

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](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](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](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.

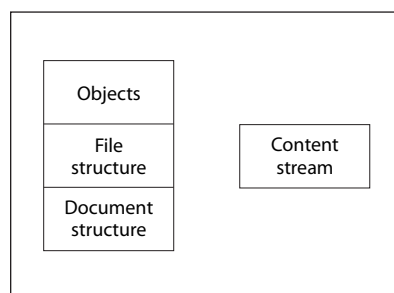
## 7 Syntax

### 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-Various***Characters?	A;Name_With-Various***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
<b>Length</b>	integer	<i>(Required)</i> The number of bytes from the beginning of the line following the keyword <b>stream</b> to the last byte just before the keyword <b>endstream</b> . (There may be an additional EOL marker, preceding <b>endstream</b> , 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.
<b>Filter</b>	name or array	<i>(Optional)</i> The name of a filter that shall be applied in processing the stream data found between the keywords <b>stream</b> and <b>endstream</b> , or an array of zero, one or several names. Multiple filters shall be specified in the order in which they are to be applied.
<b>DecodeParms</b>	dictionary or array	<i>(Optional)</i> A parameter dictionary or an array of such dictionaries, used by the filters specified by <b>Filter</b> . If there is only one filter and that filter has parameters, <b>DecodeParms</b> shall be set to the filter's parameter dictionary unless all the filter's parameters have their default values, in which case the <b>DecodeParms</b> entry may be omitted. If there are multiple filters and any of the filters has parameters set to nondefault values, <b>DecodeParms</b> 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 <b>DecodeParms</b> entry may be omitted.
<b>F</b>	file specification	<i>(Optional; PDF 1.2)</i> The file containing the stream data. If this entry is present, the bytes between <b>stream</b> and <b>endstream</b> shall be ignored. However, the <b>Length</b> entry should still specify the number of those bytes (usually, there are no bytes and <b>Length</b> is 0). The filters that are applied to the file data shall be specified by <b>FFilter</b> and the filter parameters shall be specified by <b>FDecodeParms</b> .
<b>FFilter</b>	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 <b>Filter</b> .
<b>FDecodeParms</b>	dictionary or array	<i>(Optional; PDF 1.2)</i> A parameter dictionary, or an array of such dictionaries, used by the filters specified by <b>FFilter</b> . The same rules apply as for <b>DecodeParms</b> .

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**

<b>FILTER name</b>	<b>Parameters</b>	<b>Description</b>
<b>ASCIIHexDecode</b>	no	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
<b>ASCII85Decode</b>	no	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
<b>LZWDecode</b>	yes	Decompresses data encoded using the LZW (Lempel-Ziv-Welch) adaptive compression method, reproducing the original text or binary data.
<b>FlateDecode</b>	yes	(PDF 1.2) Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
<b>RunLengthDecode</b>	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).
<b>CCITTFaxDecode</b>	yes	Decompresses data encoded using the CCITT facsimile standard, reproducing the original data (typically monochrome image data at 1 bit per pixel).
<b>JBIG2Decode</b>	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).
<b>DCTDecode</b>	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.
<b>JPXDecode</b>	no	(PDF 1.5) Decompresses data encoded using the wavelet-based JPEG2000 standard, reproducing the original image data.
<b>Crypt</b>	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$bCDF/(3fL5]Oq)^kJZ!C2H1
'TO]RI?Q:&'<5&iP!$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)IKn06I1apKDC@`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
<b>Predictor</b>	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 <b>Predictor</b> selects the predictor algorithm. For more information regarding <b>Predictor</b> values greater than 1, see 7.4.4.4, "LZW and Flate Predictor Functions." Default value: 1.
<b>Colors</b>	integer	(May be used only if <b>Predictor</b> 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
<b>BitsPerComponent</b>	integer	(May be used only if <b>Predictor</b> 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.
<b>Columns</b>	integer	(May be used only if <b>Predictor</b> is greater than 1) The number of samples in each row. Default value: 1.
<b>EarlyChange</b>	integer	( <b>LZWDecode</b> 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
<b>K</b>	integer	<p>A code identifying the encoding scheme used:</p> <p>&lt;0 Pure two-dimensional encoding (Group 4)</p> <p>=0 Pure one-dimensional encoding (Group 3, 1-D)</p> <p>&gt;0 Mixed one- and two-dimensional encoding (Group 3, 2-D), in which a line encoded one-dimensionally may be followed by at most <b>K</b> – 1 lines encoded two-dimensionally</p> <p>The filter shall distinguish among negative, zero, and positive values of <b>K</b> to determine how to interpret the encoded data; however, it shall not distinguish between different positive <b>K</b> values. Default value: 0.</p>

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

Key	Type	Value
<b>EndOfLine</b>	boolean	A flag indicating whether end-of-line bit patterns shall be present in the encoding. The <b>CCITTFaxDecode</b> filter shall always accept end-of-line bit patterns. If <b>EndOfLine</b> is <b>true</b> end-of-line bit patterns shall be present. Default value: <b>false</b> .
<b>EncodedByteAlign</b>	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 <b>true</b> , the filter shall skip over encoded bits to begin decoding each line at a byte boundary. If <b>false</b> , the filter shall not expect extra bits in the encoded representation. Default value: <b>false</b> .
<b>Columns</b>	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.
<b>Rows</b>	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.
<b>EndOfBlock</b>	boolean	A flag indicating whether the filter shall expect the encoded data to be terminated by an end-of-block pattern, overriding the <b>Rows</b> parameter. If <b>false</b> , the filter shall stop when it has decoded the number of lines indicated by <b>Rows</b> 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 <b>K</b> parameter. Default value: <b>true</b> .
<b>BlackIs1</b>	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: <b>false</b> .
<b>DamagedRowsBeforeError</b>	integer	The number of damaged rows of data that shall be tolerated before an error occurs. This entry shall apply only if <b>EndOfLine</b> is <b>true</b> and <b>K</b> is non-negative. Tolerating a damaged row shall mean locating its end in the encoded data by searching for an <b>EndOfLine</b> 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
<b>JBIG2Globals</b>	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.