such as one that represents a font name (see the **BaseFont** entry in Table 111), a colorant name in a separation or DeviceN colour space, or a structure type (see 14.7.3, "Structure Types").

In such situations, the sequence of bytes (after expansion of NUMBER SIGN sequences, if any) should be interpreted according to UTF-8, a variable-length byte-encoded representation of Unicode in which the printable ASCII characters have the same representations as in ASCII. This enables a name object to represent text virtually in any natural language, subject to the implementation limit on the length of a name.

NOTE 4 PDF does not prescribe what UTF-8 sequence to choose for representing any given piece of externally specified text as a name object. In some cases, multiple UTF-8 sequences may represent the same logical text. Name objects defined by different sequences of bytes constitute distinct name objects in PDF, even though the UTF-8 sequences may have identical external interpretations.

7.3.6 Array Objects

An *array object* is a one-dimensional collection of objects arranged sequentially. Unlike arrays in many other computer languages, PDF arrays may be heterogeneous; that is, an array's elements may be any combination of numbers, strings, dictionaries, or any other objects, including other arrays. An array may have zero elements.

An array shall be written as a sequence of objects enclosed in SQUARE BRACKETS (using LEFT SQUARE BRACKET (5Bh) and RIGHT SQUARE BRACKET (5Dh)).

EXAMPLE [549 3.14 false (Ralph) /SomeName]

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

7.3.7 Dictionary Objects

A *dictionary object* is an associative table containing pairs of objects, known as the dictionary's *entries*. The first element of each entry is the *key* and the second element is the *value*. The key shall be a name (unlike dictionary keys in PostScript, which may be objects of any type). The value may be any kind of object, including another dictionary. A dictionary entry whose value is **null** (see 7.3.9, "Null Object") shall be treated the same as if the entry does not exist. (This differs from PostScript, where **null** behaves like any other object as the value of a dictionary entry.) The number of entries in a dictionary shall be subject to an implementation limit; see Annex C. A dictionary may have zero entries.

The entries in a dictionary represent an associative table and as such shall be unordered even though an arbitrary order may be imposed upon them when written in a file. That ordering shall be ignored.

Multiple entries in the same dictionary shall not have the same key.

A dictionary shall be written as a sequence of key-value pairs enclosed in double angle brackets (<<...>>) (using LESS-THAN SIGNs (3Ch) and GREATER-THAN SIGNs (3Eh)).

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

NOTE Do not confuse the double angle brackets with single angle brackets (< and >) (using LESS-THAN SIGN (3Ch) and GREATER-THAN SIGN (3Eh)), which delimit a hexadecimal string (see 7.3.4.3, "Hexadecimal Strings").

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

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

7.3.8 Stream Objects

7.3.8.1 General

A *stream object*, like a string object, is a sequence of bytes. Furthermore, a stream may be of unlimited length, whereas a string shall be subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, shall be represented as streams.

NOTE 1 This sub-clause describes only the syntax for writing a stream as a sequence of bytes. The context in which a stream is referenced determines what the sequence of bytes represent.

A stream shall consist of a dictionary followed by zero or more bytes bracketed between the keywords **stream** (followed by newline) and **endstream**:

EXAMPLE *dictionary*
 stream
 ...Zero or more bytes...
 endstream

All streams shall be indirect objects (see 7.3.10, "Indirect Objects") and the stream dictionary shall be a direct object. The keyword **stream** that follows the stream dictionary shall be followed by an end-of-line marker consisting of either a CARRIAGE RETURN and a LINE FEED or just a LINE FEED, and not by a CARRIAGE RETURN alone. The sequence of bytes that make up a stream lie between the end-of-line marker following the **stream** keyword and the **endstream** keyword; the stream dictionary specifies the exact number of bytes. There should be an end-of-line marker after the data and before **endstream**; this marker shall not be included in the stream length. There shall not be any extra bytes, other than white space, between **endstream** and **endobj**.

Alternatively, beginning with PDF 1.2, the bytes may be contained in an external file, in which case the stream dictionary specifies the file, and any bytes between **stream** and **endstream** shall be ignored by a conforming reader.

NOTE 2 Without the restriction against following the keyword **stream** by a CARRIAGE RETURN alone, it would be impossible to differentiate a stream that uses CARRIAGE RETURN as its end-of-line marker and has a LINE FEED as its first byte of data from one that uses a CARRIAGE RETURN–LINE FEED sequence to denote end-of-line.

Table 5 lists the entries common to all stream dictionaries; certain types of streams may have additional dictionary entries, as indicated where those streams are described. The optional entries regarding *filters* for the stream indicate whether and how the data in the stream shall be transformed (decoded) before it is used. Filters are described further in 7.4, "Filters."

7.3.8.2 Stream Extent

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

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

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

Table 5 – Entries common to all stream dictionaries

Key	Type	Value
Length	integer	<i>(Required)</i> The number of bytes from the beginning of the line following the keyword stream to the last byte just before the keyword endstream . (There may be an additional EOL marker, preceding endstream , that is not included in the count and is not logically part of the stream data.) See 7.3.8.2, "Stream Extent", for further discussion.
Filter	name or array	<i>(Optional)</i> The name of a filter that shall be applied in processing the stream data found between the keywords stream and endstream , or an array of zero, one or several names. Multiple filters shall be specified in the order in which they are to be applied.
DecodeParms	dictionary or array	<i>(Optional)</i> A parameter dictionary or an array of such dictionaries, used by the filters specified by Filter . If there is only one filter and that filter has parameters, DecodeParms shall be set to the filter's parameter dictionary unless all the filter's parameters have their default values, in which case the DecodeParms entry may be omitted. If there are multiple filters and any of the filters has parameters set to nondefault values, DecodeParms shall be an array with one entry for each filter: either the parameter dictionary for that filter, or the null object if that filter has no parameters (or if all of its parameters have their default values). If none of the filters have parameters, or if all their parameters have default values, the DecodeParms entry may be omitted.
F	file specification	<i>(Optional; PDF 1.2)</i> The file containing the stream data. If this entry is present, the bytes between stream and endstream shall be ignored. However, the Length entry should still specify the number of those bytes (usually, there are no bytes and Length is 0). The filters that are applied to the file data shall be specified by FFilter and the filter parameters shall be specified by FDecodeParms .
FFilter	name or array	<i>(Optional; PDF 1.2)</i> The name of a filter to be applied in processing the data found in the stream's external file, or an array of zero, one or several such names. The same rules apply as for Filter .
FDecodeParms	dictionary or array	<i>(Optional; PDF 1.2)</i> A parameter dictionary, or an array of such dictionaries, used by the filters specified by FFilter . The same rules apply as for DecodeParms .

Table 5 – Entries common to all stream dictionaries (continued)

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

7.3.9 Null Object

The *null object* has a type and value that are unequal to those of any other object. There shall be only one object of type null, denoted by the keyword **null**. An indirect object reference (see 7.3.10, "Indirect Objects") to a nonexistent object shall be treated the same as a null object. Specifying the null object as the value of a dictionary entry (7.3.7, "Dictionary Objects") shall be equivalent to omitting the entry entirely.

7.3.10 Indirect Objects

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

- A positive integer *object number*. Indirect objects may be numbered sequentially within a PDF file, but this is not required; object numbers may be assigned in any arbitrary order.
- A non-negative integer *generation number*. In a newly created file, all indirect objects shall have generation numbers of 0. Nonzero generation numbers may be introduced when the file is later updated; see sub-clauses 7.5.4, "Cross-Reference Table" and 7.5.6, "Incremental Updates."

Together, the combination of an object number and a generation number shall uniquely identify an indirect object.

The definition of an indirect object in a PDF file shall consist of its object number and generation number (separated by white space), followed by the value of the object bracketed between the keywords **obj** and **endobj**.

EXAMPLE 1 Indirect object definition

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

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

The object may be referred to from elsewhere in the file by an *indirect reference*. Such indirect references shall consist of the object number, the generation number, and the keyword **R** (with white space separating each part):

```
12 0 R
```

Beginning with PDF 1.5, indirect objects may reside in object streams (see 7.5.7, "Object Streams"). They are referred to in the same way; however, their definition shall not include the keywords **obj** and **endobj**, and their generation number shall be zero.

An indirect reference to an undefined object shall not be considered an error by a conforming reader; it shall be treated as a reference to the null object.

EXAMPLE 2 If a file contains the indirect reference 17 0 R but does not contain the corresponding definition then the indirect reference is considered to refer to the null object.

Except where documented to the contrary any object value may be a direct or an indirect reference; the semantics are equivalent.

EXAMPLE 3 The following shows the use of an indirect object to specify the length of a stream. The value of the stream's Length entry is an integer object that follows the stream in the file. This allows applications that generate PDF in a single pass to defer specifying the stream's length until after its contents have been generated.

```

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

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

```

7.4 Filters

7.4.1 General

Stream filters are introduced in 7.3.8, "Stream Objects." An option when reading stream data is to decode it using a filter to produce the original non-encoded data. Whether to do so and which decoding filter or filters to use may be specified in the stream dictionary.

EXAMPLE 1 If a stream dictionary specifies the use of an ASCIIHexDecode filter, an application reading the data in that stream should transform the ASCII hexadecimal-encoded data in that stream in order to obtain the original binary data.

A conforming writer may encode data in a stream (for example, data for sampled images) to compress it or to convert it to a portable ASCII representation (or both). A conforming reader shall invoke the corresponding decoding filter or filters to convert the information back to its original form.

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

EXAMPLE 2 /Filter [/ASCII85Decode /LZWDecode]

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

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

- *ASCII filters* enable decoding of arbitrary binary data that has been encoded as ASCII text (see 7.2, "Lexical Conventions," for an explanation of why this type of encoding might be useful).

- *Decompression filters* enable decoding of data that has been compressed. The compressed data shall be in binary format, even if the original data is ASCII text.

NOTE 1 ASCII filters serve no useful purpose in a PDF file that is encrypted; see 7.6, “Encryption”.

NOTE 2 Compression is particularly valuable for large sampled images, since it reduces storage requirements and transmission time. Some types of compression are lossy, meaning that some data is lost during the encoding, resulting in a loss of quality when the data is decompressed. Compression in which no loss of data occurs is called lossless. Though somehow obvious it might be worth pointing out that lossy compression can only be applied to sampled image data (and only certain types of lossy compression for certain types of images). Lossless compression on the other hand can be used for any kind of stream.

The standard filters are summarized in Table 6, which also indicates whether they accept any optional parameters. The following sub-clauses describe these filters and their parameters (if any) in greater detail, including specifications of encoding algorithms for some filters.

Table 6 – Standard filters

FILTER name	Parameters	Description
ASCIIHexDecode	no	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
ASCII85Decode	no	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
LZWDecode	yes	Decompresses data encoded using the LZW (Lempel-Ziv-Welch) adaptive compression method, reproducing the original text or binary data.
FlateDecode	yes	(PDF 1.2) Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
RunLengthDecode	no	Decompresses data encoded using a byte-oriented run-length encoding algorithm, reproducing the original text or binary data (typically monochrome image data, or any data that contains frequent long runs of a single byte value).
CCITTFaxDecode	yes	Decompresses data encoded using the CCITT facsimile standard, reproducing the original data (typically monochrome image data at 1 bit per pixel).
JBIG2Decode	yes	(PDF 1.4) Decompresses data encoded using the JBIG2 standard, reproducing the original monochrome (1 bit per pixel) image data (or an approximation of that data).
DCTDecode	yes	Decompresses data encoded using a DCT (discrete cosine transform) technique based on the JPEG standard, reproducing image sample data that approximates the original data.
JPXDecode	no	(PDF 1.5) Decompresses data encoded using the wavelet-based JPEG2000 standard, reproducing the original image data.
Crypt	yes	(PDF 1.5) Decrypts data encrypted by a security handler, reproducing the data as it was before encryption.

EXAMPLE 3 The following example shows a stream, containing the marking instructions for a page, that was compressed using the LZW compression method and then encoded in ASCII base-85 representation.

```
1 0 obj
  << /Length 534
    /Filter [/ASCII85Decode /LZWDecode]
  >>
```

```

stream
J..)6T`?p&<!J9%_[umg"B7/Z7KNXbN'S+,*Q/&"OLT'F
LIDK#!n`$"<Atdi`\\Vn%b%)&'cA*VnK\CJY(sF>c!Jnl@
RM]WM;jjH6Gnc75idkL5]+cPZKEBPWdR>FF(kj1_R%W_d
&/s!;iudad7h?[L-F$+]]0A3Ck*$I0KZ?;<)CJtqi65Xb
Vc3\5ua:Q/=0$W<#N3U;H,MQKqfg1?:!UpR;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<W9k6YI\\0McJQkDeLWdPN?9A'jX*
al>iG1p&i;eVoK&juJHs9%;Xomop"5KatWRT"JQ#qYuL,
JD?M$0QP)IKn06l1apKDC@`qJ4B!!(5m+j.7F790m(Vj8
8l8Q:_CZ(Gm1%X\N1&u!FKHMB~>
endstream
endobj

```

EXAMPLE 4 The following shows the same stream without any filters applied to it. (The stream's contents are explained in 7.8.2, "Content Streams," and the operators used there are further described in clause 9, "Text".)

```

1 0 obj
  << /Length 568 >>
  stream
  2 J
  BT
  /F1 12 Tf
  0 Tc
  0 Tw
  72.5 712 TD
  [(Unfiltered streams can be read easily) 65 (,)] TJ
  0 -14 TD
  [(b) 20 (ut generally tak) 10 (e more space than \311)] TJ
  T* (compressed 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 filters are \
  suitable ) Tj
  T* (for both data and images, while others are \
  suitable only ) Tj
  T* (for continuous-tone images.) Tj
  ET
  endstream
endobj

```

7.4.2 ASCIIHexDecode Filter

The **ASCIIHexDecode** filter decodes data that has been encoded in ASCII hexadecimal form. ASCII hexadecimal encoding and ASCII base-85 encoding (7.4.3, "ASCII85Decode Filter") convert binary data, such as image data or previously compressed data, to 7-bit ASCII characters.

NOTE ASCII base-85 encoding is preferred to ASCII hexadecimal encoding. Base-85 encoding is preferred because it is more compact: it expands the data by a factor of 4:5, compared with 1:2 for ASCII hexadecimal encoding.

The **ASCIIHexDecode** filter shall produce one byte of binary data for each pair of ASCII hexadecimal digits (0–9 and A–F or a–f). All white-space characters (see 7.2, "Lexical Conventions") shall be ignored. A GREATER-THAN SIGN (3Eh) indicates EOD. Any other characters shall cause an error. If the filter encounters the EOD marker after reading an odd number of hexadecimal digits, it shall behave as if a 0 (zero) followed the last digit.

7.4.3 ASCII85Decode Filter

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

The ASCII base-85 encoding shall use the ASCII characters ! through u ((21h) - (75h)) and the character z (7Ah), with the 2-character sequence ~> (7Eh)(3Eh) as its EOD marker. The **ASCII85Decode** filter shall ignore all white-space characters (see 7.2, "Lexical Conventions"). Any other characters, and any character sequences that represent impossible combinations in the ASCII base-85 encoding shall cause an error.

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

$$(b_1 \times 256^3) + (b_2 \times 256^2) + (b_3 \times 256^1) + b_4 = \\ (c_1 \times 85^4) + (c_2 \times 85^3) + (c_3 \times 85^2) + (c_4 \times 85^1) + c_5$$

In other words, 4 bytes of binary data shall be interpreted as a base-256 number and then shall be converted to a base-85 number. The five bytes of the base-85 number shall then be converted to ASCII characters by adding 33 (the ASCII code for the character !) to each. The resulting encoded data shall contain only printable ASCII characters with codes in the range 33 (!) to 117 (u). As a special case, if all five bytes are 0, they shall be represented by the character with code 122 (z) instead of by five exclamation points (!!!!!).

If the length of the data to be encoded is not a multiple of 4 bytes, the last, partial group of 4 shall be used to produce a last, partial group of 5 output characters. Given n (1, 2, or 3) bytes of binary data, the encoder shall first append $4 - n$ zero bytes to make a complete group of 4. It shall encode this group in the usual way, but shall not apply the special z case. Finally, it shall write only the first $n + 1$ characters of the resulting group of 5. These characters shall be immediately followed by the ~> EOD marker.

The following conditions shall never occur in a correctly encoded byte sequence:

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

7.4.4 LZWDecode and FlateDecode Filters

7.4.4.1 General

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

- LZW (Lempel-Ziv-Welch) is a variable-length, adaptive compression method that has been adopted as one of the standard compression methods in the *Tag Image File Format* (TIFF) standard. For details on LZW encoding see 7.4.4.2, "Details of LZW Encoding."
- The Flate method is based on the public-domain zlib/deflate compression method, which is a variable-length Lempel-Ziv adaptive compression method cascaded with adaptive Huffman coding. It is fully defined in Internet RFCs 1950, ZLIB Compressed Data Format Specification, and 1951, DEFLATE Compressed Data Format Specification (see the Bibliography).

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

The LZW and Flate compression methods can discover and exploit many patterns in the input data, whether the data is text or images. As described later, both filters support optional transformation by a *predictor function*, which improves the compression of sampled image data.

NOTE 1 Because of its cascaded adaptive Huffman coding, Flate-encoded output is usually much more compact than LZW-encoded output for the same input. Flate and LZW decoding speeds are comparable, but Flate encoding is considerably slower than LZW encoding.

NOTE 2 Usually, both Flate and LZW encodings compress their input substantially. However, in the worst case (in which no pair of adjacent bytes appears twice), Flate encoding expands its input by no more than 11 bytes or a factor of 1.003 (whichever is larger), plus the effects of algorithm tags added by PNG predictors. For LZW encoding, the best case (all zeros) provides a compression approaching 1365:1 for long files, but the worst-case expansion is at least a factor of 1.125, which can increase to nearly 1.5 in some implementations, plus the effects of PNG tags as with Flate encoding.

7.4.4.2 Details of LZW Encoding

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

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

Whenever both the encoder and the decoder independently (but synchronously) realize that the current code length is no longer sufficient to represent the number of entries in the table, they shall increase the number of bits per code by 1. The first output code that is 10 bits long shall be the one following the creation of table entry 511, and similarly for 11 (1023) and 12 (2047) bits. Codes shall never be longer than 12 bits; therefore, entry 4095 is the last entry of the LZW table.

The encoder shall execute the following sequence of steps to generate each output code:

- a) 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.
- b) Emit the code corresponding to that sequence.
- c) Create a new table entry for the first unused code. Its value is the sequence found in step (a) followed by the next input character.

EXAMPLE 1 Suppose the input consists of the following sequence of ASCII character codes:
45 45 45 45 45 65 45 45 45 66

Starting with an empty table, the encoder proceeds as shown in Table 7.

Table 7 – Typical LZW encoding sequence

Input sequence	Output code	Code added to table	Sequence represented by new code
–	256 (clear-table)	–	–
45	45	258	45 45
45 45	258	259	45 45 45

Table 7 – Typical LZW encoding sequence (continued)

Input sequence	Output code	Code added to table	Sequence represented by new code
45 45	258	260	45 45 65
65	65	261	65 45
45 45 45	259	262	45 45 45 66
66	66	–	–
–	257 (EOD)	–	–

Codes shall be packed into a continuous bit stream, high-order bit first. This stream shall then be divided into bytes, high-order bit first. Thus, codes may straddle byte boundaries arbitrarily. After the EOD marker (code value 257), any leftover bits in the final byte shall be set to 0.

In the example above, all the output codes are 9 bits long; they would pack into bytes as follows (represented in hexadecimal):

EXAMPLE 2 80 0B 60 50 22 0C 0C 85 01

To adapt to changing input sequences, the encoder may at any point issue a clear-table code, which causes both the encoder and the decoder to restart with initial tables and a 9-bit code length. The encoder shall begin by issuing a clear-table code. It shall issue a clear-table code when the table becomes full; it may do so sooner.

7.4.4.3 LZWDecode and FlateDecode Parameters

The **LZWDecode** and **FlateDecode** filters shall accept optional parameters to control the decoding process.

NOTE Most of these parameters are related to techniques that reduce the size of compressed sampled images (rectangular arrays of colour values, described in 8.9, "Images"). For example, image data typically changes very little from sample to sample. Therefore, subtracting the values of adjacent samples (a process called differencing), and encoding the differences rather than the raw sample values, can reduce the size of the output data. Furthermore, when the image data contains several colour components (red-green-blue or cyan-magenta-yellow-black) per sample, taking the difference between the values of corresponding components in adjacent samples, rather than between different colour components in the same sample, often reduces the output data size.

Table 8 shows the parameters that may optionally be specified for **LZWDecode** and **FlateDecode** filters. Except where otherwise noted, all values supplied to the decoding filter for any optional parameters shall match those used when the data was encoded.

Table 8 – Optional parameters for LZWDecode and FlateDecode filters

Key	Type	Value
Predictor	integer	A code that selects the predictor algorithm, if any. If the value of this entry is 1, the filter shall assume that the normal algorithm was used to encode the data, without prediction. If the value is greater than 1, the filter shall assume that the data was differenced before being encoded, and Predictor selects the predictor algorithm. For more information regarding Predictor values greater than 1, see 7.4.4.4, "LZW and Flate Predictor Functions." Default value: 1.
Colors	integer	(May be used only if Predictor is greater than 1) The number of interleaved colour components per sample. Valid values are 1 to 4 (PDF 1.0) and 1 or greater (PDF 1.3). Default value: 1.

Table 8 – Optional parameters for LZWDecode and FlateDecode filters (continued)

Key	Type	Value
BitsPerComponent	integer	(May be used only if Predictor is greater than 1) The number of bits used to represent each colour component in a sample. Valid values are 1, 2, 4, 8, and (PDF 1.5) 16. Default value: 8.
Columns	integer	(May be used only if Predictor is greater than 1) The number of samples in each row. Default value: 1.
EarlyChange	integer	(LZWDecode only) An indication of when to increase the code length. If the value of this entry is 0, code length increases shall be postponed as long as possible. If the value is 1, code length increases shall occur one code early. This parameter is included because LZW sample code distributed by some vendors increases the code length one code earlier than necessary. Default value: 1.

7.4.4.4 LZW and Flate Predictor Functions

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

PDF supports two groups of predictor functions. The first, the *TIFF* group, consists of the single function that is Predictor 2 in the TIFF 6.0 specification.

NOTE 1 (In the TIFF 6.0 specification, Predictor 2 applies only to LZW compression, but here it applies to Flate compression as well.) TIFF Predictor 2 predicts that each colour component of a sample is the same as the corresponding colour component of the sample immediately to its left.

The second supported group of predictor functions, the *PNG* group, consists of the filters of the World Wide Web Consortium's Portable Network Graphics recommendation, documented in Internet RFC 2083, PNG (Portable Network Graphics) Specification (see the Bibliography).

The term predictors is used here instead of filters to avoid confusion.

There are five basic PNG predictor algorithms (and a sixth that chooses the optimum predictor function separately for each row).

Table 9 – PNG predictor algorithms

PNG Predictor Algorithms	Description
None	No prediction
Sub	Predicts the same as the sample to the left
Up	Predicts the same as the sample above
Average	Predicts the average of the sample to the left and the sample above
Paeth	A nonlinear function of the sample above, the sample to the left, and the sample to the upper left

The predictor algorithm to be used, if any, shall be indicated by the **Predictor** filter parameter (see Table 8), whose value shall be one of those listed in Table 10.

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

Table 10 – Predictor values

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

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

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

The predictor function groups also differ in significant ways:

- The postprediction data for each PNG-predicted row shall begin with an explicit algorithm tag; therefore, different rows can be predicted with different algorithms to improve compression. TIFF Predictor 2 has no such identifier; the same algorithm applies to all rows.
- The TIFF function group shall predict each colour component from the prior instance of that component, taking into account the number of bits per component and components per sample. In contrast, the PNG function group shall predict each byte of data as a function of the corresponding byte of one or more previous image samples, regardless of whether there are multiple colour components in a byte or whether a single colour component spans multiple bytes.

NOTE 2 This can yield significantly better speed at the cost of somewhat worse compression.

7.4.5 RunLengthDecode Filter

The **RunLengthDecode** filter decodes data that has been encoded in a simple byte-oriented format based on run length. The encoded data shall be a sequence of *runs*, where each run shall consist of a *length* byte followed by 1 to 128 bytes of data. If the *length* byte is in the range 0 to 127, the following *length* + 1 (1 to 128) bytes shall be copied literally during decompression. If *length* is in the range 129 to 255, the following single byte shall be copied 257 - *length* (2 to 128) times during decompression. A *length* value of 128 shall denote EOD.

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

7.4.6 CCITTFaxDecode Filter

The **CCITTFaxDecode** filter decodes image data that has been encoded using either Group 3 or Group 4 CCITT facsimile (fax) encoding.

NOTE 1 CCITT encoding is designed to achieve efficient compression of monochrome (1 bit per pixel) image data at relatively low resolutions, and so is useful only for bitmap image data, not for colour images, grayscale images, or general data.

NOTE 2 The CCITT encoding standard is defined by the International Telecommunications Union (ITU), formerly known as the Comité Consultatif International Téléphonique et Télégraphique (International Coordinating Committee for Telephony and Telegraphy). The encoding algorithm is not described in detail in this standard but can be found in ITU Recommendations T.4 and T.6 (see the Bibliography). For historical reasons, we refer to these documents as the CCITT standard.

CCITT encoding is bit-oriented, not byte-oriented. Therefore, in principle, encoded or decoded data need not end at a byte boundary. This problem shall be dealt with in the following ways:

- Unencoded data shall be treated as complete scan lines, with unused bits inserted at the end of each scan line to fill out the last byte. This approach is compatible with the PDF convention for sampled image data.
- Encoded data shall ordinarily be treated as a continuous, unbroken bit stream. The **EncodedByteAlign** parameter (described in Table 11) may be used to cause each encoded scan line to be filled to a byte boundary.

NOTE 3 Although this is not prescribed by the CCITT standard and fax machines never do this, some software packages find it convenient to encode data this way.

- When a filter reaches EOD, it shall always skip to the next byte boundary following the encoded data.

The filter shall not perform any error correction or resynchronization, except as noted for the **DamagedRowsBeforeError** parameter in Table 11.

Table 11 lists the optional parameters that may be used to control the decoding. Except where noted otherwise, all values supplied to the decoding filter by any of these parameters shall match those used when the data was encoded.

Table 11 – Optional parameters for the CCITTFaxDecode filter

Key	Type	Value
K	integer	<p>A code identifying the encoding scheme used:</p> <p><0 Pure two-dimensional encoding (Group 4)</p> <p>=0 Pure one-dimensional encoding (Group 3, 1-D)</p> <p>>0 Mixed one- and two-dimensional encoding (Group 3, 2-D), in which a line encoded one-dimensionally may be followed by at most K – 1 lines encoded two-dimensionally</p> <p>The filter shall distinguish among negative, zero, and positive values of K to determine how to interpret the encoded data; however, it shall not distinguish between different positive K values. Default value: 0.</p>

Table 11 – Optional parameters for the CCITTFaxDecode filter (continued)

Key	Type	Value
EndOfLine	boolean	A flag indicating whether end-of-line bit patterns shall be present in the encoding. The CCITTFaxDecode filter shall always accept end-of-line bit patterns. If EndOfLine is true end-of-line bit patterns shall be present. Default value: false .
EncodedByteAlign	boolean	A flag indicating whether the filter shall expect extra 0 bits before each encoded line so that the line begins on a byte boundary. If true , the filter shall skip over encoded bits to begin decoding each line at a byte boundary. If false , the filter shall not expect extra bits in the encoded representation. Default value: false .
Columns	integer	The width of the image in pixels. If the value is not a multiple of 8, the filter shall adjust the width of the unencoded image to the next multiple of 8 so that each line starts on a byte boundary. Default value: 1728.
Rows	integer	The height of the image in scan lines. If the value is 0 or absent, the image's height is not predetermined, and the encoded data shall be terminated by an end-of-block bit pattern or by the end of the filter's data. Default value: 0.
EndOfBlock	boolean	A flag indicating whether the filter shall expect the encoded data to be terminated by an end-of-block pattern, overriding the Rows parameter. If false , the filter shall stop when it has decoded the number of lines indicated by Rows or when its data has been exhausted, whichever occurs first. The end-of-block pattern shall be the CCITT end-of-facsimile-block (EOFB) or return-to-control (RTC) appropriate for the K parameter. Default value: true .
BlackIs1	boolean	A flag indicating whether 1 bits shall be interpreted as black pixels and 0 bits as white pixels, the reverse of the normal PDF convention for image data. Default value: false .
DamagedRowsBeforeError	integer	The number of damaged rows of data that shall be tolerated before an error occurs. This entry shall apply only if EndOfLine is true and K is non-negative. Tolerating a damaged row shall mean locating its end in the encoded data by searching for an EndOfLine pattern and then substituting decoded data from the previous row if the previous row was not damaged, or a white scan line if the previous row was also damaged. Default value: 0.

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

7.4.7 JBIG2Decode Filter

The **JBIG2Decode** filter (*PDF 1.4*) decodes monochrome (1 bit per pixel) image data that has been encoded using JBIG2 encoding.

NOTE 1 JBIG stands for the Joint Bi-Level Image Experts Group, a group within the International Organization for Standardization (ISO) that developed the format. JBIG2 is the second version of a standard originally released as JBIG1.

JBIG2 encoding, which provides for both lossy and lossless compression, is useful only for monochrome images, not for colour images, grayscale images, or general data. The algorithms used by the encoder, and

the details of the format, are not described here. See ISO/IEC 11544 published standard for the current JBIG2 specification. Additional information can be found through the Web site for the JBIG and JPEG (Joint Photographic Experts Group) committees at <<http://www.jpeg.org>>.

In general, JBIG2 provides considerably better compression than the existing CCITT standard (discussed in 7.4.6, "CCITTFaxDecode Filter"). The compression it achieves depends strongly on the nature of the image. Images of pages containing text in any language compress particularly well, with typical compression ratios of 20:1 to 50:1 for a page full of text.

The JBIG2 encoder shall build a table of unique symbol bitmaps found in the image, and other symbols found later in the image shall be matched against the table. Matching symbols shall be replaced by an index into the table, and symbols that fail to match shall be added to the table. The table itself shall be compressed using other means.

NOTE 2 This method results in high compression ratios for documents in which the same symbol is repeated often, as is typical for images created by scanning text pages. It also results in high compression of white space in the image, which does not need to be encoded because it contains no symbols.

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

The JBIG2 compression method may also be used for encoding multiple images into a single JBIG2 bit stream.

NOTE 3 Typically, these images are scanned pages of a multiple-page document. Since a single table of symbol bitmaps is used to match symbols across multiple pages, this type of encoding can result in higher compression ratios than if each of the pages had been individually encoded using JBIG2.

In general, an image may be specified in PDF as either an *image XObject* or an *inline image* (as described in 8.9, "Images"); however, the **JBIG2Decode** filter shall not be used with inline images.

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

- The filter shall use the embedded file organization of JBIG2. (The details of this and the other types of file organization are provided in an annex of the ISO specification.) The optional 2-byte combination (marker) mentioned in the specification shall not be used in PDF. JBIG2 bit streams in random-access organization should be converted to the embedded file organization. Bit streams in sequential organization need no reorganization, except for the mappings described below.
- The JBIG2 file header, end-of-page segments, and end-of-file segment shall not be used in PDF. These should be removed before the PDF objects described below are created.
- The image XObject to which the **JBIG2Decode** filter is applied shall contain all segments that are associated with the JBIG2 page represented by that image; that is, all segments whose segment page association field contains the page number of the JBIG2 page represented by the image. In the image XObject, however, the segment's page number should always be 1; that is, when each such segment is written to the XObject, the value of its segment page association field should be set to 1.
- If the bit stream contains global segments (segments whose segment page association field contains 0), these segments shall be placed in a separate PDF stream, and the filter parameter listed in Table 12 should refer to that stream. The stream can be shared by multiple image XObjects whose JBIG2 encodings use the same global segments.

Table 12 – Optional parameter for the JBIG2Decode filter

Key	Type	Value
JBIG2Globals	stream	A stream containing the JBIG2 global (page 0) segments. Global segments shall be placed in this stream even if only a single JBIG2 image XObject refers to it.

EXAMPLE 1 The following shows an image that was compressed using the JBIG2 compression method and then encoded in ASCII hexadecimal representation. Since the JBIG2 bit stream contains global segments, these segments are placed in a separate PDF stream, as indicated by the JBIG2Globals filter parameter.

```
5 0 obj
  << /Type /XObject
    /Subtype /Image
    /Width 52
    /Height 66
    /ColorSpace /DeviceGray
    /BitsPerComponent 1
    /Length 224
    /Filter [/ASCIIHexDecode /JBIG2Decode]
    /DecodeParms [null << /JBIG2Globals 6 0 R >>]
  >>
  stream
  000000013000010000001300000034000000420000000000
  000000400000000000002062000010000001e000000340000
  00420000000000000000200100000000231db51ce51ffac>
  endstream
endobj

6 0 obj
  << /Length 126
    /Filter /ASCIIHexDecode
  >>
  stream
  000000000000100000000032000003ffdf02fefefe000000
  01000000012ae225aea9a5a538b4d9999c5c8e56ef0f872
  7f2b53d4e37ef795cc5506dffac>
  endstream
endobj
```

The JBIG2 bit stream for this example is as follows:

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

This bit stream is made up of the following parts (in the order listed):

a) The JBIG2 file header

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

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

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

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

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

c) The page information segment

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

and the immediate text region segment

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

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

d) The end-of-page segment

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

and the end-of-file segment

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

Since these segments shall not be used in PDF, they are discarded.

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

7.4.8 DCTDecode Filter

The **DCTDecode** filter decodes grayscale or colour image data that has been encoded in the JPEG baseline format. See Adobe Technical Note #5116 for additional information about the use of JPEG “markers.”

NOTE 1 JPEG stands for the Joint Photographic Experts Group, a group within the International Organization for Standardization that developed the format; DCT stands for discrete cosine transform, the primary technique used in the encoding.

JPEG encoding is a lossy compression method, designed specifically for compression of sampled continuous-tone images and not for general data compression.

Data to be encoded using JPEG shall consist of a stream of image samples, each consisting of one, two, three, or four colour components. The colour component values for a particular sample shall appear consecutively. Each component value shall occupy a byte.

During encoding, several parameters shall control the algorithm and the information loss. The values of these parameters, which include the dimensions of the image and the number of components per sample, are entirely under the control of the encoder and shall be stored in the encoded data. **DCTDecode** may obtain the parameter values it requires directly from the encoded data. However, in one instance, the parameter need not be present in the encoded data but shall be specified in the filter parameter dictionary; see Table 13.

NOTE 2 The details of the encoding algorithm are not presented here but are in the ISO standard and in JPEG: Still Image Data Compression Standard, by Pennebaker and Mitchell (see the Bibliography). Briefly, the JPEG algorithm breaks an image up into blocks that are 8 samples wide by 8 samples high. Each colour component in an image is treated separately. A two-dimensional DCT is performed on each block. This operation produces 64 coefficients, which are then quantized. Each coefficient may be quantized with a different step size. It is this quantization that results in the loss of information in the JPEG algorithm. The quantized coefficients are then compressed.

Table 13 – Optional parameter for the DCTDecode filter

Key	Type	Value
ColorTransform	integer	<p>(Optional) A code specifying the transformation that shall be performed on the sample values:</p> <p>0 No transformation.</p> <p>1 If the image has three colour components, <i>RGB</i> values shall be transformed to <i>YUV</i> before encoding and from <i>YUV</i> to <i>RGB</i> after decoding. If the image has four components, <i>CMYK</i> values shall be transformed to <i>YUVK</i> before encoding and from <i>YUVK</i> to <i>CMYK</i> after decoding. This option shall be ignored if the image has one or two colour components.</p> <p>If the encoding algorithm has inserted the Adobe-defined marker^a code in the encoded data indicating the ColorTransform value, then the colours shall be transformed, or not, after the DCT decoding has been performed according to the value provided in the encoded data and the value of this dictionary entry shall be ignored. If the Adobe-defined marker code in the encoded data indicating the ColorTransform value is not present then the value specified in this dictionary entry will be used. If the Adobe-defined marker code in the encoded data indicating the ColorTransform value is not present and this dictionary entry is not present in the filter dictionary then the default value of ColorTransform shall be 1 if the image has three components and 0 otherwise.</p>
<p>^a Parameters that control the decoding process as well as other metadata is embedded within the encoded data stream using a notation referred to as “markers”. When it defined the use of JPEG images within PostScript data streams, Adobe System Incorporated defined a particular set of rules pertaining to which markers are to be recognized, which are to be ignored and which are considered errors. A specific Adobe-defined marker was also introduced. The exact rules for producing and consuming DCT encoded data within PostScript are provide in Adobe Technical Note #5116 (reference). PDF DCT Encoding shall exactly follow those rules established by Adobe for PostScript.</p>		

NOTE 3 The encoding algorithm can reduce the information loss by making the step size in the quantization smaller at the expense of reducing the amount of compression achieved by the algorithm. The compression achieved by the JPEG algorithm depends on the image being compressed and the amount of loss that is acceptable. In general, a compression of 15:1 can be achieved without perceptible loss of information, and 30:1 compression causes little impairment of the image.

NOTE 4 Better compression is often possible for colour spaces that treat luminance and chrominance separately than for those that do not. The RGB-to-YUV conversion provided by the filters is one attempt to separate luminance and chrominance; it conforms to CCIR recommendation 601-1. Other colour spaces, such as the CIE 1976 L*a*b* space, may also achieve this objective. The chrominance components can then be compressed more than the luminance by using coarser sampling or quantization, with no degradation in quality.

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

NOTE 5 There is no benefit to using progressive JPEG for stream data that is embedded in a PDF file. Decoding progressive JPEG is slower and consumes more memory than baseline JPEG. The purpose of this feature is to enable a stream to refer to an external file whose data happens to be already encoded in progressive JPEG.

7.4.9 JPXDecode Filter

The **JPXDecode** filter (*PDF 1.5*) decodes data that has been encoded using the JPEG2000 compression method, an ISO standard for the compression and packaging of image data.

NOTE 1 JPEG2000 defines a wavelet-based method for image compression that gives somewhat better size reduction than other methods such as regular JPEG or CCITT. Although the filter can reproduce samples that are losslessly compressed.

This filter shall only be applied to image XObjects, and not to inline images (see 8.9, "Images"). It is suitable both for images that have a single colour component and for those that have multiple colour components. The colour components in an image may have different numbers of bits per sample. Any value from 1 to 38 shall be allowed.

NOTE 2 From a single JPEG2000 data stream, multiple versions of an image may be decoded. These different versions form progressions along four degrees of freedom: sampling resolution, colour depth, band, and location. For example, with a resolution progression, a thumbnail version of the image may be decoded from the data, followed by a sequence of other versions of the image, each with approximately four times as many samples (twice the width times twice the height) as the previous one. The last version is the full-resolution image.

NOTE 3 Viewing and printing applications may gain performance benefits by using the resolution progression. If the full-resolution image is densely sampled, the application may be able to select and decode only the data making up a lower-resolution version, thereby spending less time decoding. Fewer bytes need be processed, a particular benefit when viewing files over the Web. The tiling structure of the image may also provide benefits if only certain areas of an image need to be displayed or printed.

NOTE 4 Information on these progressions is encoded in the data; no decode parameters are needed to describe them. The decoder deals with any progressions it encounters to deliver the correct image data. Progressions that are of no interest may simply have performance consequences.

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

NOTE 5 To promote interoperability, the specifications define a subset of JPX called JPX baseline (of which JP2 is also a subset). The complete details of the baseline set of JPX features are contained in ISO/IEC 15444-2, Information Technology—JPEG 2000 Image Coding System: Extensions (see the Bibliography). See also <<http://www.jpeg.org/jpeg2000/>>.

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

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

- An *ordinary* channel contains values that, when decoded, shall become samples for a specified colour component.
- An *opacity* channel provides samples that shall be interpreted as raw opacity information.
- A *premultiplied opacity* channel shall provide samples that have been multiplied into the colour samples of those channels with which it is associated.

Opacity and premultiplied opacity channels shall be associated with specific colour channels. There shall not be more than one opacity channel (of either type) associated with a given colour channel.

EXAMPLE It is possible for one opacity channel to apply to the red samples and another to apply to the green and blue colour channels of an RGB image.

NOTE 6 The method by which the opacity information is to be used is explicitly not specified, although one possible method shows a normal blending mode.

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

Images in JPX files may have one of the following colour spaces:

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

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

There are other features in the JPX format beyond describing a simple image. These include provisions for describing layering and giving instructions on composition, specifying simple animation, and including generic XML metadata (along with JPEG2000-specific schemas for such data). Relevant metadata should be replicated in the image dictionary's **Metadata** stream in XMP format (see 14.3.2, "Metadata Streams").

When using the **JPXDecode** filter with image XObjects, the following changes to and constraints on some entries in the image dictionary shall apply (see 8.9.5, "Image Dictionaries" for details on these entries):

- **Width** and **Height** shall match the corresponding width and height values in the JPEG2000 data.
- **ColorSpace** shall be optional since JPEG2000 data contain colour space specifications. If present, it shall determine how the image samples are interpreted, and the colour space specifications in the JPEG2000 data shall be ignored. The number of colour channels in the JPEG2000 data shall match the number of components in the colour space; a conforming writer shall ensure that the samples are consistent with the colour space used.
- Any colour space other than **Pattern** may be specified. If an **Indexed** colour space is used, it shall be subject to the PDF limit of 256 colours. If the colour space does not match one of JPX's enumerated colour spaces (for example, if it has two colour components or more than four), it should be specified as a vendor colour space in the JPX data.
- If **ColorSpace** is not present in the image dictionary, the colour space information in the JPEG2000 data shall be used. A JPEG2000 image within a PDF shall have one of: the baseline JPX colorspace; or enumerated colorspace 19 (CIEJab) or enumerated colorspace 12 (CMYK); or at least one ICC profile that is valid within PDF. Conforming PDF readers shall support the JPX baseline set of enumerated colour spaces; they shall also be responsible for dealing with the interaction between the colour spaces and the bit depth of samples.
- If multiple colour space specifications are given in the JPEG2000 data, a conforming reader should attempt to use the one with the highest precedence and best approximation value. If the colour space is given by an unsupported ICC profile, the next lower colour space, in terms of precedence and approximation value, shall be used. If no supported colour space is found, the colour space used shall be DeviceGray, DeviceRGB, or DeviceCMYK, depending on the whether the number of channels in the JPEG2000 data is 1,3, or 4.
- **SMaskInData** specifies whether soft-mask information packaged with the image samples shall be used (see 11.6.5.3, "Soft-Mask Images"); if it is, the **SMask** entry shall not be present. If **SMaskInData** is nonzero, there shall be only one opacity channel in the JPEG2000 data and it shall apply to all colour channels.
- **Decode** shall be ignored, except in the case where the image is treated as a mask; that is, when **ImageMask** is **true**. In this case, the JPEG2000 data shall provide a single colour channel with 1-bit samples.

7.4.10 Crypt Filter

The **Crypt** filter (PDF 1.5) allows the document-level security handler (see 7.6, "Encryption") to determine which algorithms should be used to decrypt the input data. The **Name** parameter in the decode parameters dictionary for this filter (see Table 14) shall specify which of the named crypt filters in the document (see 7.6.5, "Crypt Filters") shall be used. The Crypt filter shall be the first filter in the Filter array entry.

Table 14 – Optional parameters for Crypt filters

Key	Type	Value
Type	name	(Optional) If present, shall be CryptFilterDecodeParms for a Crypt filter decode parameter dictionary.
Name	name	(Optional) The name of the crypt filter that shall be used to decrypt this stream. The name shall correspond to an entry in the CF entry of the encryption dictionary (see Table 20) or one of the standard crypt filters (see Table 26). Default value: Identity .

In addition, the decode parameters dictionary may include entries that are private to the security handler. Security handlers may use information from both the crypt filter decode parameters dictionary and the crypt filter dictionaries (see Table 25) when decrypting data or providing a key to decrypt data.

NOTE When adding private data to the decode parameters dictionary, security handlers should name these entries in conformance with the PDF name registry (see Annex E).

If a stream specifies a crypt filter, then the security handler does not apply "Algorithm 1: Encryption of data using the RC4 or AES algorithms" in 7.6.2, "General Encryption Algorithm," to the key prior to decrypting the stream. Instead, the security handler shall decrypt the stream using the key as is. Sub-clause 7.4, "Filters," explains how a stream specifies filters.

7.5 File Structure

7.5.1 General

This sub-clause describes how objects are organized in a PDF file for efficient random access and incremental update. A basic conforming PDF file shall be constructed of following four elements (see Figure 2):

- A one-line *header* identifying the version of the PDF specification to which the file conforms
- A *body* containing the objects that make up the document contained in the file
- A *cross-reference table* containing information about the indirect objects in the file
- A *trailer* giving the location of the cross-reference table and of certain special objects within the body of the file

This initial structure may be modified by later updates, which append additional elements to the end of the file; see 7.5.6, "Incremental Updates," for details.

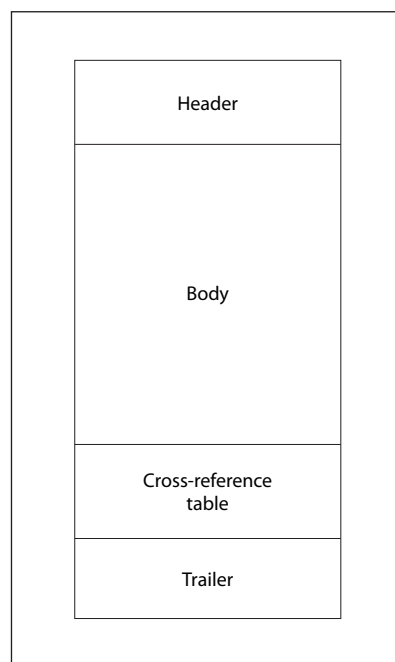


Figure 2 – Initial structure of a PDF file

As a matter of convention, the tokens in a PDF file are arranged into lines; see 7.2, "Lexical Conventions." Each line shall be terminated by an end-of-line (EOL) marker, which may be a CARRIAGE RETURN (0Dh), a LINE FEED (0Ah), or both. PDF files with binary data may have arbitrarily long lines.

NOTE To increase compatibility with compliant programs that process PDF files, lines that are not part of stream object data are limited to no more than 255 characters, with one exception. Beginning with PDF 1.3, the Contents string of a signature dictionary (see 12.8, "Digital Signatures") is not subject to the restriction on line length.

The rules described here are sufficient to produce a basic conforming PDF file. However, additional rules apply to organizing a PDF file to enable efficient incremental access to a document's components in a network environment. This form of organization, called *Linearized PDF*, is described in Annex F.

7.5.2 File Header

The first line of a PDF file shall be a *header* consisting of the 5 characters `%PDF-` followed by a version number of the form 1.N, where N is a digit between 0 and 7.

A conforming reader shall accept files with any of the following headers:

```

%PDF-1.0
%PDF-1.1
%PDF-1.2
%PDF-1.3
%PDF-1.4
%PDF-1.5
%PDF-1.6
%PDF-1.7
  
```

Beginning with PDF 1.4, the **Version** entry in the document's catalog dictionary (located via the **Root** entry in the file's trailer, as described in 7.5.5, "File Trailer"), if present, shall be used instead of the version specified in the Header.

NOTE This allows a conforming writer to update the version using an incremental update (see 7.5.6, "Incremental Updates").

Under some conditions, a conforming reader may be able to process PDF files conforming to a later version than it was designed to accept. New PDF features are often introduced in such a way that they can safely be ignored by a conforming reader that does not understand them (see 1.2, "PDF Version Numbers").

This part of ISO 32000 defines the Extensions entry in the document's catalog dictionary. If present, it shall identify any developer-defined extensions that are contained in this PDF file. See 7.12, "Extensions Dictionary".

If a PDF file contains binary data, as most do (see 7.2, "Lexical Conventions"), the header line shall be immediately followed by a comment line containing at least four binary characters—that is, characters whose codes are 128 or greater. This ensures proper behaviour of file transfer applications that inspect data near the beginning of a file to determine whether to treat the file's contents as text or as binary.

7.5.3 File Body

The *body* of a PDF file shall consist of a sequence of indirect objects representing the contents of a document. The objects, which are of the basic types described in 7.3, "Objects," represent components of the document such as fonts, pages, and sampled images. Beginning with PDF 1.5, the body can also contain object streams, each of which contains a sequence of indirect objects; see 7.5.7, "Object Streams."

7.5.4 Cross-Reference Table

The *cross-reference table* contains information that permits random access to indirect objects within the file so that the entire file need not be read to locate any particular object. The table shall contain a one-line entry for each indirect object, specifying the byte offset of that object within the body of the file. (Beginning with PDF 1.5, some or all of the cross-reference information may alternatively be contained in cross-reference streams; see 7.5.8, "Cross-Reference Streams.")

NOTE 1 The cross-reference table is the only part of a PDF file with a fixed format, which permits entries in the table to be accessed randomly.

The table comprises one or more *cross-reference sections*. Initially, the entire table consists of a single section (or two sections if the file is linearized; see Annex F). One additional section shall be added each time the file is incrementally updated (see 7.5.6, "Incremental Updates").

Each cross-reference section shall begin with a line containing the keyword **xref**. Following this line shall be one or more *cross-reference subsections*, which may appear in any order. For a file that has never been incrementally updated, the cross-reference section shall contain only one subsection, whose object numbering begins at 0.

NOTE 2 The subsection structure is useful for incremental updates, since it allows a new cross-reference section to be added to the PDF file, containing entries only for objects that have been added or deleted.

Each cross-reference subsection shall contain entries for a contiguous range of object numbers. The subsection shall begin with a line containing two numbers separated by a SPACE (20h), denoting the object number of the first object in this subsection and the number of entries in the subsection.

EXAMPLE 1 The following line introduces a subsection containing five objects numbered consecutively from 28 to 32.

28 5

A given object number shall not have an entry in more than one subsection within a single section.

Following this line are the cross-reference entries themselves, one per line. Each entry shall be exactly 20 bytes long, including the end-of-line marker. There are two kinds of cross-reference entries: one for objects that are in use and another for objects that have been deleted and therefore are free. Both types of entries have

similar basic formats, distinguished by the keyword **n** (for an in-use entry) or **f** (for a free entry). The format of an in-use entry shall be:

nnnnnnnnnn ggggg n eol

where:

nnnnnnnnnn shall be a 10-digit byte offset in the decoded stream

ggggg shall be a 5-digit generation number

n shall be a keyword identifying this as an in-use entry

eol shall be a 2-character end-of-line sequence

The byte offset in the decoded stream shall be a 10-digit number, padded with leading zeros if necessary, giving the number of bytes from the beginning of the file to the beginning of the object. It shall be separated from the generation number by a single SPACE. The generation number shall be a 5-digit number, also padded with leading zeros if necessary. Following the generation number shall be a single SPACE, the keyword **n**, and a 2-character end-of-line sequence consisting of one of the following: SP CR, SP LF, or CR LF. Thus, the overall length of the entry shall always be exactly 20 bytes.

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

nnnnnnnnnn ggggg f eol

where:

nnnnnnnnnn shall be the 10-digit object number of the next free object

ggggg shall be a 5-digit generation number

f shall be a keyword identifying this as a free entry

eol shall be a 2-character end-of-line sequence

There are two ways an entry may be a member of the free entries list. Using the basic mechanism the free entries in the cross-reference table may form a linked list, with each free entry containing the object number of the next. The first entry in the table (object number 0) shall always be free and shall have a generation number of 65,535; it shall be the head of the linked list of free objects. The last free entry (the tail of the linked list) links back to object number 0. Using the second mechanism, the table may contain other free entries that link back to object number 0 and have a generation number of 65,535, even though these entries are not in the linked list itself.

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

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

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

```

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

```

EXAMPLE 3 The following 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.

```

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

```

See H.7, "Updating Example", for a more extensive example of the structure of a PDF file that has been updated several times.

7.5.5 File Trailer

The *trailer* of a PDF file enables a conforming reader to quickly find the cross-reference table and certain special objects. Conforming readers should read a PDF file from its end. The last line of the file shall contain only the end-of-file marker, **%%EOF**. The two preceding lines shall contain, one per line and in order, the keyword **startxref** and the byte offset in the decoded stream from the beginning of the file to the beginning of the **xref** keyword in the last cross-reference section. The **startxref** line shall be preceded by the *trailer dictionary*, consisting of the keyword **trailer** followed by a series of key-value pairs enclosed in double angle brackets (<<...>>) (using LESS-THAN SIGNS (3Ch) and GREATER-THAN SIGNS (3Eh)). Thus, the trailer has the following overall structure:

```

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

```

Table 15 lists the contents of the trailer dictionary.

Table 15 – Entries in the file trailer dictionary

Key	Type	Value
Size	integer	<i>(Required; shall not be an indirect reference)</i> The total number of entries in the file's cross-reference table, as defined by the combination of the original section and all update sections. Equivalently, this value shall be 1 greater than the highest object number defined in the file. Any object in a cross-reference section whose number is greater than this value shall be ignored and defined to be missing by a conforming reader.
Prev	integer	<i>(Present only if the file has more than one cross-reference section; shall be an indirect reference)</i> The byte offset in the decoded stream from the beginning of the file to the beginning of the previous cross-reference section.
Root	dictionary	<i>(Required; shall be an indirect reference)</i> The catalog dictionary for the PDF document contained in the file (see 7.7.2, "Document Catalog").
Encrypt	dictionary	<i>(Required if document is encrypted; PDF 1.1)</i> The document's encryption dictionary (see 7.6, "Encryption").
Info	dictionary	<i>(Optional; shall be an indirect reference)</i> The document's information dictionary (see 14.3.3, "Document Information Dictionary").
ID	array	<i>(Required if an Encrypt entry is present; optional otherwise; PDF 1.1)</i> An array of two byte-strings constituting a file identifier (see 14.4, "File Identifiers") for the file. If there is an Encrypt entry this array and the two byte-strings shall be direct objects and shall be unencrypted. NOTE 1 Because the ID entries are not encrypted it is possible to check the ID key to assure that the correct file is being accessed without decrypting the file. The restrictions that the string be a direct object and not be encrypted assure that this is possible. NOTE 2 Although this entry is optional, its absence might prevent the file from functioning in some workflows that depend on files being uniquely identified. NOTE 3 The values of the ID strings are used as input to the encryption algorithm. If these strings were indirect, or if the ID array were indirect, these strings would be encrypted when written. This would result in a circular condition for a reader: the ID strings must be decrypted in order to use them to decrypt strings, including the ID strings themselves. The preceding restriction prevents this circular condition.

NOTE Table 19 defines an additional entry, **XRefStm**, that appears only in the trailer of hybrid-reference files, described in 7.5.8.4, "Compatibility with Applications That Do Not Support Compressed Reference Streams."

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

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

7.5.6 Incremental Updates

The contents of a PDF file can be updated incrementally without rewriting the entire file. When updating a PDF file incrementally, changes shall be appended to the end of the file, leaving its original contents intact.

NOTE 1 The main advantage to updating a file in this way is that small changes to a large document can be saved quickly. There are additional advantages:

In certain contexts, such as when editing a document across an HTTP connection or using OLE embedding (a Windows-specific technology), a conforming writer cannot overwrite the contents of the original file. Incremental updates may be used to save changes to documents in these contexts.

NOTE 2 The resulting file has the structure shown in Figure 3. A complete example of an updated file is shown in H.7, "Updating Example".

A cross-reference section for an incremental update shall contain entries only for objects that have been changed, replaced, or deleted. Deleted objects shall be left unchanged in the file, but shall be marked as deleted by means of their cross-reference entries. The added trailer shall contain all the entries except the **Prev** entry (if present) from the previous trailer, whether modified or not. In addition, the added trailer dictionary shall contain a **Prev** entry giving the location of the previous cross-reference section (see Table 15). Each trailer shall be terminated by its own end-of-file (%%EOF) marker.

NOTE 3 As shown in Figure 3, a file that has been updated several times contains several trailers. Because updates are appended to PDF files, a file may have several copies of an object with the same object identifier (object number and generation number).

EXAMPLE Several copies of an object can occur if a text annotation (see 12.5, "Annotations") is changed several times and the file is saved between changes. Because the text annotation object is not deleted, it retains the same object number and generation number as before. The updated copy of the object is included in the new update section added to the file.

The update's cross-reference section shall include a byte offset to this new copy of the object, overriding the old byte offset contained in the original cross-reference section. When a conforming reader reads the file, it shall build its cross-reference information in such a way that the most recent copy of each object shall be the one accessed from the file.

In versions of PDF 1.4 or later a conforming writer may use the **Version** entry in the document's catalog dictionary (see 7.7.2, "Document Catalog") to override the version specified in the header. A conforming writer may also need to update the Extensions dictionary, see 7.12, "Extensions Dictionary", if the update either deleted or added developer-defined extensions.

NOTE 4 The version entry enables the version to be altered when performing an incremental update.

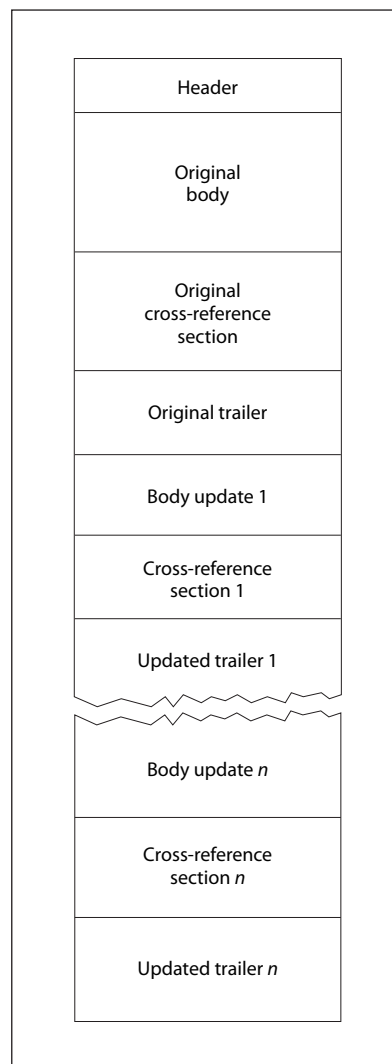


Figure 3 – Structure of an updated PDF file

7.5.7 Object Streams

An *object stream*, is a stream object in which a sequence of indirect objects may be stored, as an alternative to their being stored at the outermost file level.

NOTE 1 Object streams are first introduced in PDF 1.5. The purpose of object streams is to allow indirect objects other than streams to be stored more compactly by using the facilities provided by stream compression filters.

NOTE 2 The term “compressed object” is used regardless of whether the stream is actually encoded with a compression filter.

The following objects shall not be stored in an object stream:

- Stream objects
- Objects with a generation number other than zero
- A document’s encryption dictionary (see 7.6, "Encryption")
- An object representing the value of the **Length** entry in an object stream dictionary

- In linearized files (see Annex F), the document catalog, the linearization dictionary, and page objects shall not appear in an object stream.

NOTE 3 Indirect references to objects inside object streams use the normal syntax: for example, 14 0 R. Access to these objects requires a different way of storing cross-reference information; see 7.5.8, "Cross-Reference Streams." Use of compressed objects requires a PDF 1.5 conforming reader. However, compressed objects can be stored in a manner that a PDF 1.4 conforming reader can ignore.

In addition to the regular keys for streams shown in Table 5, the stream dictionary describing an object stream contains the following entries:

Table 16 – Additional entries specific to an object stream dictionary

key	type	description
Type	name	<i>(Required)</i> The type of PDF object that this dictionary describes; shall be ObjStm for an object stream.
N	integer	<i>(Required)</i> The number of indirect objects stored in the stream.
First	integer	<i>(Required)</i> The byte offset in the decoded stream of the first compressed object.
Extends	stream	<i>(Optional)</i> A reference to another object stream, of which the current object stream shall be considered an extension. Both streams are considered part of a <i>collection</i> of object streams (see below). A given collection consists of a set of streams whose Extends links form a directed acyclic graph.

A conforming writer determines which objects, if any, to store in object streams.

EXAMPLE 1 It can be useful to store objects having common characteristics together, such as "fonts on page 1," or "Comments for draft #3." These objects are known as a *collection*.

NOTE 4 To avoid a degradation of performance, such as would occur when downloading and decompressing a large object stream to access a single compressed object, the number of objects in an individual object stream should be limited. This may require a group of object streams to be linked as a collection, which can be done by means of the **Extends** entry in the object stream dictionary.

NOTE 5 **Extends** may also be used when a collection is being updated to include new objects. Rather than modifying the original object stream, which could entail duplicating much of the stream data, the new objects can be stored in a separate object stream. This is particularly important when adding an update section to a document.

The stream data in an object stream shall contain the following items:

- **N** pairs of integers separated by white space, where the first integer in each pair shall represent the object number of a compressed object and the second integer shall represent the byte offset in the decoded stream of that object, relative to the first object stored in the object stream, the value of the stream's first entry. The offsets shall be in increasing order.

NOTE 6 There is no restriction on the order of objects in the object stream; in particular, the objects need not be stored in object-number order.

- The value of the **First** entry in the stream dictionary shall be the byte offset in the decoded stream of the first object.
- The **N** objects are stored consecutively. Only the object values are stored in the stream; the **obj** and **endobj** keywords shall not be used.

NOTE 7 A compressed dictionary or array may contain indirect references.

An object in an object stream shall not consist solely of an object reference.

EXAMPLE 2 3 0 R

In an encrypted file (i.e., entire object stream is encrypted), strings occurring anywhere in an object stream shall not be separately encrypted.

A conforming writer shall store the first object immediately after the last byte offset. A conforming reader shall rely on the **First** entry in the stream dictionary to locate the first object.

An object stream itself, like any stream, shall be an indirect object, and therefore, there shall be an entry for it in a cross-reference table or cross-reference stream (see 7.5.8, "Cross-Reference Streams"), although there might not be any references to it (of the form 243 0 R).

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

EXAMPLE 3 The following shows three objects (two fonts and a font descriptor) as they would be represented in a PDF 1.4 or earlier file, along with a cross-reference table.

```

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

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

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

...

xref
0 32
0000000000 65535 f
... cross-reference entries for objects 1 through 10 ...
0000001434 00000 n
0000001735 00000 n
0000002155 00000 n
... cross-reference entries for objects 14 and on ...
trailer
  << /Size 32
    /Root ...
  >>

```

NOTE 8 For readability, the object stream has been shown unencoded. In a real PDF 1.5 file, Flate encoding would typically be used to gain the benefits of compression.

EXAMPLE 4 The following shows the same objects from the previous example stored in an object stream in a PDF 1.5 file, along with a cross-reference stream.

The cross-reference stream (see 7.5.8, "Cross-Reference Streams") contains entries for the fonts (objects 11 and 13) and the descriptor (object 12), which are compressed objects in an object stream. The first field of these entries is the entry type (2), the second field is the number of the object stream (15), and the third field is the position within the sequence of objects in the object stream (0, 1, and 2). The cross-reference stream also contains a type 1 entry for the object stream itself.

```

15 0 obj                                % The object stream
  << /Type /ObjStm
    /Length 1856
    /N 3                                % The number of objects in the stream
    /First 24                           % The byte offset in the decoded stream of the first object
  % The object numbers and offsets of the objects, relative to the first are shown on the first line of
  % the stream (i.e., 11 0 12 547 13 665).
  >>
stream
  11 0 12 547 13 665
  << /Type /Font
    /Subtype /TrueType
    ...other keys...
    /FontDescriptor 12 0 R
  >>

  << /Type /FontDescriptor
    /Ascent 891
    ...other keys...
    /FontFile2 22 0 R
  >>

  << /Type /Font
    /Subtype /Type0
    ...other keys...
    /ToUnicode 10 0 R
  >>
...
endstream
endobj

99 0 obj                                % The cross-reference stream
  << /Type /XRef
    /Index [0 32]                       % This section has one subsection with 32 objects
    /W [1 2 2]                          % Each entry has 3 fields: 1, 2 and 2 bytes in width,
                                         % respectively
    /Filter /ASCIISHexDecode             % For readability in this example
    /Size 32
    ...
  >>
stream
  00 0000 FFFF
  ... cross-references for objects 1 through 10 ...
  02 000F 0000
  02 000F 0001
  02 000F 0002
  ... cross-reference for object 14 ...
  01 BA5E 0000
  ...
endstream
endobj

startxref
  54321
%%EOF

```

NOTE 9 The number 54321 in Example 4 is the offset for object 99 0.

7.5.8 Cross-Reference Streams

7.5.8.1 General

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

- A more compact representation of cross-reference information
- The ability to access compressed objects that are stored in object streams (see 7.5.7, "Object Streams") and to allow new cross-reference entry types to be added in the future

Cross-reference streams are stream objects (see 7.3.8, "Stream Objects"), and contain a dictionary and a data stream. Each cross-reference stream contains the information equivalent to the cross-reference table (see 7.5.4, "Cross-Reference Table") and trailer (see 7.5.5, "File Trailer") for one cross-reference section.

EXAMPLE In this example, the trailer dictionary entries are stored in the stream dictionary, and the cross-reference table entries are stored as the stream data.

```
... objects ...

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

... more objects ...

startxref
byte_offset_of_cross-reference_stream (points to object 12)
%%EOF
```

The value following the **startxref** keyword shall be the offset of the cross-reference stream rather than the **xref** keyword. For files that use cross-reference streams entirely (that is, files that are not hybrid-reference files; see 7.5.8.4, "Compatibility with Applications That Do Not Support Compressed Reference Streams"), the keywords **xref** and **trailer** shall no longer be used. Therefore, with the exception of the **startxref address %%EOF** segment and comments, a file may be entirely a sequence of objects.

In linearized files (see F.3, "Linearized PDF Document Structure"), the document catalog, the linearization dictionary, and page objects shall not appear in an object stream.

7.5.8.2 Cross-Reference Stream Dictionary

Cross-reference streams may contain the entries shown in Table 17 in addition to the entries common to all streams (Table 5) and trailer dictionaries (Table 15). Since some of the information in the cross-reference stream is needed by the conforming reader to construct the index that allows indirect references to be resolved, the entries in cross-reference streams shall be subject to the following restrictions:

- The values of all entries shown in Table 17 shall be direct objects; indirect references shall not be permitted. For arrays (the **Index** and **W** entries), all of their elements shall be direct objects as well. If the stream is encoded, the **Filter** and **DecodeParms** entries in Table 5 shall also be direct objects.
- Other cross-reference stream entries not listed in Table 17 may be indirect; in fact, some (such as **Root** in Table 15) shall be indirect.

- The cross-reference stream shall not be encrypted and strings appearing in the cross-reference stream dictionary shall not be encrypted. It shall not have a **Filter** entry that specifies a **Crypt** filter (see 7.4.10, "Crypt Filter").

Table 17 – Additional entries specific to a cross-reference stream dictionary

key	type	description
Type	name	<i>(Required)</i> The type of PDF object that this dictionary describes; shall be XRef for a cross-reference stream.
Size	integer	<i>(Required)</i> The number one greater than the highest object number used in this section or in any section for which this shall be an update. It shall be equivalent to the Size entry in a trailer dictionary.
Index	array	<i>(Optional)</i> An array containing a pair of integers for each subsection in this section. The first integer shall be the first object number in the subsection; the second integer shall be the number of entries in the subsection The array shall be sorted in ascending order by object number. Subsections cannot overlap; an object number may have at most one entry in a section. Default value: [0 Size].
Prev	integer	<i>(Present only if the file has more than one cross-reference stream; not meaningful in hybrid-reference files; see 7.5.8.4, "Compatibility with Applications That Do Not Support Compressed Reference Streams")</i> The byte offset in the decoded stream from the beginning of the file to the beginning of the previous cross-reference stream. This entry has the same function as the Prev entry in the trailer dictionary (Table 15).
W	array	<i>(Required)</i> An array of integers representing the size of the fields in a single cross-reference entry. Table 18 describes the types of entries and their fields. For PDF 1.5, W always contains three integers; the value of each integer shall be the number of bytes (in the decoded stream) of the corresponding field. EXAMPLE [1 2 1] means that the fields are one byte, two bytes, and one byte, respectively. A value of zero for an element in the W array indicates that the corresponding field shall not be present in the stream, and the default value shall be used, if there is one. If the first element is zero, the type field shall not be present, and shall default to type 1. The sum of the items shall be the total length of each entry; it can be used with the Index array to determine the starting position of each subsection. Different cross-reference streams in a PDF file may use different values for W .

7.5.8.3 Cross-Reference Stream Data

Each entry in a cross-reference stream shall have one or more fields, the first of which designates the entry's type (see Table 18). In PDF 1.5 through PDF 1.7, only types 0, 1, and 2 are allowed. Any other value shall be interpreted as a reference to the null object, thus permitting new entry types to be defined in the future.

The fields are written in increasing order of field number; the length of each field shall be determined by the corresponding value in the **W** entry (see Table 17). Fields requiring more than one byte are stored with the high-order byte first.

Table 18 – Entries in a cross-reference stream

Type	Field	Description
0	1	The type of this entry, which shall be 0. Type 0 entries define the linked list of free objects (corresponding to f entries in a cross-reference table).
	2	The object number of the next free object.
	3	The generation number to use if this object number is used again.
1	1	The type of this entry, which shall be 1. Type 1 entries define objects that are in use but are not compressed (corresponding to n entries in a cross-reference table).
	2	The byte offset of the object, starting from the beginning of the file.
	3	The generation number of the object. Default value: 0.
2	1	The type of this entry, which shall be 2. Type 2 entries define compressed objects.
	2	The object number of the object stream in which this object is stored. (The generation number of the object stream shall be implicitly 0.)
	3	The index of this object within the object stream.

Like any stream, a cross-reference stream shall be an indirect object. Therefore, an entry for it shall exist in either a cross-reference stream (usually itself) or in a cross-reference table (in hybrid-reference files; see 7.5.8.4, "Compatibility with Applications That Do Not Support Compressed Reference Streams").

7.5.8.4 Compatibility with Applications That Do Not Support Compressed Reference Streams

Readers designed only to support versions of PDF before PDF 1.5, and hence do not support cross-reference streams, cannot access objects that are referenced by cross-reference streams. If a file uses cross-reference streams exclusively, it cannot be opened by such readers.

However, it is possible to construct a file called a *hybrid-reference* file that is readable by readers designed only to support versions of PDF before PDF 1.5. Such a file contains objects referenced by standard cross-reference tables in addition to objects in object streams that are referenced by cross-reference streams.

In these files, the trailer dictionary may contain, in addition to the entry for trailers shown in Table 15, an entry, as shown in Table 19. This entry may be ignored by readers designed only to support versions of PDF before PDF 1.5, which therefore have no access to entries in the cross-reference stream the entry refers to.

Table 19 – Additional entries in a hybrid-reference file's trailer dictionary

Key	Type	Value
XRefStm	integer	(<i>Optional</i>) The byte offset in the decoded stream from the beginning of the file of a cross-reference stream.

The **Size** entry of the trailer shall be large enough to include all objects, including those defined in the cross-reference stream referenced by the **XRefStm** entry. However, to allow random access, a main cross-reference section shall contain entries for all objects numbered 0 through **Size** - 1 (see 7.5.4, "Cross-Reference Table"). Therefore, the **XRefStm** entry shall not be used in the trailer dictionary of the main cross-reference section but only in an update cross-reference section.

When a conforming reader opens a hybrid-reference file, objects with entries in cross-reference streams are not hidden. When the conforming reader searches for an object, if an entry is not found in any given standard cross-reference section, the search shall proceed to a cross-reference stream specified by the **XRefStm** entry before looking in the previous cross-reference section (the **Prev** entry in the trailer).

Hidden objects, therefore, have two cross-reference entries. One is in the cross-reference stream. The other is a free entry in some previous section, typically the section referenced by the **Prev** entry. A conforming reader shall look in the cross-reference stream first, shall find the object there, and shall ignore the free entry in the previous section. A reader designed only to support versions of PDF before PDF 1.5 ignores the cross-reference stream and looks in the previous section, where it finds the free entry. The free entry shall have a next-generation number of 65535 so that the object number shall not be reused.

There are limitations on which objects in a hybrid-reference file can be hidden without making the file appear invalid to readers designed only to support versions of PDF before PDF 1.5. In particular, the root of the PDF file and the document catalog (see 7.7.2, "Document Catalog") shall not be hidden, nor any object that is *visible from the root*. Such objects can be determined by starting from the root and working recursively:

- In any dictionary that is visible, direct objects shall be visible. The value of any required key-value pair shall be visible.
- In any array that is visible, every element shall be visible.
- Resource dictionaries in content streams shall be visible. Although a resource dictionary is not required, strictly speaking, the content stream to which it is attached is assumed to contain references to the resources.

In general, the objects that may be hidden are optional objects specified by indirect references. A conforming reader can resolve those references by processing the cross-reference streams. In a reader designed only to support versions of PDF before PDF 1.5, the objects appear to be free, and the references shall be treated as references to the null object.

EXAMPLE 1 The **Outlines** entry in the catalog dictionary is optional. Therefore, its value may be an indirect reference to a hidden object. A reader designed only to support versions of PDF before PDF 1.5 treats it as a reference to the null object, which is equivalent to having omitted the entry entirely; a conforming reader recognizes it.

If the value of the **Outlines** entry is an indirect reference to a visible object, the entire outline tree shall be visible because nodes in the outline tree contain required pointers to other nodes.

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

EXAMPLE 2 In this example, an **ASCIIHexDecode** filter is specified to make the format and contents of the cross-reference stream readable.

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

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

One choice for a reader designed only to support versions of PDF before PDF 1.5 is to continue the search by following the **Prev** pointer to the main cross-reference table. That table defines object 3 as a free object, which is treated as the **null** object. Therefore, the entry is considered missing, and the document has no structure tree.

Another choice for a conforming reader, is to continue the search by following the **XRefStm** pointer to the cross-reference stream (object 11). It defines object 3 as a compressed object, stored at index 0 in the object stream (2 0 obj). Therefore, the document has a structure tree.

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

```
12 0 obj
...
endobj
...
99 0 obj
...
endobj
```

% The main xref section, at offset 2664 is next with entries for objects 0-99.
 % Objects 2 through 11 are marked free and objects 12, 13 and 99 are marked in use.

```
xref
0 100
0000000002 65535 f
0000000023 00000 n
0000000003 65535 f
0000000004 65535 f
0000000005 65535 f
0000000006 65535 f
0000000007 65535 f
0000000008 65535 f
0000000009 65535 f
0000000010 65535 f
0000000011 65535 f
0000000000 65535 f
0000000045 00000 n
0000000179 00000 n
... cross-reference entries for objects 14 through 98 ...
0000002201 00000 n
```

```
trailer
  << /Size 100
    /Root 1 0 R
    /ID ...
  >>
```

% The main xref section starts at offset 2664.

```
startxref
2664
%%EOF
```

```
2 0 obj                                % The object stream, at offset 3722
  << /Length ...
    /N 8                                % This stream contains 8 objects.
    /First 47                           % The stream-offset of the first object
  >>
```

```
stream
  3 0 4 50 5 72 ... the numbers and stream-offsets of the remaining 5 objects followed by dictionary
  objects 3-5 ...
  << /Type /StructTreeRoot
    /K 4 0 R
    /RoleMap 5 0 R
```

```

    /ClassMap 6 0 R
    /ParentTree 7 0 R
    /ParentTreeNextKey 8
  >>
  << /S /Workbook
    /P 8 0 R
    /K 9 0 R
  >>
  << /Workbook /Div
    /Worksheet /Sect
    /TextBox /Figure
    /Shape /Figure
  >>
  ... definitions for objects 6 through 10 ...
endstream
endobj

11 0 obj
  << /Type /XRef
    /Index [2 10]
    /Size 100
    /W [1 2 1]
    /Filter /ASCIIHexDecode
    ...
  >>
stream
  01 0E8A 0
  02 0002 00
  02 0002 01
  02 0002 02
  02 0002 03
  02 0002 04
  02 0002 05
  02 0002 06
  02 0002 07
  01 1323 0
endstream
endobj
% The entries above are for: object 2 (0x0E8A = 3722), object 3 (in object stream 2, index 0),
% object 4 (in object stream 2, index 1) ... object 10 (in object stream 2, index 7),
% object 11 (0x1323 = 4899).

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

```

The previous example illustrates several other points:

- The object stream is unencoded and the cross-reference stream uses an ASCII hexadecimal encoding for clarity. In practice, both streams should be Flate-encoded. PDF comments shall not be included in a cross-reference table or in cross-reference streams.
- The hidden objects, 2 through 11, are numbered consecutively. In practice, hidden objects and other free items in a cross-reference table need not be linked in ascending order until the end.

- The update cross-reference table need not contain any entries. A conforming writer that uses the hybrid-reference format creates the main cross-reference table, the update cross-reference table, and the cross-reference stream at the same time. Objects 12 and 13, for example, are not compressed. They might have entries in the update table. Since objects 2 and 11, the object stream and the cross-reference stream, are not compressed, they might also be defined in the update table. Since they are part of the hidden section, however, it makes sense to define them in the cross-reference stream.
- The update cross-reference section shall appear at the end of the file, but otherwise, there are no ordering restrictions on any of the objects or on the main cross-reference section. However, a file that uses both the hybrid-reference format and the linearized format has ordering requirements (see Annex F).

7.6 Encryption

7.6.1 General

A PDF document can be *encrypted* (PDF 1.1) to protect its contents from unauthorized access. Encryption applies to all strings and streams in the document's PDF file, with the following exceptions:

- The values for the ID entry in the trailer
- Any strings in an Encrypt dictionary
- Any strings that are inside streams such as content streams and compressed object streams, which themselves are encrypted

Encryption is not applied to other object types such as integers and boolean values, which are used primarily to convey information about the document's structure rather than its contents. Leaving these values unencrypted allows random access to the objects within a document, whereas encrypting the strings and streams protects the document's contents.

When a PDF stream object (see 7.3.8, "Stream Objects") refers to an external file, the stream's contents shall not be encrypted, since they are not part of the PDF file itself. However, if the contents of the stream are embedded within the PDF file (see 7.11.4, "Embedded File Streams"), they shall be encrypted like any other stream in the file. Beginning with PDF 1.5, embedded files can be encrypted in an otherwise unencrypted document (see 7.6.5, "Crypt Filters").

Encryption-related information shall be stored in a document's *encryption dictionary*, which shall be the value of the **Encrypt** entry in the document's trailer dictionary (see Table 15). The absence of this entry from the trailer dictionary means that a conforming reader shall consider the document to be not encrypted. The entries shown in Table 20 are common to all encryption dictionaries.

The encryption dictionary's **Filter** entry identifies the file's *security handler*, a software module that implements various aspects of the encryption process and controls access to the contents of the encrypted document. PDF specifies a standard password-based security handler that all conforming readers shall support, but conforming readers can optionally provide additional security handlers of their own.

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

The **V** entry, in specifying which algorithm to use, determines the length of the encryption key, on which the encryption (and decryption) of data in a PDF file shall be based. For **V** values 2 and 3, the **Length** entry specifies the exact length of the encryption key. In PDF 1.5, a value of 4 for **V** permits the security handler to use its own encryption and decryption algorithms and to specify *crypt filters* to use on specific streams (see 7.6.5, "Crypt Filters").

The remaining contents of the encryption dictionary shall be determined by the security handler and may vary from one handler to another. Entries for the standard security handler are described in 7.6.3, "Standard

Security Handler." Entries for public-key security handlers are described in 7.6.4, "Public-Key Security Handlers."

Table 20 – Entries common to all encryption dictionaries

Key	Type	Value
Filter	name	<p>(Required) The name of the preferred <i>security handler</i> for this document. It shall be the name of the security handler that was used to encrypt the document. If SubFilter is not present, only this security handler shall be used when opening the document. If it is present, a conforming reader can use any security handler that implements the format specified by SubFilter.</p> <p>Standard shall be the name of the built-in password-based security handler. Names for other security handlers may be registered by using the procedure described in Annex E.</p>
SubFilter	name	<p>(Optional; PDF 1.3) A name that completely specifies the format and interpretation of the contents of the encryption dictionary. It allows security handlers other than the one specified by Filter to decrypt the document. If this entry is absent, other security handlers shall not decrypt the document.</p> <p>NOTE This entry was introduced in PDF 1.3 to support the use of public-key cryptography in PDF files (see 7.6.4, "Public-Key Security Handlers"); however, it was not incorporated into the <i>PDF Reference</i> until the fourth edition (PDF 1.5).</p>
V	number	<p>(Optional) A code specifying the algorithm to be used in encrypting and decrypting the document:</p> <ul style="list-style-type: none"> 0 An algorithm that is undocumented. This value shall not be used. 1 "Algorithm 1: Encryption of data using the RC4 or AES algorithms" in 7.6.2, "General Encryption Algorithm," with an encryption key length of 40 bits; see below. 2 (PDF 1.4) "Algorithm 1: Encryption of data using the RC4 or AES algorithms" in 7.6.2, "General Encryption Algorithm," but permitting encryption key lengths greater than 40 bits. 3 (PDF 1.4) An unpublished algorithm that permits encryption key lengths ranging from 40 to 128 bits. This value shall not appear in a conforming PDF file. 4 (PDF 1.5) The security handler defines the use of encryption and decryption in the document, using the rules specified by the CF, StmF, and StrF entries. <p>The default value if this entry is omitted shall be 0, but when present should be a value of 1 or greater.</p>
Length	integer	<p>(Optional; PDF 1.4; only if V is 2 or 3) The length of the encryption key, in bits. The value shall be a multiple of 8, in the range 40 to 128. Default value: 40.</p>
CF	dictionary	<p>(Optional; meaningful only when the value of V is 4; PDF 1.5) A dictionary whose keys shall be crypt filter names and whose values shall be the corresponding crypt filter dictionaries (see Table 25). Every crypt filter used in the document shall have an entry in this dictionary, except for the standard crypt filter names (see Table 26).</p> <p>The conforming reader shall ignore entries in CF dictionary with the keys equal to those listed in Table 26 and use properties of the respective standard crypt filters.</p>
StmF	name	<p>(Optional; meaningful only when the value of V is 4; PDF 1.5) The name of the crypt filter that shall be used by default when decrypting streams. The name shall be a key in the CF dictionary or a standard crypt filter name specified in Table 26. All streams in the document, except for cross-reference streams (see 7.5.8, "Cross-Reference Streams") or streams that have a Crypt entry in their Filter array (see Table 6), shall be decrypted by the security handler, using this crypt filter.</p> <p>Default value: Identity.</p>

Table 20 – Entries common to all encryption dictionaries (continued)

Key	Type	Value
StrF	name	(Optional; meaningful only when the value of <i>V</i> is 4; PDF 1.5) The name of the crypt filter that shall be used when decrypting all strings in the document. The name shall be a key in the CF dictionary or a standard crypt filter name specified in Table 26. Default value: Identity .
EFF	name	(Optional; meaningful only when the value of <i>V</i> is 4; PDF 1.6) The name of the crypt filter that shall be used when encrypting embedded file streams that do not have their own crypt filter specifier; it shall correspond to a key in the CF dictionary or a standard crypt filter name specified in Table 26. This entry shall be provided by the security handler. Conforming writers shall respect this value when encrypting embedded files, except for embedded file streams that have their own crypt filter specifier. If this entry is not present, and the embedded file stream does not contain a crypt filter specifier, the stream shall be encrypted using the default stream crypt filter specified by StrmF .

Unlike strings within the body of the document, those in the encryption dictionary shall be direct objects. The contents of the encryption dictionary shall not be encrypted (the algorithm specified by the **V** entry). Security handlers shall be responsible for encrypting any data in the encryption dictionary that they need to protect.

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

7.6.2 General Encryption Algorithm

One of the following algorithms shall be used when encrypting data in a PDF file:

- A proprietary encryption algorithm known as RC4. RC4 is a symmetric stream cipher: the same algorithm shall be used for both encryption and decryption, and the algorithm does not change the length of the data. RC4 is a copyrighted, proprietary algorithm of RSA Security, Inc. Independent software vendors may be required to license RC4 to develop software that encrypts or decrypts PDF documents. For further information, visit the RSA Web site at <<http://www.rsasecurity.com>> or send e-mail to <products@rsasecurity.com>.
- The AES (Advanced Encryption Standard) algorithm (beginning with PDF 1.6). AES is a symmetric block cipher: the same algorithm shall be used for both encryption and decryption, and the length of the data when encrypted is rounded up to a multiple of the block size, which is fixed to always be 16 bytes, as specified in FIPS 197, *Advanced Encryption Standard (AES)*; see the Bibliography).

Strings and streams encrypted with AES shall use a padding scheme that is described in Internet RFC 2898, *PKCS #5: Password-Based Cryptography Specification Version 2.0*; see the Bibliography. For an original message length of *M*, the pad shall consist of $16 - (M \bmod 16)$ bytes whose value shall also be $16 - (M \bmod 16)$.

EXAMPLE A 9-byte message has a pad of 7 bytes, each with the value 0x07. The pad can be unambiguously removed to determine the original message length when decrypting. Note that the pad is present when *M* is evenly divisible by 16; it contains 16 bytes of 0x10.

PDF's standard encryption methods also make use of the MD5 message-digest algorithm for key generation purposes (described in Internet RFC 1321, *The MD5 Message-Digest Algorithm*; see the Bibliography).

The encryption of data in a PDF file shall be based on the use of an *encryption key* computed by the security handler. Different security handlers compute the encryption key using their own mechanisms. Regardless of how the key is computed, its use in the encryption of data shall always be the same (see "Algorithm 1:

Encryption of data using the RC4 or AES algorithms"). Because the RC4 algorithm and AES algorithms are symmetric, this same sequence of steps shall be used both to encrypt and to decrypt data.

Algorithms in 7.6, "Encryption" are uniquely numbered within that clause in a manner that maintains compatibility with previous documentation.

Algorithm 1: Encryption of data using the RC4 or AES algorithms

- a) Obtain the object number and generation number from the object identifier of the string or stream to be encrypted (see 7.3.10, "Indirect Objects"). If the string is a direct object, use the identifier of the indirect object containing it.
- b) For all strings and streams without crypt filter specifier; treating the object number and generation number as binary integers, extend the original n -byte encryption key to $n + 5$ bytes by appending the low-order 3 bytes of the object number and the low-order 2 bytes of the generation number in that order, low-order byte first. (n is 5 unless the value of **V** in the encryption dictionary is greater than 1, in which case n is the value of **Length** divided by 8.)

If using the AES algorithm, extend the encryption key an additional 4 bytes by adding the value "sAIT", which corresponds to the hexadecimal values 0x73, 0x41, 0x6C, 0x54. (This addition is done for backward compatibility and is not intended to provide additional security.)

- c) Initialize the MD5 hash function and pass the result of step (b) as input to this function.
- d) Use the first $(n + 5)$ bytes, up to a maximum of 16, of the output from the MD5 hash as the key for the RC4 or AES symmetric key algorithms, along with the string or stream data to be encrypted.

If using the AES algorithm, the Cipher Block Chaining (CBC) mode, which requires an initialization vector, is used. The block size parameter is set to 16 bytes, and the initialization vector is a 16-byte random number that is stored as the first 16 bytes of the encrypted stream or string.

The output is the encrypted data to be stored in the PDF file.

Stream data shall be encrypted after applying all stream encoding filters and shall be decrypted before applying any stream decoding filters. The number of bytes to be encrypted or decrypted shall be given by the **Length** entry in the stream dictionary. Decryption of strings (other than those in the encryption dictionary) shall be done after escape-sequence processing and hexadecimal decoding as appropriate to the string representation described in 7.3.4, "String Objects."

7.6.3 Standard Security Handler

7.6.3.1 General

PDF's standard security handler shall allow *access permissions* and up to two passwords to be specified for a document: an *owner password* and a *user password*. An application's decision to encrypt a document shall be based on whether the user creating the document specifies any passwords or access restrictions.

EXAMPLE A conforming writer may have a security settings dialog box that the user can invoke before saving the PDF file.

If passwords or access restrictions are specified, the document shall be encrypted, and the permissions and information required to validate the passwords shall be stored in the encryption dictionary. Documents in which only file attachments are encrypted shall use the same password as the *user* and *owner* password.

NOTE 1 A conforming writer may also create an encrypted document without any user interaction if it has some other source of information about what passwords and permissions to use.

If a user attempts to open an encrypted document that has a user password, the conforming reader shall first try to authenticate the encrypted document using the padding string defined in 7.6.3.3, "Encryption Key Algorithm" (default user password):

- If this authentication attempt is successful, the conforming reader may open, decrypt and display the document on the screen.
- If this authentication attempt fails, the application should prompt for a password. Correctly supplying either password (*owner* or *user* password) should enable the user to open the document, decrypt it, and display it on the screen.

Whether additional operations shall be allowed on a decrypted document depends on which password (if any) was supplied when the document was opened and on any access restrictions that were specified when the document was created:

- Opening the document with the correct *owner* password should allow full (owner) access to the document. This unlimited access includes the ability to change the document's passwords and access permissions.
- Opening the document with the correct *user* password (or opening a document with the default password) should allow additional operations to be performed according to the user access permissions specified in the document's encryption dictionary.

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

- Modifying the document's contents
- Copying or otherwise extracting text and graphics from the document, including extraction for accessibility purposes (that is, to make the contents of the document accessible through assistive technologies such as screen readers or Braille output devices; see 14.9, "Accessibility Support").
- Adding or modifying text annotations (see 12.5.6.4, "Text Annotations") and interactive form fields (see 12.7, "Interactive Forms")
- Printing the document

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

- Filling in forms (that is, filling in existing interactive form fields) and signing the document (which amounts to filling in existing signature fields, a type of interactive form field).
- Assembling the document: inserting, rotating, or deleting pages and creating navigation elements such as bookmarks or thumbnail images (see 12.3, "Document-Level Navigation").
- Printing to a representation from which a faithful digital copy of the PDF content could be generated. Disallowing such printing may result in degradation of output quality.

In addition, security handlers of revisions 3 and greater shall enable the extraction of text and graphics (in support of accessibility to users with disabilities or for other purposes) to be controlled separately.

If a security handler of revision 4 is specified, the standard security handler shall support crypt filters (see 7.6.5, "Crypt Filters"). The support shall be limited to the **Identity** crypt filter (see Table 26) and crypt filters named **StdCF** whose dictionaries contain a **CFM** value of **V2** or **AESV2** and an **AuthEvent** value of **DocOpen**. Public-Key security handlers in this case shall use crypt filters named **DefaultCryptFilter** when all document content is encrypted, and shall use crypt filters named **DefEmbeddedFile** when file attachments only are encrypted in

place of **StdCF** name. This nomenclature shall not be used as indicator of the type of the security handler or encryption.

Once the document has been opened and decrypted successfully, a conforming reader technically has access to the entire contents of the document. There is nothing inherent in PDF encryption that enforces the document permissions specified in the encryption dictionary. Conforming readers shall respect the intent of the document creator by restricting user access to an encrypted PDF file according to the permissions contained in the file.

NOTE 2 PDF 1.5 introduces a set of access permissions that do not require the document to be encrypted (see 12.8.4, "Permissions"). This enables limited access to a document when a user is not be able to respond to a prompt for a password. For example, there may be conforming readers that do not have a person running them such as printing off-line or on a server.

7.6.3.2 Standard Encryption Dictionary

Table 21 shows the encryption dictionary entries for the standard security handler (in addition to those in Table 20).

Table 21 – Additional encryption dictionary entries for the standard security handler

Key	Type	Value
R	number	<p>(Required) A number specifying which revision of the standard security handler shall be used to interpret this dictionary:</p> <p>2 if the document is encrypted with a V value less than 2 (see Table 20) and does not have any of the access permissions set to 0 (by means of the P entry, below) that are designated "Security handlers of revision 3 or greater" in Table 22</p> <p>3 if the document is encrypted with a V value of 2 or 3, or has any "Security handlers of revision 3 or greater" access permissions set to 0</p> <p>4 if the document is encrypted with a V value of 4</p>
O	string	<p>(Required) A 32-byte string, based on both the owner and user passwords, that shall be used in computing the encryption key and in determining whether a valid owner password was entered. For more information, see 7.6.3.3, "Encryption Key Algorithm," and 7.6.3.4, "Password Algorithms."</p>
U	string	<p>(Required) A 32-byte string, based on the user password, that shall be used in determining whether to prompt the user for a password and, if so, whether a valid user or owner password was entered. For more information, see 7.6.3.4, "Password Algorithms."</p>
P	integer	<p>(Required) A set of flags specifying which operations shall be permitted when the document is opened with user access (see Table 22).</p>
EncryptMetadata	boolean	<p>(Optional; meaningful only when the value of V is 4; PDF 1.5) Indicates whether the document-level metadata stream (see 14.3.2, "Metadata Streams") shall be encrypted. Conforming products should respect this value.</p> <p>Default value: true.</p>

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

The value of the **P** entry shall be interpreted as an unsigned 32-bit quantity containing a set of flags specifying which access permissions shall be granted when the document is opened with user access. Table 22 shows the meanings of these flags. Bit positions within the flag word shall be numbered from 1 (low-order) to 32 (high-order). A 1 bit in any position shall enable the corresponding access permission. Which bits shall be meaningful, and in some cases how they shall be interpreted, shall depend on the security handler's revision number (specified in the encryption dictionary's **R** entry).

Conforming readers shall ignore all flags other than those at bit positions 3, 4, 5, 6, 9, 10, 11, and 12.

NOTE PDF integer objects can be interpreted as binary values in a signed twos-complement form. Since all the reserved high-order flag bits in the encryption dictionary's **P** value are required to be 1, the integer value **P** shall be specified as a negative integer. For example, assuming revision 2 of the security handler, the value -44 permits printing and copying but disallows modifying the contents and annotations.

Table 22 – User access permissions

Bit position	Meaning
3	<i>(Security handlers of revision 2)</i> Print the document. <i>(Security handlers of revision 3 or greater)</i> Print the document (possibly not at the highest quality level, depending on whether bit 12 is also set).
4	Modify the contents of the document by operations other than those controlled by bits 6, 9, and 11.
5	<i>(Security handlers of revision 2)</i> Copy or otherwise extract text and graphics from the document, including extracting text and graphics (in support of accessibility to users with disabilities or for other purposes). <i>(Security handlers of revision 3 or greater)</i> Copy or otherwise extract text and graphics from the document by operations other than that controlled by bit 10.
6	Add or modify text annotations, fill in interactive form fields, and, if bit 4 is also set, create or modify interactive form fields (including signature fields).
9	<i>(Security handlers of revision 3 or greater)</i> Fill in existing interactive form fields (including signature fields), even if bit 6 is clear.
10	<i>(Security handlers of revision 3 or greater)</i> Extract text and graphics (in support of accessibility to users with disabilities or for other purposes).
11	<i>(Security handlers of revision 3 or greater)</i> Assemble the document (insert, rotate, or delete pages and create bookmarks or thumbnail images), even if bit 4 is clear.
12	<i>(Security handlers of revision 3 or greater)</i> Print the document to a representation from which a faithful digital copy of the PDF content could be generated. When this bit is clear (and bit 3 is set), printing is limited to a low-level representation of the appearance, possibly of degraded quality.

7.6.3.3 Encryption Key Algorithm

As noted earlier, one function of a security handler is to generate an encryption key for use in encrypting and decrypting the contents of a document. Given a password string, the standard security handler computes an encryption key as shown in "Algorithm 2: Computing an encryption key".

Algorithm 2: Computing an encryption key

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

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

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

- b) Initialize the MD5 hash function and pass the result of step (a) as input to this function.
- c) Pass the value of the encryption dictionary's **O** entry to the MD5 hash function. ("Algorithm 3: Computing the encryption dictionary's O (owner password) value" shows how the **O** value is computed.)
- d) Convert the integer value of the **P** entry to a 32-bit unsigned binary number and pass these bytes to the MD5 hash function, low-order byte first.
- e) Pass the first element of the file's file identifier array (the value of the **ID** entry in the document's trailer dictionary; see Table 15) to the MD5 hash function.

NOTE The first element of the ID array generally remains the same for a given document. However, in some situations, conforming writers may regenerate the ID array if a new generation of a document is created. Security handlers are encouraged not to rely on the ID in the encryption key computation.

- f) (*Security handlers of revision 4 or greater*) If document metadata is not being encrypted, pass 4 bytes with the value 0xFFFFFFFF to the MD5 hash function.
- g) Finish the hash.
- h) (*Security handlers of revision 3 or greater*) Do the following 50 times: Take the output from the previous MD5 hash and pass the first n bytes of the output as input into a new MD5 hash, where n is the number of bytes of the encryption key as defined by the value of the encryption dictionary's **Length** entry.
- i) Set the encryption key to the first n bytes of the output from the final MD5 hash, where n shall always be 5 for security handlers of revision 2 but, for security handlers of revision 3 or greater, shall depend on the value of the encryption dictionary's **Length** entry.

This algorithm, when applied to the user password string, produces the encryption key used to encrypt or decrypt string and stream data according to "Algorithm 1: Encryption of data using the RC4 or AES algorithms" in 7.6.2, "General Encryption Algorithm." Parts of this algorithm are also used in the algorithms described below.

7.6.3.4 Password Algorithms

In addition to the encryption key, the standard security handler shall provide the contents of the encryption dictionary (Table 20 and Table 21). The values of the **Filter**, **V**, **Length**, **R**, and **P** entries are straightforward, but the computation of the **O** (owner password) and **U** (user password) entries requires further explanation. The algorithms 3 through 7 that follow show how the values of the owner password and user password entries shall be computed (with separate versions of the latter depending on the revision of the security handler).

Algorithm 3: Computing the encryption dictionary's O (owner password) value

- a) Pad or truncate the owner password string as described in step (a) of "Algorithm 2: Computing an encryption key". If there is no owner password, use the user password instead.
- b) Initialize the MD5 hash function and pass the result of step (a) as input to this function.
- c) (*Security handlers of revision 3 or greater*) Do the following 50 times: Take the output from the previous MD5 hash and pass it as input into a new MD5 hash.

- d) Create an RC4 encryption key using the first n bytes of the output from the final MD5 hash, where n shall always be 5 for security handlers of revision 2 but, for security handlers of revision 3 or greater, shall depend on the value of the encryption dictionary's **Length** entry.
- e) Pad or truncate the user password string as described in step (a) of "Algorithm 2: Computing an encryption key".
- f) Encrypt the result of step (e), using an RC4 encryption function with the encryption key obtained in step (d).
- g) (*Security handlers of revision 3 or greater*) Do the following 19 times: Take the output from the previous invocation of the RC4 function and pass it as input to a new invocation of the function; use an encryption key generated by taking each byte of the encryption key obtained in step (d) and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 1 to 19).
- h) Store the output from the final invocation of the RC4 function as the value of the **O** entry in the encryption dictionary.

Algorithm 4: Computing the encryption dictionary's U (user password) value (Security handlers of revision 2)

- a) Create an encryption key based on the user password string, as described in "Algorithm 2: Computing an encryption key".
- b) Encrypt the 32-byte padding string shown in step (a) of "Algorithm 2: Computing an encryption key", using an RC4 encryption function with the encryption key from the preceding step.
- c) Store the result of step (b) as the value of the **U** entry in the encryption dictionary.

Algorithm 5: Computing the encryption dictionary's U (user password) value (Security handlers of revision 3 or greater)

- a) Create an encryption key based on the user password string, as described in "Algorithm 2: Computing an encryption key".
- b) Initialize the MD5 hash function and pass the 32-byte padding string shown in step (a) of "Algorithm 2: Computing an encryption key" as input to this function.
- c) Pass the first element of the file's file identifier array (the value of the **ID** entry in the document's trailer dictionary; see Table 15) to the hash function and finish the hash.
- d) Encrypt the 16-byte result of the hash, using an RC4 encryption function with the encryption key from step (a).
- e) Do the following 19 times: Take the output from the previous invocation of the RC4 function and pass it as input to a new invocation of the function; use an encryption key generated by taking each byte of the original encryption key obtained in step (a) and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 1 to 19).
- f) Append 16 bytes of arbitrary padding to the output from the final invocation of the RC4 function and store the 32-byte result as the value of the **U** entry in the encryption dictionary.

NOTE The standard security handler uses the algorithms 6 and 7 that follow, to determine whether a supplied password string is the correct user or owner password. Note too that algorithm 6 can be used to determine whether a document's user password is the empty string, and therefore whether to suppress prompting for a password when the document is opened.

Algorithm 6: Authenticating the user password

- a) Perform all but the last step of "Algorithm 4: Computing the encryption dictionary's U (user password) value (Security handlers of revision 2)" or "Algorithm 5: Computing the encryption dictionary's U (user password) value (Security handlers of revision 3 or greater)" using the supplied password string.
- b) If the result of step (a) is equal to the value of the encryption dictionary's **U** entry (comparing on the first 16 bytes in the case of security handlers of revision 3 or greater), the password supplied is the correct user password. The key obtained in step (a) (that is, in the first step of "Algorithm 4: Computing the encryption dictionary's U (user password) value (Security handlers of revision 2)" or "Algorithm 5: Computing the encryption dictionary's U (user password) value (Security handlers of revision 3 or greater)") shall be used to decrypt the document.

Algorithm 7: Authenticating the owner password

- a) Compute an encryption key from the supplied password string, as described in steps (a) to (d) of "Algorithm 3: Computing the encryption dictionary's O (owner password) value".
- b) (*Security handlers of revision 2 only*) Decrypt the value of the encryption dictionary's **O** entry, using an RC4 encryption function with the encryption key computed in step (a).

(*Security handlers of revision 3 or greater*) Do the following 20 times: Decrypt the value of the encryption dictionary's **O** entry (first iteration) or the output from the previous iteration (all subsequent iterations), using an RC4 encryption function with a different encryption key at each iteration. The key shall be generated by taking the original key (obtained in step (a)) and performing an XOR (exclusive or) operation between each byte of the key and the single-byte value of the iteration counter (from 19 to 0).

- c) The result of step (b) purports to be the user password. Authenticate this user password using "Algorithm 6: Authenticating the user password". If it is correct, the password supplied is the correct owner password.

7.6.4 Public-Key Security Handlers**7.6.4.1 General**

Security handlers may use *public-key* encryption technology to encrypt a document (or strings and streams within a document). When doing so, specifying one or more lists of recipients, where each list has its own unique access permissions may be done. Only specified recipients shall open the encrypted document or content, unlike the standard security handler, where a password determines access. The permissions defined for public-key security handlers are shown in Table 24 in 7.6.4.2, "Public-Key Encryption Dictionary".

Public-key security handlers use the industry standard Public Key Cryptographic Standard Number 7 (PKCS#7) binary encoding syntax to encode recipient list, decryption key, and access permission information. The PKCS#7 specification is in Internet RFC 2315, *PKCS #7: Cryptographic Message Syntax, Version 1.5* (see the Bibliography).

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

7.6.4.2 Public-Key Encryption Dictionary

Encryption dictionaries for public-key security handlers contain the common entries shown in Table 20, whose values are described above. In addition, they may contain the entry shown in Table 23 as described below.

The **Filter** entry shall be the name of a public-key security handler.

NOTE Examples of existing security handlers that support public-key encryption are **Entrust.PPKEF**, **Adobe.PPKLite**, and **Adobe.PubSec**. This handler will be the preferred handler when encrypting the document.

Permitted values of the **SubFilter** entry for use with conforming public-key security handlers are **adbe.pkcs7.s3**, **adbe.pkcs7.s4**, which shall be used when not using crypt filters (see 7.6.5, "Crypt Filters") and **adbe.pkcs7.s5**, which shall be used when using crypt filters.

The **CF**, **StmF**, and **StrF** entries may be present when **SubFilter** is **adbe.pkcs7.s5**.

Table 23 – Additional encryption dictionary entries for public-key security handlers

Key	Type	Value
Recipients	array	<p>(Required when SubFilter is adbe.pkcs7.s3 or adbe.pkcs7.s4; PDF 1.3) An array of byte-strings, where each string is a PKCS#7 object listing recipients who have been granted equal access rights to the document. The data contained in the PKCS#7 object shall include both a cryptographic key that shall be used to decrypt the encrypted data and the access permissions (see Table 24) that apply to the recipient list. There shall be only one PKCS#7 object per unique set of access permissions; if a recipient appears in more than one list, the permissions used shall be those in the first matching list.</p> <p>When SubFilter is adbe.pkcs7.s5, recipient lists shall be specified in the crypt filter dictionary; see Table 27.</p>
P	integer	<p>(Required) A set of flags specifying which operations shall be permitted when the document is opened with user access. If bit 2 is set to 1, all other bits are ignored and all operations are permitted. If bit 2 is set to 0, permission for operations are based on the values of the remaining flags defined in Table 24.</p>

The value of the **P** entry shall be interpreted as an unsigned 32-bit quantity containing a set of flags specifying which access permissions shall be granted when the document is opened with user access. Table 24 shows the meanings of these flags. Bit positions within the flag word shall be numbered from 1 (low-order) to 32 (high-order). A 1 bit in any position shall enable the corresponding access permission.

Conforming readers shall ignore all flags other than those at bit positions 2, 3, 4, 5, 6, 9, 10, 11, and 12.

Table 24 – Public-Key security handler user access permissions

Bit position	Meaning
2	When set permits change of encryption and enables all other permissions.
3	Print the document (possibly not at the highest quality level, depending on whether bit 12 is also set).
4	Modify the contents of the document by operations other than those controlled by bits 6, 9, and 11.
5	Copy or otherwise extract text and graphics from the document by operations other than that controlled by bit 10.
6	Add or modify text annotations, fill in interactive form fields, and, if bit 4 is also set, create or modify interactive form fields (including signature fields).
9	Fill in existing interactive form fields (including signature fields), even if bit 6 is clear.
10	Extract text and graphics (in support of accessibility to users with disabilities or for other purposes).

Table 24 – Public-Key security handler user access permissions (continued)

Bit position	Meaning
11	Assemble the document (insert, rotate, or delete pages and create bookmarks or thumbnail images), even if bit 4 is clear.
12	Print the document to a representation from which a faithful digital copy of the PDF content could be generated. When this bit is clear (and bit 3 is set), printing is limited to a low-level representation of the appearance, possibly of degraded quality.

7.6.4.3 Public-Key Encryption Algorithms

Figure 4 illustrates how PKCS#7 objects shall be used when encrypting PDF files. A PKCS#7 object is designed to encapsulate and encrypt what is referred to as the *enveloped data*.

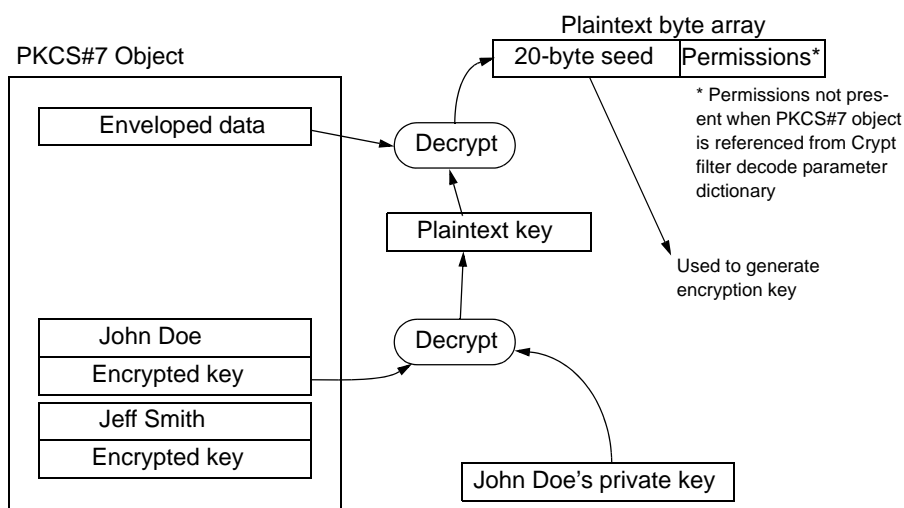


Figure 4 – Public-key encryption algorithm

The enveloped data in the PKCS#7 object contains keying material that shall be used to decrypt the document (or individual strings or streams in the document, when crypt filters are used; see 7.6.5, "Crypt Filters"). A key shall be used to encrypt (and decrypt) the enveloped data. This key (the *plaintext key* in Figure 4) shall be encrypted for each recipient, using that recipient's public key, and shall be stored in the PKCS#7 object (as the *encrypted key* for each recipient). To decrypt the document, that key shall be decrypted using the recipient's private key, which yields a decrypted (plaintext) key. That key, in turn, shall be used to decrypt the enveloped data in the PKCS#7 object, resulting in a byte array that includes the following information:

- A 20-byte seed that shall be used to create the encryption key that is used by "Algorithm 1: Encryption of data using the RC4 or AES algorithms". The seed shall be a unique random number generated by the security handler that encrypted the document.
- A 4-byte value defining the permissions, least significant byte first. See Table 24 for the possible permission values.
- When **SubFilter** is **adbe.pkcs7.s3**, the relevant permissions shall be only those specified for revision 2 of the standard security handler.
- For **adbe.pkcs7.s4**, security handlers of revision 3 permissions shall apply.
- For **adbe.pkcs7.s5**, which supports the use of crypt filters, the permissions shall be the same as **adbe.pkcs7.s4** when the crypt filter is referenced from the **StmF** or **StrF** entries of the encryption

dictionary. When referenced from the **Crypt** filter decode parameter dictionary of a stream object (see Table 14), the 4 bytes of permissions shall be absent from the enveloped data.

The algorithms that shall be used to encrypt the enveloped data in the PKCS#7 object are: RC4 with key lengths up to 256-bits, DES, Triple DES, RC2 with key lengths up to 128 bits, 128-bit AES in Cipher Block Chaining (CBC) mode, 192-bit AES in CBC mode, 256-bit AES in CBC mode. The PKCS#7 specification is in Internet RFC 2315, *PKCS #7: Cryptographic Message Syntax, Version 1.5* (see the Bibliography).

The encryption key used by "Algorithm 1: Encryption of data using the RC4 or AES algorithms" shall be calculated by means of an SHA-1 message digest operation that digests the following data, in order:

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

The first $n/8$ bytes of the resulting digest shall be used as the encryption key, where n is the bit length of the encryption key.

7.6.5 Crypt Filters

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

- The encryption dictionary (see Table 20) contains entries that enumerate the crypt filters in the document (**CF**) and specify which ones are used by default to decrypt all the streams (**StmF**) and strings (**StrF**) in the document. In addition, the value of the **V** entry shall be 4 to use crypt filters.
- Each crypt filter specified in the **CF** entry of the encryption dictionary shall be represented by a *crypt filter dictionary*, whose entries are shown in Table 25.
- A stream filter type, the **Crypt** filter (see 7.4.10, "Crypt Filter") can be specified for any stream in the document to override the default filter for streams. A conforming reader shall provide a standard **Identity** filter which shall pass the data unchanged (see Table 26) to allow specific streams, such as document metadata, to be unencrypted in an otherwise encrypted document. The stream's **DecodeParms** entry shall contain a **Crypt** filter decode parameters dictionary (see Table 14) whose **Name** entry specifies the particular crypt filter to be used (if missing, **Identity** is used). Different streams may specify different crypt filters.

Authorization to decrypt a stream shall always be obtained before the stream can be accessed. This typically occurs when the document is opened, as specified by a value of **DocOpen** for the **AuthEvent** entry in the crypt filter dictionary. Conforming readers and security handlers shall treat any attempt to access a stream for which authorization has failed as an error. **AuthEvent** can also be **EFOpen**, which indicates the presence of an embedded file that is encrypted with a crypt filter that may be different from the crypt filters used by default to encrypt strings and streams in the document.

In the file specification dictionary (see 7.11.3, "File Specification Dictionaries"), related files (**RF**) shall use the same crypt filter as the embedded file (**EF**).

A value of **None** for the **CFM** entry in the crypt filter dictionary allows the security handler to do its own decryption. This allows the handler to tightly control key management and use any preferred symmetric-key cryptographic algorithm.

Table 25 – Entries common to all crypt filter dictionaries

Key	Type	Value
Type	name	(Optional) If present, shall be CryptFilter for a crypt filter dictionary.
CFM	name	<p>(Optional) The method used, if any, by the conforming reader to decrypt data. The following values shall be supported:</p> <p>None The application shall not decrypt data but shall direct the input stream to the security handler for decryption.</p> <p>V2 The application shall ask the security handler for the encryption key and shall implicitly decrypt data with "Algorithm 1: Encryption of data using the RC4 or AES algorithms", using the RC4 algorithm.</p> <p>AESV2 (PDF 1.6) The application shall ask the security handler for the encryption key and shall implicitly decrypt data with "Algorithm 1: Encryption of data using the RC4 or AES algorithms", using the AES algorithm in Cipher Block Chaining (CBC) mode with a 16-byte block size and an initialization vector that shall be randomly generated and placed as the first 16 bytes in the stream or string.</p> <p>When the value is V2 or AESV2, the application may ask once for this encryption key and cache the key for subsequent use for streams that use the same crypt filter. Therefore, there shall be a one-to-one relationship between a crypt filter name and the corresponding encryption key.</p> <p>Only the values listed here shall be supported. Applications that encounter other values shall report that the file is encrypted with an unsupported algorithm.</p> <p>Default value: None.</p>
AuthEvent	name	<p>(Optional) The event to be used to trigger the authorization that is required to access encryption keys used by this filter. If authorization fails, the event shall fail. Valid values shall be:</p> <p>DocOpen: Authorization shall be required when a document is opened.</p> <p>EFOpen: Authorization shall be required when accessing embedded files.</p> <p>Default value: DocOpen.</p> <p>If this filter is used as the value of StrF or StmF in the encryption dictionary (see Table 20), the conforming reader shall ignore this key and behave as if the value is DocOpen.</p>
Length	integer	<p>(Optional) The bit length of the encryption key. It shall be a multiple of 8 in the range of 40 to 128.</p> <p>Security handlers may define their own use of the Length entry and should use it to define the bit length of the encryption key. Standard security handler expresses the length in multiples of 8 (16 means 128) and public-key security handler expresses it as is (128 means 128).</p>

Security handlers may add their own private data to crypt filter dictionaries. Names for private data entries shall conform to the PDF name registry (see Annex E).

Table 26 – Standard crypt filter names

Name	Description
Identity	Input data shall be passed through without any processing.

Table 27 lists the additional crypt filter dictionary entries used by public-key security handlers (see 7.6.4, "Public-Key Security Handlers"). When these entries are present, the value of **CFM** shall be **V2** or **AESV2**.

Table 27 – Additional crypt filter dictionary entries for public-key security handlers

Key	Type	Value
Recipients	array or string	<p><i>(Required)</i> If the crypt filter is referenced from StmF or StrF in the encryption dictionary, this entry shall be an array of byte strings, where each string shall be a binary-encoded PKCS#7 object that shall list recipients that have been granted equal access rights to the document. The enveloped data contained in the PKCS#7 object shall include both a 20-byte seed value that shall be used to compute the encryption key (see 7.6.4.3, "Public-Key Encryption Algorithms") followed by 4 bytes of permissions settings (see Table 22) that shall apply to the recipient list. There shall be only one object per unique set of access permissions. If a recipient appears in more than one list, the permissions used shall be those in the first matching list.</p> <p>If the crypt filter is referenced from a Crypt filter decode parameter dictionary (see Table 14), this entry shall be a string that shall be a binary-encoded PKCS#7 object that shall contain a list of all recipients who are permitted to access the corresponding encrypted stream. The enveloped data contained in the PKCS#7 object shall be a 20-byte seed value that shall be used to create the encryption key that shall be used by the algorithm in "Algorithm 1: Encryption of data using the RC4 or AES algorithms".</p>
EncryptMetadata	boolean	<p><i>(Optional; used only by crypt filters that are referenced from StmF in an encryption dictionary)</i> Indicates whether the document-level metadata stream (see 14.3.2, "Metadata Streams") shall be encrypted. Conforming readers shall respect this value when determining whether metadata shall be encrypted. The value of the EncryptMetadata entry is set by the security handler rather than the conforming reader.</p> <p>Default value: true.</p>

EXAMPLE The following shows the use of crypt filters in an encrypted document containing a plaintext document-level metadata stream. The metadata stream is left as is by applying the **Identity** crypt filter. The remaining streams and strings are decrypted using the default filters.

```
%PDF-1.5
1 0 obj                                % Document catalog
<< /Type /Catalog
    /Pages 2 0 R
    /Metadata 6 0 R
>>
endobj
2 0 obj                                % Page tree
<< /Type /Pages
    /Kids [3 0 R]
    /Count 1
>>
endobj
3 0 obj                                % 1st page
<< /Type /Page
```

```

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

```

7.7 Document Structure

7.7.1 General

A PDF document can be regarded as a hierarchy of objects contained in the body section of a PDF file. At the root of the hierarchy is the document's *catalog* dictionary (see 7.7.2, "Document Catalog").

NOTE Most of the objects in the hierarchy are dictionaries. Figure 5 illustrates the structure of the object hierarchy.

EXAMPLE Each page of the document is represented by a *page object*—a dictionary that includes references to the page's contents and other attributes, such as its thumbnail image (12.3.4, "Thumbnail Images") and any annotations (12.5, "Annotations") associated with it. The individual page objects are tied together in a structure called the *page tree* (described in 7.7.3, "Page Tree"), which in turn is specified by an indirect reference in the document catalog. Parent, child, and sibling relationships within the hierarchy are defined by dictionary entries whose values are indirect references to other dictionaries.

The data structures described in this sub-clause, particularly the **Catalog** and **Page** dictionaries, combine entries describing document structure with ones dealing with the detailed semantics of documents and pages. All entries are listed here, but many of their descriptions are deferred to subsequent sub-clauses.

7.7.2 Document Catalog

The root of a document's object hierarchy is the *catalog* dictionary, located by means of the **Root** entry in the trailer of the PDF file (see 7.5.5, "File Trailer"). The catalog contains references to other objects defining the document's contents, outline, article threads, named destinations, and other attributes. In addition, it contains information about how the document shall be displayed on the screen, such as whether its outline and thumbnail page images shall be displayed automatically and whether some location other than the first page shall be shown when the document is opened. Table 28 shows the entries in the catalog dictionary.

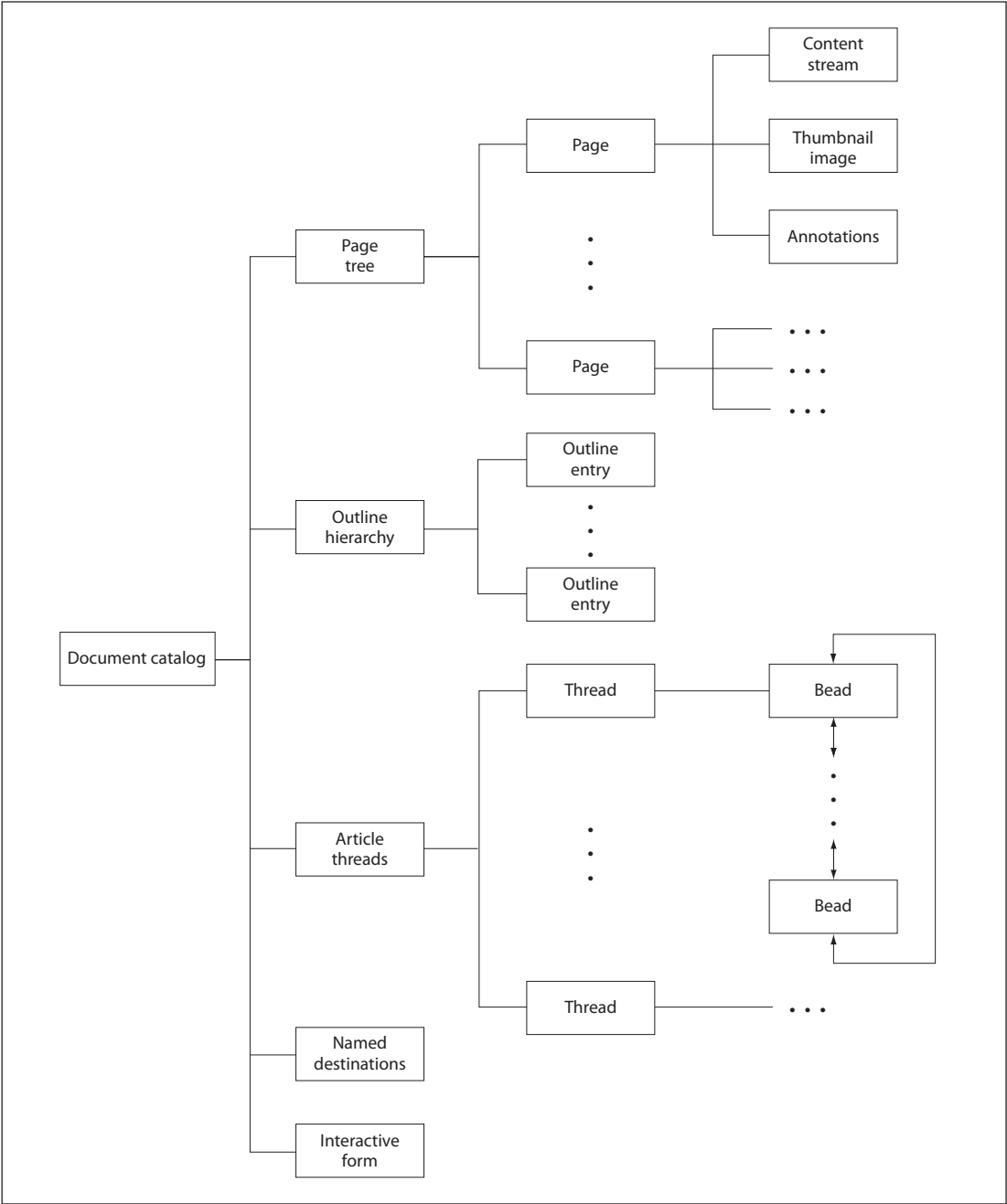


Figure 5 – Structure of a PDF document

Table 28 – Entries in the catalog dictionary

Key	Type	Value
Type	name	<i>(Required)</i> The type of PDF object that this dictionary describes; shall be Catalog for the catalog dictionary.
Version	name	<i>(Optional; PDF 1.4)</i> The version of the PDF specification to which the document conforms (for example, 1.4) if later than the version specified in the file's header (see 7.5.2, "File Header"). If the header specifies a later version, or if this entry is absent, the document shall conform to the version specified in the header. This entry enables a conforming writer to update the version using an incremental update; see 7.5.6, "Incremental Updates." The value of this entry shall be a name object, not a number, and therefore shall be preceded by a SOLIDUS (2Fh) character (/) when written in the PDF file (for example, /1.4).
Extensions	dictionary	<i>(Optional; ISO 32000)</i> An extensions dictionary containing developer prefix identification and version numbers for developer extensions that occur in this document. 7.12, "Extensions Dictionary", describes this dictionary and how it shall be used.
Pages	dictionary	<i>(Required; shall be an indirect reference)</i> The <i>page tree node</i> that shall be the root of the document's <i>page tree</i> (see 7.7.3, "Page Tree").
PageLabels	number tree	<i>(Optional; PDF 1.3)</i> A number tree (see 7.9.7, "Number Trees") defining the page labelling for the document. The keys in this tree shall be page indices; the corresponding values shall be <i>page label dictionaries</i> (see 12.4.2, "Page Labels"). Each page index shall denote the first page in a <i>labelling range</i> to which the specified page label dictionary applies. The tree shall include a value for page index 0.
Names	dictionary	<i>(Optional; PDF 1.2)</i> The document's name dictionary (see 7.7.4, "Name Dictionary").
Dests	dictionary	<i>(Optional; PDF 1.1; shall be an indirect reference)</i> A dictionary of names and corresponding <i>destinations</i> (see 12.3.2.3, "Named Destinations").
ViewerPreferences	dictionary	<i>(Optional; PDF 1.2)</i> A viewer preferences dictionary (see 12.2, "Viewer Preferences") specifying the way the document shall be displayed on the screen. If this entry is absent, conforming readers shall use their own current user preference settings.
PageLayout	name	<i>(Optional)</i> A name object specifying the page layout shall be used when the document is opened: SinglePage Display one page at a time OneColumn Display the pages in one column TwoColumnLeft Display the pages in two columns, with odd-numbered pages on the left TwoColumnRight Display the pages in two columns, with odd-numbered pages on the right TwoPageLeft <i>(PDF 1.5)</i> Display the pages two at a time, with odd-numbered pages on the left TwoPageRight <i>(PDF 1.5)</i> Display the pages two at a time, with odd-numbered pages on the right Default value: SinglePage.

Table 28 – Entries in the catalog dictionary (continued)

Key	Type	Value
PageMode	name	<p>(Optional) A name object specifying how the document shall be displayed when opened:</p> <p>UseNone Neither document outline nor thumbnail images visible</p> <p>UseOutlines Document outline visible</p> <p>UseThumbs Thumbnail images visible</p> <p>FullScreen Full-screen mode, with no menu bar, window controls, or any other window visible</p> <p>UseOC (PDF 1.5) Optional content group panel visible</p> <p>UseAttachments (PDF 1.6) Attachments panel visible</p> <p>Default value: UseNone.</p>
Outlines	dictionary	(Optional; shall be an indirect reference) The outline dictionary that shall be the root of the document's <i>outline hierarchy</i> (see 12.3.3, "Document Outline").
Threads	array	(Optional; PDF 1.1; shall be an indirect reference) An array of thread dictionaries that shall represent the document's <i>article threads</i> (see 12.4.3, "Articles").
OpenAction	array or dictionary	(Optional; PDF 1.1) A value specifying a <i>destination</i> that shall be displayed or an <i>action</i> that shall be performed when the document is opened. The value shall be either an array defining a destination (see 12.3.2, "Destinations") or an action dictionary representing an action (12.6, "Actions"). If this entry is absent, the document shall be opened to the top of the first page at the default magnification factor.
AA	dictionary	(Optional; PDF 1.4) An additional-actions dictionary defining the actions that shall be taken in response to various <i>trigger events</i> affecting the document as a whole (see 12.6.3, "Trigger Events").
URI	dictionary	(Optional; PDF 1.1) A URI dictionary containing document-level information for <i>URI (uniform resource identifier) actions</i> (see 12.6.4.7, "URI Actions").
AcroForm	dictionary	(Optional; PDF 1.2) The document's <i>interactive form (AcroForm) dictionary</i> (see 12.7.2, "Interactive Form Dictionary").
Metadata	stream	(Optional; PDF 1.4; shall be an indirect reference) A <i>metadata stream</i> that shall contain metadata for the document (see 14.3.2, "Metadata Streams").
StructTreeRoot	dictionary	(Optional; PDF 1.3) The document's structure tree root dictionary (see 14.7.2, "Structure Hierarchy").
MarkInfo	dictionary	(Optional; PDF 1.4) A mark information dictionary that shall contain information about the document's usage of Tagged PDF conventions (see 14.7, "Logical Structure").
Lang	text string	(Optional; PDF 1.4) A <i>language identifier</i> that shall specify the natural language for all text in the document except where overridden by language specifications for structure elements or marked content (see 14.9.2, "Natural Language Specification"). If this entry is absent, the language shall be considered unknown.
SpiderInfo	dictionary	(Optional; PDF 1.3) A Web Capture information dictionary that shall contain state information used by any Web Capture extension (see 14.10.2, "Web Capture Information Dictionary").

Table 28 – Entries in the catalog dictionary (continued)

Key	Type	Value
OutputIntents	array	(Optional; PDF 1.4) An array of output intent dictionaries that shall specify the colour characteristics of output devices on which the document might be rendered (see 14.11.5, "Output Intents").
PieceInfo	dictionary	(Optional; PDF 1.4) A page-piece dictionary associated with the document (see 14.5, "Page-Piece Dictionaries").
OCProperties	dictionary	(Optional; PDF 1.5; required if a document contains optional content) The document's optional content properties dictionary (see 8.11.4, "Configuring Optional Content").
Perms	dictionary	(Optional; PDF 1.5) A permissions dictionary that shall specify user access permissions for the document. 12.8.4, "Permissions", describes this dictionary and how it shall be used.
Legal	dictionary	(Optional; PDF 1.5) A dictionary that shall contain attestations regarding the content of a PDF document, as it relates to the legality of digital signatures (see 12.8.5, "Legal Content Attestations").
Requirements	array	(Optional; PDF 1.7) An array of requirement dictionaries that shall represent requirements for the document. 12.10, "Document Requirements", describes this dictionary and how it shall be used.
Collection	dictionary	(Optional; PDF 1.7) A collection dictionary that a conforming reader shall use to enhance the presentation of file attachments stored in the PDF document. (see 12.3.5, "Collections").
NeedsRendering	boolean	(Optional; PDF 1.7) A flag used to expedite the display of PDF documents containing XFA forms. It specifies whether the document shall be regenerated when the document is first opened. See the <i>XML Forms Architecture (XFA) Specification</i> (Bibliography). Default value: false .

EXAMPLE The following shows a sample catalog object.

```

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

```

7.7.3 Page Tree

7.7.3.1 General

The pages of a document are accessed through a structure known as the *page tree*, which defines the ordering of pages in the document. Using the tree structure, conforming readers using only limited memory, can quickly open a document containing thousands of pages. The tree contains nodes of two types—intermediate nodes, called *page tree nodes*, and leaf nodes, called *page objects*—whose form is described in the subsequent sub-clauses. Conforming products shall be prepared to handle any form of tree structure built of such nodes.

NOTE The simplest structure can consist of a single page tree node that references all of the document's page objects directly. However, to optimize application performance, a conforming writer can construct trees of a particular form, known as *balanced trees*. Further information on this form of tree can be found in *Data Structures and Algorithms*, by Aho, Hopcroft, and Ullman (see the Bibliography).

7.7.3.2 Page Tree Nodes

Table 29 shows the entries in a page tree node that shall always be present (Required).

Table 29 – Required entries in a page tree node

Key	Type	Value
Type	name	<i>(Required)</i> The type of PDF object that this dictionary describes; shall be Pages for a page tree node.
Parent	dictionary	<i>(Required except in root node; prohibited in the root node; shall be an indirect reference)</i> The page tree node that is the immediate parent of this one.
Kids	array	<i>(Required)</i> An array of indirect references to the immediate children of this node. The children shall only be page objects or other page tree nodes.
Count	integer	<i>(Required)</i> The number of leaf nodes (page objects) that are descendants of this node within the page tree.

NOTE The structure of the page tree is not necessarily related to the logical structure of the document; that is, page tree nodes do not represent chapters, sections, and so forth. Other data structures are defined for this purpose; see 14.7, "Logical Structure".

Conforming products shall not be required to preserve the existing structure of the page tree.

EXAMPLE The following illustrates the page tree for a document with three pages. See 7.7.3.3, "Page Objects," for the contents of the individual page objects, and H.5, "Page Tree Example", for a more extended example showing the page tree for a longer document.

```

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

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

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

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

```

In addition to the entries shown in Table 29, a page tree node may contain further entries defining *inherited attributes* for the page objects that are its descendants (see 7.7.3.4, "Inheritance of Page Attributes").

7.7.3.3 Page Objects

The leaves of the page tree are *page objects*, each of which is a dictionary specifying the attributes of a single page of the document. Table 30 shows the contents of this dictionary. The table also identifies which attributes a page may inherit from its ancestor nodes in the page tree, as described under 7.7.3.4, "Inheritance of Page Attributes." Attributes that are not explicitly identified in the table as inheritable shall not be inherited.

Table 30 – Entries in a page object

Key	Type	Value
Type	name	<i>(Required)</i> The type of PDF object that this dictionary describes; shall be Page for a page object.
Parent	dictionary	<i>(Required; shall be an indirect reference)</i> The page tree node that is the immediate parent of this page object.
LastModified	date	<i>(Required if PieceInfo is present; optional otherwise; PDF 1.3)</i> The date and time (see 7.9.4, "Dates") when the page's contents were most recently modified. If a page-piece dictionary (PieceInfo) is present, the modification date shall be used to ascertain which of the application data dictionaries that it contains correspond to the current content of the page (see 14.5, "Page-Piece Dictionaries").
Resources	dictionary	<i>(Required; inheritable)</i> A dictionary containing any resources required by the page (see 7.8.3, "Resource Dictionaries"). If the page requires no resources, the value of this entry shall be an empty dictionary. Omitting the entry entirely indicates that the resources shall be inherited from an ancestor node in the page tree.
MediaBox	rectangle	<i>(Required; inheritable)</i> A rectangle (see 7.9.5, "Rectangles"), expressed in default user space units, that shall define the boundaries of the physical medium on which the page shall be displayed or printed (see 14.11.2, "Page Boundaries").
CropBox	rectangle	<i>(Optional; inheritable)</i> A rectangle, expressed in default user space units, that shall define the visible region of default user space. When the page is displayed or printed, its contents shall be clipped (cropped) to this rectangle and then shall be imposed on the output medium in some implementation-defined manner (see 14.11.2, "Page Boundaries"). Default value: the value of MediaBox .
BleedBox	rectangle	<i>(Optional; PDF 1.3)</i> A rectangle, expressed in default user space units, that shall define the region to which the contents of the page shall be clipped when output in a production environment (see 14.11.2, "Page Boundaries"). Default value: the value of CropBox .
TrimBox	rectangle	<i>(Optional; PDF 1.3)</i> A rectangle, expressed in default user space units, that shall define the intended dimensions of the finished page after trimming (see 14.11.2, "Page Boundaries"). Default value: the value of CropBox .
ArtBox	rectangle	<i>(Optional; PDF 1.3)</i> A rectangle, expressed in default user space units, that shall define the extent of the page's meaningful content (including potential white space) as intended by the page's creator (see 14.11.2, "Page Boundaries"). Default value: the value of CropBox .
BoxColorInfo	dictionary	<i>(Optional; PDF 1.4)</i> A box colour information dictionary that shall specify the colours and other visual characteristics that should be used in displaying guidelines on the screen for the various page boundaries (see 14.11.2.2, "Display of Page Boundaries"). If this entry is absent, the application shall use its own current default settings.

Table 30 – Entries in a page object (continued)

Key	Type	Value
Contents	stream or array	<p>(Optional) A <i>content stream</i> (see 7.8.2, "Content Streams") that shall describe the contents of this page. If this entry is absent, the page shall be empty.</p> <p>The value shall be either a single stream or an array of streams. If the value is an array, the effect shall be as if all of the streams in the array were concatenated, in order, to form a single stream. Conforming writers can create image objects and other resources as they occur, even though they interrupt the content stream. The division between streams may occur only at the boundaries between lexical tokens (see 7.2, "Lexical Conventions") but shall be unrelated to the page's logical content or organization. Applications that consume or produce PDF files need not preserve the existing structure of the Contents array. Conforming writers shall not create a Contents array containing no elements.</p>
Rotate	integer	<p>(Optional; inheritable) The number of degrees by which the page shall be rotated clockwise when displayed or printed. The value shall be a multiple of 90. Default value: 0.</p>
Group	dictionary	<p>(Optional; PDF 1.4) A <i>group attributes dictionary</i> that shall specify the attributes of the page's page group for use in the transparent imaging model (see 11.4.7, "Page Group" and 11.6.6, "Transparency Group XObjects").</p>
Thumb	stream	<p>(Optional) A stream object that shall define the page's <i>thumbnail image</i> (see 12.3.4, "Thumbnail Images").</p>
B	array	<p>(Optional; PDF 1.1; recommended if the page contains article beads) An array that shall contain indirect references to all <i>article beads</i> appearing on the page (see 12.4.3, "Articles"). The beads shall be listed in the array in natural reading order.</p> <p>NOTE The information in this entry can be created or recreated from the information obtained from the Threads key in the Catalog.</p>
Dur	number	<p>(Optional; PDF 1.1) The page's <i>display duration</i> (also called its <i>advance timing</i>): the maximum length of time, in seconds, that the page shall be displayed during presentations before the viewer application shall automatically advance to the next page (see 12.4.4, "Presentations"). By default, the viewer shall not advance automatically.</p>
Trans	dictionary	<p>(Optional; PDF 1.1) A <i>transition dictionary</i> describing the transition effect that shall be used when displaying the page during presentations (see 12.4.4, "Presentations").</p>
Annots	array	<p>(Optional) An array of <i>annotation dictionaries</i> that shall contain indirect references to all annotations associated with the page (see 12.5, "Annotations").</p>
AA	dictionary	<p>(Optional; PDF 1.2) An <i>additional-actions dictionary</i> that shall define actions to be performed when the page is opened or closed (see 12.6.3, "Trigger Events").</p> <p>(PDF 1.3) additional-actions dictionaries are not inheritable.</p>
Metadata	stream	<p>(Optional; PDF 1.4) A <i>metadata stream</i> that shall contain metadata for the page (see 14.3.2, "Metadata Streams").</p>
PieceInfo	dictionary	<p>(Optional; PDF 1.3) A <i>page-piece dictionary</i> associated with the page (see 14.5, "Page-Piece Dictionaries").</p>

Table 30 – Entries in a page object (continued)

Key	Type	Value
StructParents	integer	<i>(Required if the page contains structural content items; PDF 1.3)</i> The integer key of the page's entry in the <i>structural parent tree</i> (see 14.7.4.4, "Finding Structure Elements from Content Items").
ID	byte string	<i>(Optional; PDF 1.3; indirect reference preferred)</i> The digital identifier of the page's parent <i>Web Capture content set</i> (see 14.10.6, "Object Attributes Related to Web Capture").
PZ	number	<i>(Optional; PDF 1.3)</i> The page's preferred <i>zoom (magnification) factor</i> : the factor by which it shall be scaled to achieve the natural display magnification (see 14.10.6, "Object Attributes Related to Web Capture").
SeparationInfo	dictionary	<i>(Optional; PDF 1.3)</i> A <i>separation dictionary</i> that shall contain information needed to generate colour separations for the page (see 14.11.4, "Separation Dictionaries").
Tabs	name	<i>(Optional; PDF 1.5)</i> A name specifying the tab order that shall be used for annotations on the page. The possible values shall be R (row order), C (column order), and S (structure order). See 12.5, "Annotations" for details.
TemplateInstantiated	name	<i>(Required if this page was created from a named page object; PDF 1.5)</i> The name of the originating page object (see 12.7.6, "Named Pages").
PresSteps	dictionary	<i>(Optional; PDF 1.5)</i> A <i>navigation node dictionary</i> that shall represent the first node on the page (see 12.4.4.2, "Sub-page Navigation").
UserUnit	number	<i>(Optional; PDF 1.6)</i> A positive number that shall give the size of default user space units, in multiples of 1/72 inch. The range of supported values shall be implementation-dependent. Default value: 1.0 (user space unit is 1/72 inch).
VP	dictionary	<i>(Optional; PDF 1.6)</i> An array of <i>viewport dictionaries</i> (see Table 260) that shall specify rectangular regions of the page.

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

```

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

```

7.7.3.4
Inheritance of Page Attributes

Some of the page attributes shown in Table 30 are designated as *inheritable*. If such an attribute is omitted from a page object, its value shall be inherited from an ancestor node in the page tree. If the attribute is a required one, a value shall be supplied in an ancestor node. If the attribute is optional and no inherited value is specified, the default value shall be used.

An attribute can thus be defined once for a whole set of pages by specifying it in an intermediate page tree node and arranging the pages that share the attribute as descendants of that node.

EXAMPLE A document may specify the same media box for all of its pages by including a **MediaBox** entry in the root node of the page tree. If necessary, an individual page object may override this inherited value with a **MediaBox** entry of its own.

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

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

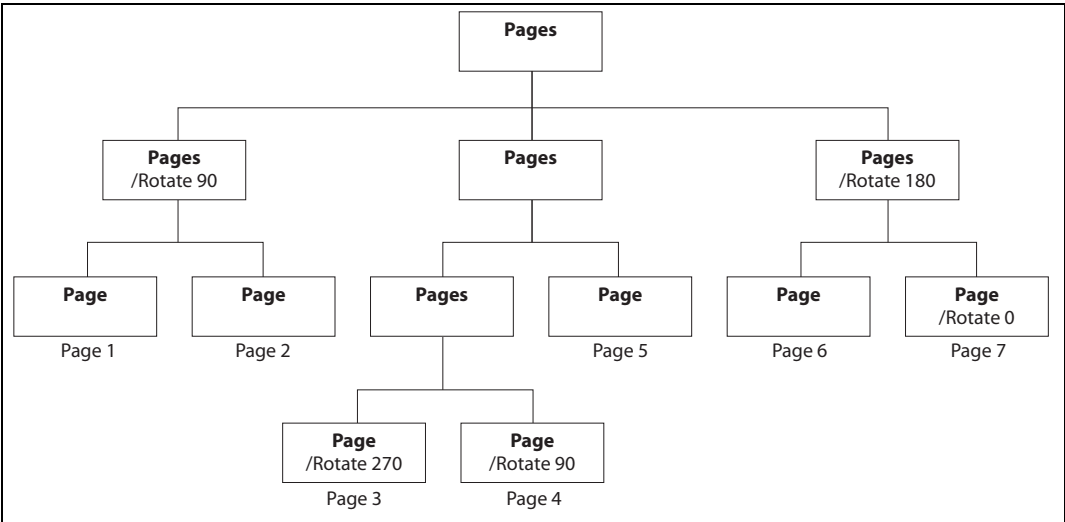


Figure 6 – Inheritance of attributes

7.7.4
Name Dictionary

Some categories of objects in a PDF file can be referred to by name rather than by object reference. The correspondence between names and objects is established by the document's *name dictionary* (*PDF 1.2*), located by means of the **Names** entry in the document's catalog (see 7.7.2, "Document Catalog"). Each entry in this dictionary designates the root of a name tree (see 7.9.6, "Name Trees") defining names for a particular category of objects. Table 31 shows the contents of the name dictionary.

Table 31 – Entries in the name dictionary

Key	Type	Value
Dests	name tree	(Optional; PDF 1.2) A name tree mapping name strings to destinations (see 12.3.2.3, "Named Destinations").
AP	name tree	(Optional; PDF 1.3) A name tree mapping name strings to annotation appearance streams (see 12.5.5, "Appearance Streams").

Table 31 – Entries in the name dictionary (continued)

Key	Type	Value
JavaScript	name tree	(Optional; PDF 1.3) A name tree mapping name strings to document-level JavaScript actions (see 12.6.4.16, "JavaScript Actions").
Pages	name tree	(Optional; PDF 1.3) A name tree mapping name strings to visible pages for use in interactive forms (see 12.7.6, "Named Pages").
Templates	name tree	(Optional; PDF 1.3) A name tree mapping name strings to invisible (template) pages for use in interactive forms (see 12.7.6, "Named Pages").
IDS	name tree	(Optional; PDF 1.3) A name tree mapping digital identifiers to Web Capture content sets (see 14.10.4, "Content Sets").
URLS	name tree	(Optional; PDF 1.3) A name tree mapping uniform resource locators (URLs) to Web Capture content sets (see 14.10.4, "Content Sets").
EmbeddedFiles	name tree	(Optional; PDF 1.4) A name tree mapping name strings to file specifications for embedded file streams (see 7.11.4, "Embedded File Streams").
AlternatePresentations	name tree	(Optional; PDF 1.4) A name tree mapping name strings to alternate presentations (see 13.5, "Alternate Presentations").
Renditions	name tree	(Optional; PDF 1.5) A name tree mapping name strings (which shall have Unicode encoding) to rendition objects (see 13.2.3, "Renditions").

7.8 Content Streams and Resources

7.8.1 General

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

7.8.2 Content Streams

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

Each page of a document shall be represented by one or more content streams. Content streams shall also be used to package sequences of instructions as self-contained graphical elements, such as forms (see 8.10, "Form XObjects"), patterns (8.7, "Patterns"), certain fonts (9.6.5, "Type 3 Fonts"), and annotation appearances (12.5.5, "Appearance Streams").

A content stream, after decoding with any specified filters, shall be interpreted according to the PDF syntax rules described in 7.2, "Lexical Conventions." It consists of PDF objects denoting operands and operators. The operands needed by an operator shall precede it in the stream. See EXAMPLE 4 in 7.4, "Filters," for an example of a content stream.

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

An *operator* is a PDF keyword specifying some action that shall be performed, such as painting a graphical shape on the page. An operator keyword shall be distinguished from a name object by the absence of an initial SOLIDUS character (2Fh) (/). Operators shall be meaningful only inside a content stream.

NOTE 1 This postfix notation, in which an operator is preceded by its operands, is superficially the same as in the PostScript language. However, PDF has no concept of an operand stack as PostScript has.

In PDF, all of the operands needed by an operator shall immediately precede that operator. Operators do not return results, and operands shall not be left over when an operator finishes execution.

NOTE 2 Most operators have to do with painting graphical elements on the page or with specifying parameters that affect subsequent painting operations. The individual operators are described in the clauses devoted to their functions:

Clause 8, "Graphics" describes operators that paint general graphics, such as filled areas, strokes, and sampled images, and that specify device-independent graphical parameters, such as colour.

Clause 9, "Text" describes operators that paint text using character glyphs defined in fonts.

Clause 10, "Rendering" describes operators that specify device-dependent rendering parameters.

Clause 14, "Document Interchange" describes the marked-content operators that associate higher-level logical information with objects in the content stream. These operators do not affect the rendered appearance of the content; they specify information useful to applications that use PDF for document interchange.

Ordinarily, when a conforming reader encounters an operator in a content stream that it does not recognize, an error shall occur. A pair of compatibility operators, **BX** and **EX** (*PDF 1.1*), shall modify this behaviour (see Table 32). These operators shall occur in pairs and may be nested. They bracket a *compatibility section*, a portion of a content stream within which unrecognized operators shall be ignored without error. This mechanism enables a conforming writer to use operators defined in later versions of PDF without sacrificing compatibility with older applications. It should be used only in cases where ignoring such newer operators is the appropriate thing to do. The **BX** and **EX** operators are not themselves part of any graphics object (see 8.2, "Graphics Objects") or of the graphics state (8.4, "Graphics State").

Table 32 – Compatibility operators

Operands	Operator	Description
—	BX	(<i>PDF 1.1</i>) Begin a compatibility section. Unrecognized operators (along with their operands) shall be ignored without error until the balancing EX operator is encountered.
—	EX	(<i>PDF 1.1</i>) End a compatibility section begun by a balancing BX operator. Ignore any unrecognized operands and operators from previous matching BX onward.

7.8.3 Resource Dictionaries

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

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

A content stream's named resources shall be defined by a *resource dictionary*, which shall enumerate the named resources needed by the operators in the content stream and the names by which they can be referred to.

EXAMPLE 1 If a text operator appearing within the content stream needs a certain font, the content stream's resource dictionary can associate the name F42 with the corresponding font dictionary. The text operator can use this name to refer to the font.

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

- For a content stream that is the value of a page's **Contents** entry (or is an element of an array that is the value of that entry), the resource dictionary shall be designated by the page dictionary's **Resources** or is inherited, as described under 7.7.3.4, "Inheritance of Page Attributes," from some ancestor node of the page object.
- For other content streams, a conforming writer shall include a **Resources** entry in the stream's dictionary specifying the resource dictionary which contains all the resources used by that content stream. This shall apply to content streams that define form XObjects, patterns, Type 3 fonts, and annotation.
- PDF files written obeying earlier versions of PDF may have omitted the **Resources** entry in all form XObjects and Type 3 fonts used on a page. All resources that are referenced from those forms and fonts shall be inherited from the resource dictionary of the page on which they are used. This construct is obsolete and should not be used by conforming writers.

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

Each key in a resource dictionary shall be the name of a resource type, as shown in Table 33. The corresponding values shall be as follows:

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

Table 33 – Entries in a resource dictionary

Key	Type	Value
ExtGState	dictionary	(Optional) A dictionary that maps resource names to graphics state parameter dictionaries (see 8.4.5, "Graphics State Parameter Dictionaries").
ColorSpace	dictionary	(Optional) A dictionary that maps each resource name to either the name of a device-dependent colour space or an array describing a colour space (see 8.6, "Colour Spaces").
Pattern	dictionary	(Optional) A dictionary that maps resource names to pattern objects (see 8.7, "Patterns").
Shading	dictionary	(Optional; PDF 1.3) A dictionary that maps resource names to shading dictionaries (see 8.7.4.3, "Shading Dictionaries").
XObject	dictionary	(Optional) A dictionary that maps resource names to external objects (see 8.8, "External Objects").
Font	dictionary	(Optional) A dictionary that maps resource names to font dictionaries (see clause 9, "Text").
ProcSet	array	(Optional) An array of predefined procedure set names (see 14.2, "Procedure Sets").

Table 33 – Entries in a resource dictionary (continued)

Key	Type	Value
Properties	dictionary	(Optional; PDF 1.2) A dictionary that maps resource names to property list dictionaries for marked content (see 14.6.2, "Property Lists").

EXAMPLE 2 The following shows a resource dictionary containing procedure sets, fonts, and external objects. The procedure sets are specified by an array, as described in 14.2, "Procedure Sets". The fonts are specified with a subdictionary associating the names F5, F6, F7, and F8 with objects 6, 8, 10, and 12, respectively. Likewise, the **XObject** subdictionary associates the names Im1 and Im2 with objects 13 and 15, respectively.

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

7.9 Common Data Structures

7.9.1 General

As mentioned at the beginning of this clause, there are some general-purpose data structures that are built from the basic object types described in 7.3, "Objects," and are used in many places throughout PDF. This sub-clause describes data structures for text strings, dates, rectangles, name trees, and number trees. More complex data structures are described in 7.10, "Functions," and 7.11, "File Specifications."

All of these data structures are meaningful only as part of the document hierarchy; they may not appear within content streams. In particular, the special conventions for interpreting the values of string objects apply only to strings outside content streams. An entirely different convention is used within content streams for using strings to select sequences of glyphs to be painted on the page (see clause 9, "Text"). Table 34 summarizes the basic and higher-level data types that are used throughout this standard to describe the values of dictionary entries and other PDF data values.

Table 34 – PDF data types

Type	Description	Sub-Clause
ASCII string	Bytes containing ASCII characters	7.9.2 7.9.2.4
array	Array object	7.3.6
boolean	Boolean value	7.3.2
byte string	A series of bytes that shall represent characters or other binary data. If such a type represents characters, the encoding shall be determined by the context.	7.9.2
date	Date (ASCII string)	7.9.4
dictionary	Dictionary object	7.3.7
file specification	File specification (string or dictionary)	7.11

Table 34 – PDF data types (continued)

Type	Description	Sub-Clause
function	Function (dictionary or stream)	7.10
integer	Integer number	
name	Name object	7.3.5
name tree	Name tree (dictionary)	7.9.6
null	Null object	7.3.9
number	Number (integer or real)	
number tree	Number tree (dictionary)	7.9.7
PDFDocEncoded string	Bytes containing a string that shall be encoded using PDFDocEncoding	7.9.2
rectangle	Rectangle (array)	7.9.5
stream	Stream object	7.3.8
string	Any string that is not a text string. Beginning with PDF 1.7, this type is further qualified as the types: PDFDocEncoded string, ASCII string, and byte string.	7.9.2
text string	Bytes that represent characters that shall be encoded using either PDFDocEncoding or UTF-16BE with a leading byte-order marker (as defined in "Text String Type" on page 86.)	7.9.2.2 7.9.2
text stream	Text stream	7.9.3

7.9.2 String Object Types

7.9.2.1 General

PDF supports one fundamental string object (see 7.3.4, "String Objects"). The string object shall be further qualified as a text string, a PDFDocEncoded string, ASCII string, or byte string. The further qualification reflects the encoding used to represent the characters or glyphs described by the string.

Table 35 summarizes the string object types that represent data encoded using specific conventions.

Table 35 – String Object Types

Type	Description
text string	Shall be used for human-readable text, such as text annotations, bookmark names, article names, and document information. These strings shall be encoded using either PDFDocEncoding or UTF-16BE with a leading byte-order marker. This type is described in 7.9.2.2, "Text String Type."
PDFDocEncoded string	Shall be used for characters and glyphs that are represented in a single byte, using PDFDocEncoding. This type is described in 7.9.2.3, "PDFDocEncoded String Type."

Table 35 – String Object Types (continued)

Type	Description
ASCII string	Shall be used for characters that are represented in a single byte using ASCII encoding.
byte string	Shall be used for binary data represented as a series of bytes, where each byte can be any value representable in 8 bits. The string may represent characters but the encoding is not known. The bytes of the string need not represent characters. This type shall be used for data such as MD5 hash values, signature certificates, and Web Capture identification values. This type is described in 7.9.2.4, "Byte String Type."

The string types described in Table 35 specify increasingly specific encoding schemes, as shown in Figure 7.

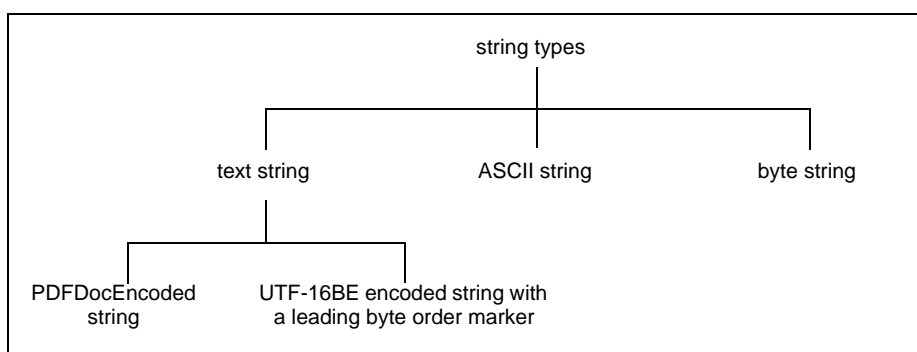


Figure 7 – Relationship between string types

7.9.2.2 Text String Type

The text string type shall be used for character strings that contain information intended to be human-readable, such as text annotations, bookmark names, article names, document information, and so forth.

NOTE 1 Text string type is a subtype of string type and represents data encoded using specific conventions.

The text string type shall be used for character strings that shall be encoded in either **PDFDocEncoding** or the UTF-16BE Unicode character encoding scheme. **PDFDocEncoding** can encode all of the ISO Latin 1 character set and is documented in Annex D. UTF-16BE can encode all Unicode characters. UTF-16BE and Unicode character encoding are described in the *Unicode Standard* by the Unicode Consortium (see the Bibliography).

NOTE 2 **PDFDocEncoding** does not support all Unicode characters whereas UTF-16BE does.

For text strings encoded in Unicode, the first two bytes shall be 254 followed by 255. These two bytes represent the Unicode byte order marker, U+FEFF, indicating that the string is encoded in the UTF-16BE (big-endian) encoding scheme specified in the Unicode standard.

NOTE 3 This mechanism precludes beginning a string using **PDFDocEncoding** with the two characters thorn ydieresis, which is unlikely to be a meaningful beginning of a word or phrase.

Conforming readers that process PDF files containing Unicode text strings shall be prepared to handle supplementary characters; that is, characters requiring more than two bytes to represent.

An escape sequence may appear anywhere in a Unicode text string to indicate the language in which subsequent text shall be written.

NOTE 4 This is useful when the language cannot be determined from the character codes used in the text.

The escape sequence shall consist of the following elements, in order:

- a) The Unicode value U+001B (that is, the byte sequence 0 followed by 27).
- b) A 2- byte ISO 639 language code.

EXAMPLE en for English or ja for Japanese encoded as ASCII characters.

- c) *(Optional)* A 2-byte ISO 3166 country code.

EXAMPLE US for the United States or JP for Japan.

- d) The Unicode value U+001B.

NOTE 5 The complete list of codes defined by ISO 639 and ISO 3166 can be obtained from the International Organization for Standardization (see the Bibliography).

7.9.2.3 PDFDocEncoded String Type

A PDFDocEncoded string is a character string in which the characters shall be represented in a single byte using PDFDocEncoding.

NOTE **PDFDocEncoding** does not support all Unicode characters whereas UTF-16BE does.

7.9.2.4 Byte String Type

The byte string type shall be used for binary data that shall be represented as a series of bytes, where each byte may be any value representable in 8 bits. Byte string type is a subtype of string type.

NOTE The string may represent characters but the encoding is not known. The bytes of the string may not represent characters.

7.9.3 Text Streams

A *text stream* (PDF 1.5) shall be a PDF stream object (7.3.8, "Stream Objects") whose unencoded bytes shall meet the same requirements as a text string (7.9.2.2, "Text String Type") with respect to encoding, byte order, and lead bytes.

7.9.4 Dates

Date values used in a PDF shall conform to a standard date format, which closely follows that of the international standard ASN.1 (Abstract Syntax Notation One), defined in ISO/IEC 8824. A date shall be a text string of the form

(D: YYYYMMDDHHmmSSOHH' mm)

where:

YYYY shall be the year

MM shall be the month (01–12)

DD shall be the day (01–31)

HH shall be the hour (00–23)

mm shall be the minute (00–59)

SS shall be the second (00–59)

O shall be the relationship of local time to Universal Time (UT), and shall be denoted by one of the characters PLUS SIGN (U+002B) (+), HYPHEN-MINUS (U+002D) (-), or LATIN CAPITAL LETTER Z (U+005A) (Z) (see below)

HH followed by APOSTROPHE (U+0027) (') shall be the absolute value of the offset from UT in hours (00–23)

mm shall be the absolute value of the offset from UT in minutes (00–59)

The prefix *D:* shall be present, the year field (YYYY) shall be present and all other fields may be present but only if all of their preceding fields are also present. The APOSTROPHE following the hour offset field (HH) shall only be present if the HH field is present. The minute offset field (mm) shall only be present if the APOSTROPHE following the hour offset field (HH) is present. The default values for *MM* and *DD* shall be both 01; all other numerical fields shall default to zero values. A PLUS SIGN as the value of the *O* field signifies that local time is later than UT, a HYPHEN-MINUS signifies that local time is earlier than UT, and the LATIN CAPITAL LETTER Z signifies that local time is equal to UT. If no UT information is specified, the relationship of the specified time to UT shall be considered to be GMT. Regardless of whether the time zone is specified, the rest of the date shall be specified in local time.

EXAMPLE For example, December 23, 1998, at 7:52 PM, U.S. Pacific Standard Time, is represented by the string *D:199812231952-08'00*

7.9.5 Rectangles

Rectangles are used to describe locations on a page and bounding boxes for a variety of objects. A rectangle shall be written as an array of four numbers giving the coordinates of a pair of diagonally opposite corners.

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

Typically, the array takes the form

$[ll_x \ ll_y \ ur_x \ ur_y]$

specifying the lower-left *x*, lower-left *y*, upper-right *x*, and upper-right *y* coordinates of the rectangle, in that order. The other two corners of the rectangle are then assumed to have coordinates (ll_x, ur_y) and (ur_x, ll_y) .

7.9.6 Name Trees

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

- Unlike the keys in a dictionary, which are name objects, those in a name tree are strings.
- The keys are ordered.
- The values associated with the keys may be objects of any type. Stream objects shall be specified by indirect object references (7.3.8, "Stream Objects"). The dictionary, array, and string objects should be specified by indirect object references, and other PDF objects (nulls, numbers, booleans, and names) should be specified as direct objects.
- The data structure can represent an arbitrarily large collection of key-value pairs, which can be looked up efficiently without requiring the entire data structure to be read from the PDF file. (In contrast, a dictionary can be subject to an implementation limit on the number of entries it can contain.)

A name tree shall be constructed of *nodes*, each of which shall be a dictionary object. Table 36 shows the entries in a node dictionary. The nodes shall be of three kinds, depending on the specific entries they contain. The tree shall always have exactly one *root node*, which shall contain a single entry: either **Kids** or **Names** but not both. If the root node has a **Names** entry, it shall be the only node in the tree. If it has a **Kids** entry, each of the remaining nodes shall be either an *intermediate node*, that shall contain a **Limits** entry and a **Kids** entry, or a *leaf node*, that shall contain a **Limits** entry and a **Names** entry.

Table 36 – Entries in a name tree node dictionary

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

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

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

EXAMPLE 1 The following is an abbreviated outline, showing object numbers and nodes, of a name tree that maps the names of all the chemical elements, from actinium to zirconium, to their atomic numbers.

Example of a name tree

1: Root node

2: Intermediate node: Actinium to Gold

5: Leaf node: Actinium = 25, ..., Astatine = 31

25: Integer: 89

...

31: Integer: 85

...

11: Leaf node: Gadolinium = 56, ..., Gold = 59

56: Integer: 64

...

59: Integer: 79

3: Intermediate node: Hafnium to Protactinium

12: Leaf node: Hafnium = 60, ..., Hydrogen = 65

60: Integer: 72

...

65: Integer: 1

...

```

19: Leaf node: Palladium = 92, ..., Protactinium = 100
    92: Integer: 46
    ...
    100: Integer: 91
4: Intermediate node: Radium to Zirconium
    20: Leaf node: Radium = 101, ..., Ruthenium = 107
        101: Integer: 89
        ...
        107: Integer: 85
    ...
24: Leaf node: Xenon = 129, ..., Zirconium = 133
    129: Integer: 54
    ...
    133: Integer: 40

```

EXAMPLE 2 The following shows the representation of this tree in a PDF file

```

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

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

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

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

5 0 obj
  << /Limits [(Actinium) (Astatine)]
      /Names [ (Actinium) 25 0 R
            ]
  >>
endobj
% Leaf node

```

```

        (Aluminum) 26 0 R
        (Americium) 27 0 R
        (Antimony) 28 0 R
        (Argon) 29 0 R
        (Arsenic) 30 0 R
        (Astatine) 31 0 R
    ]
>>
endobj

...

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

25 0 obj
  89
endobj
% Atomic number (Actinium)

...

133 0 obj
  40
endobj
% Atomic number (Zirconium)

```

7.9.7 Number Trees

A *number tree* is similar to a name tree (see 7.9.6, "Name Trees"), except that its keys shall be integers instead of strings and shall be sorted in ascending numerical order. The entries in the leaf (or root) nodes containing the key-value pairs shall be named **Nums** instead of **Names** as in a name tree. Table 37 shows the entries in a number tree's node dictionaries.

Table 37 – Entries in a number tree node dictionary

Key	Type	Value
Kids	array	<i>(Root and intermediate nodes only; required in intermediate nodes; present in the root node if and only if Nums is not present)</i> Shall be an array of indirect references to the immediate children of this node. The children may be intermediate or leaf nodes.
Nums	array	<i>(Root and leaf nodes only; shall be required in leaf nodes; present in the root node if and only if Kids is not present)</i> Shall be an array of the form $[key_1\ value_1\ key_2\ value_2\ \dots\ key_n\ value_n]$ where each key_i is an integer and the corresponding $value_i$ shall be the object associated with that key. The keys shall be sorted in numerical order, analogously to the arrangement of keys in a name tree as described in 7.9.6, "Name Trees."
Limits	array	<i>(Shall be present in Intermediate and leaf nodes only)</i> Shall be an array of two integers, that shall specify the (numerically) least and greatest keys included in the Nums array of a leaf node or in the Nums arrays of any leaf nodes that are descendants of an intermediate node.

7.10 Functions

7.10.1 General

PDF is not a programming language, and a PDF file is not a program. However, PDF provides several types of *function objects* (PDF 1.2) that represent parameterized classes of functions, including mathematical formulas and sampled representations with arbitrary resolution.

NOTE 1 Functions may be used in various ways in PDF, including device-dependent rasterization information for high-quality printing (halftone spot functions and transfer functions), colour transform functions for certain colour spaces, and specification of colours as a function of position for smooth shadings.

Functions in PDF represent static, self-contained numerical transformations.

NOTE 2 A function to add two numbers has two input values and one output value:

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

Similarly, a function that computes the arithmetic and geometric mean of two numbers can be viewed as a function of two input values and two output values:

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

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

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

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

Each function definition includes a *domain*, the set of legal values for the input. Some types of functions also define a *range*, the set of legal values for the output. Input values passed to the function shall be clipped to the domain, and output values produced by the function shall be clipped to the range.

EXAMPLE Suppose the following function is defined with a domain of [-1 1]. If the function is called with the input value 6, that value is replaced with the nearest value in the defined domain, 1, before the function is evaluated; the resulting output value is therefore 3.

$$f(x) = x + 2$$

Similarly, if the following function is defined with a range of [0 100], and if the input values -6 and 4 are passed to the function (and are within its domain), then the output value produced by the function, -14, is replaced with 0, the nearest value in the defined range.

$$f(x_0, x_1) = 3 \times x_0 + x_1$$

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

- (PDF 1.2) A *sampled function* (type 0) uses a table of *sample values* to define the function. Various techniques are used to interpolate values between the sample values; see 7.10.2, "Type 0 (Sampled) Functions."