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_4 c_5), using the relation

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

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 meth0d 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 variablelength 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, Elate-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 worstcase 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 ASC Il 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 added to table	Sequence represented by new code
-	256 (clear-table)	-	-
45	45	258	45 45

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

Codes 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 cyanmagenta-yellow-black) per sample, taking the difference between the values of corresponding components in adjacent samples, rather than between different colour components in the same sample, often reduces the output data size.

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

Table 8 — Optional parameters for LZWDecode and FlateDecode filters

Key	Type	Value
Predictor	integer	A code that selects the predictor algorithm, if any. If the value of this entry is 1, the filter shall assume that the normal algorithm was used to encode the data, without prediction. If the value is greater than 1, the filter shall assume that the data was differenced before being encoded, and **Predictor** selects the predictor algorithm. For more information regarding **Predictor** values greater than 1, see 7.4.4.4. "LZW and Flate Predictor Functions." Default value: 1
Colors	integer	*(May be used only if **Predictor** is greater than 1)* The number of interleaved colour components per sample. Valid values are 1 to 4 *(PDF 1.0)* and 1 or greater *(PDF 1.3).* Default value: 1.
BitsPerComponent	integer	*(May be used only if **Predictor** is greater than 1)* The number of bits used to represent each colour component in a sample. Valid values are 1, 2, 4, 8, and *(PDF 1.5)* 16. Default value: 8.
Columns	integer	*(May be used only if **Predictor** is greater than 1)* The number of samples in each row. Default value: 1.
EarlyChange	integer	*(**LZWDecode** only)* An indication of when to increase the code length. If the value of this entry is 0, code length increases shall be postponed as long as possible. If the value is 1, code length increases shall occur one code early. This parameter is included because LZW sample code distributed by some vendors increases the code length one code earlier than necessary. Default value: 1.

7.4.4.4 LZW and Flate Predictor Functions

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

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

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

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

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

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

Table 9 — PNG predictor algorithms

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

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

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

Table 10 — Predictor values

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

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

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

The predictor function groups also differ in significant ways:

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

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

7.4.5 RunLengthDecode Filter

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

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

7.4.6 CCITTFaxDecode Filter

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

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

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

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

- Unencoded data shall be treated as complete scan lines, with unused bits inserted at the end of each scan line to fill out the last byte. This approach is compatible with the PDF convention for sampled image data.

- Encoded data shall ordinarily be treated as a continuous, unbroken bit stream. The **EncodedByteAlign** parameter (described in Table 11) may be used to cause each encoded scan line to be filled to a byte boundary.

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

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

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

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

Table 11 — Optional parameters for the CCITTFaxDecode filter

Key	Type	Value
K	integer	A code identifying the encoding scheme used: <0 Pure two-dimensional encoding (Group 4) =0 Pure one-dimensional encoding (Group 3, 1 -D) >0 Mixed one- and two-dimensional encoding (Group 3, 2-0), in which a line encoded one-dimensionally may be followed by at most **K** — 1 lines encoded two-dimensionally The filter shall distinguish among negative, zero, and positive values of **K** to determine how to interpret the encoded data; however, it shall not distinguish between different positive **K** values. Default value: 0.
EndOLine	boolean	A flag indicating whether end-of-line bit patterns shall be present in the encoding. The **CCITTFaxDecode** filter shall always accept end-of-line bit patterns. If EndOfLine is **true** end-of-line bit patterns shall be present. Default value: **false**.
EncodedByteAlign	boolean	A flag indicating whether the filter shall expect extra 0 bits before each encoded line so that the line begins on a byte boundary. If **true**, the filter shall skip over encoded bits to begin decoding each line at a byte boundary. If **false**, the filter shall not expect extra bits in the encoded representation. Default value: **false**.
Columns	integer	The width of the image in pixels. If the value is not a multiple of 8, the filter shall adjust the width of the unencoded image to the next multiple of 8 so that each line starts on a byte boundary. Default value: 1728.

Key	Type	Value
Rows	integer	The height of the image in scan lines. If the value is 0 or absent. the image's height is not predetermined, and the encoded data shall be terminated by an end-of-block bit pattern or by the end of the filter's data. Default value: 0.
EndOfBlock	boolean	A flag indicating whether the filter shall expect the encoded data to be terminated by an end-of-block pattern, overriding the **Rows** parameter. If **false**, the filter shall stop when it has decoded the number of lines indicated by **Rows** or when its data has been exhausted, whichever occurs first. The end-ofblock pattern shall be the CCITT end-of-facsimile-block (EOFB) or return-to-control (RTC) appropriate for the **K** parameter. Default value: **true**.
Blackls1	boolean	A flag indicating whether 1 bits shall be interpreted as black pixels and 0 bits as white pixels, the reverse of the normal PDF convention for image data. Default value: **false**.
DamagedRowsBefore Error	integer	The number of damaged rows of data that shall be tolerated before an error occurs. This entry shall apply only if **EndOfLine** is **true** and **K** is non-negative. Tolerating a damaged row shall mean locating its end in the encoded data by searching for an **EndOfLine** pattern and then Substituting decoded data from the previous row if the previous row was not damaged, or a white scan line if the previous row was also damaged. Default value: 0.

NOTE 4 The compression achieved using CCITT encoding depends on the data, as well as on the value of various optional parameters. For Group 3 one-dimensional encoding, in the best Case (all zeros), each scan line compresses to 4 bytesб 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 JBIGI.

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 ration 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
JBIG2G1oba1s	stream	A stream containing the JBIG2 global (page 0) segments. Global segments shall be placed in this stream even if only a single JBIG2 image XObject refers to it.

EXAMPLE 1 The following shows an image that was compressed using the JBIG2 compression method and then encoded in ASCII hexadecimal representation. Since the JBIG2 bit stream contains global segments, these segments are placed in a separate PDF stream, as indicated by the JBIG2G10bals filter parameter.

```
5 0 obj
   << /Type /XObject
      /Subtype /Image
      /Width 52
      /Height 66
      /ColorSpace /DeviceGray
      /BitsPerComponent 1
      Length 224
      Filter [/ASCllHexDecode /JBIG2Decode]
     /DecodeParms [null << /JBIG2Globals 6 0 R>>]
```

```
    >>
stream
00000001300001000000130000003400000042000000000
00000040000000000002062000010000001e000000340000
00420000000000000000020010000000231db51ce51ffac>
endstream
endobj

6 0 obj
    << /Length 126
        Filter /ASCllHexDecode
    >>
 stream
00000000001000000003200000 3fffdff02fefefe000000
01000000012ae225aea9a5a538b4d9999c5c8e56ef0f872
7f2b53d4e37ef795cc5506dffac>
endstream
endobj
```

The JBIG2 bit stream for this example is as follows:

EXAMPLE 2 97 4A 42 32 OD OA IA OA 01 00 00 00 01 00 00 00 00 00 01 00 00 00 00 32 00 00 03 FF FD FF 02 FE FE FE 00 00 00 01 00 00 00 01 2A E2 25 AE A9 A5 A5 38 B4 09 99 9C 5C BE 56 EF OF 87 27 F2 B5 3D 4E 37 EF 79 5C (35 50 6D FF AC 00 00 00 01 30 00 01 00 00 00 13 00 00 00 34 00 00 00 42 00 00 00 00 00 00 00 00 00 40 00 00 00 00 00 02 06 20 00 01 00 00 00 IE 00 00 00 34 00 00 00 42 00 00 00 00 00 00 00 00 02 00 10 00 00 00 02 31 DB 51 CE 51 FF AC 00 00 00 03 31 00 01 00 00 00 00 00 00 00 04 33 01 00 00 00 00

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

a) The JBIG2 file header

97 4A 42 32 OD OA IA OA 01 00 00 00 01

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

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

00 00 00 00 01 00 00 00 00 32 00 00 03 FF FD FF 02 FE FE FE 00 00 00

00 00 00 01 2A E2 25 AE A9 A5 A5 38 84 09 99 9C 5C 8E 56 EF OF 87

27 F2 B5 3D 4E 37 EF 79 5C C5 50 60 FF AC

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

c) The page information segment

00 00 00 01 30 00 01 00 00 00 13 00 00 00 34 00 00 00 42 00 00 00 00

00 00 00 00 40 00 00

and the immediate text region segment

00 00 00 02 06 20 00 01 00 00 00 1E 00 00 00 34 00 00 00 42 00 00 00

00 00 00 00 00 02 00 10 00 00 00 02 31 DB 51 CE 51 FF AC

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

d) The end-of-page segment

00 00 00 03 31 00 01 00 00 00 00

and the end-of-file segment

00 00 00 04 33 01 00 00 00 00

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

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

7.4.8 DCTDecode Filter

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

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

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

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

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

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

Table 13 — Optional parameter for the DCTDecode filter

Key	Type	Value
ColorTransform	integer	(*Optional*) A code specifying the transformation that shall be performed on the sample values: 0 No transformation. 1 If the image has three colour components, *RGB* values shall be transformed to *YUV* before encoding and from *YUV* to *RGB* after decoding. If the image has four components, *CMYK* values shall be transformed to *YUVK* before encoding and from *YUVK* to *CMYK* after decoding. This option shall be ignored if the image has one or two colour components. If the encoding algorithm has inserted the Adobe-defined marker[a] code in the encoded data indicating the ColorTransform value, then the colours shall be transformed, or not, after the DCT decoding has been performed according to the value provided in the encoded data and the value of this dictionary entry shall be ignored. If the Adobedefined marker code in the encoded data indicating the ColorTransform value is not present then the value specified in this dictionary entry will be used. If the Adobe-defined marker code in the encoded data indicating the ColorTransform value is not present

Key	Type	Value
		and this dictionary entry is not present in the filter dictionary then the default value of ColorTransform shall be 1 if the image has three components and 0 otherwise.

[a] Parameters that control the decoding process as well as other is embedded within the encoded data stream using a notation referred to as "markers". When it defined the use of JPEG images within PostScript data streams. Adobe System Incorporated defined a particular set of rules pertaining to which markers are to be recognized, which are to be ignored and which are considered errors. A specific Adobe-defined marker was also The exact rules for producing and consuming OCT encoded data within PostScript are provide in Adobe Technical Note #5116 (reference). PDF DCT Encoding shall exactly follow those rules established by Adobe for PostScript

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

NOTE 4 Better compression is often possible for colour spaces that treat luminance and chrominance separately than for those that do not. The RGB-to-YUV conversion provided by the filters is one attempt to separate luminance and chrominance; it conforms to CCIR recommendation 601-1. Other colour spaces, such as the CIE 1976 L*a*b8*space, may also achieve this objective. The chrominance components can then be compressed more than the luminance by using coarser sampling or quantization, with no degradation in quality.

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

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

7.4.9 JPXDecode Filter

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

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

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

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

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

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

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

NOTE 5 To promote interoperability. the specifications define a subset of JPX called JPX baseline (of which JP2 is also a subset). The complete details of the baseline set of JPX features are contained in ISO/IEC 15444-2, Information Technology—JPEG 2000 Image Coding System: Extensions (see the Bibliography). See also <http:/twww.jpeg.org/Jpeg2000/>.

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

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

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

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

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

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

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

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

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

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

There are other features in the JPX format beyond describing a simple image. These include provisions for describing layering and giving instructions on composition, specifying simple animation, and including generic XML metadata (along with JPEG2000-specific schemas for such data). Relevant metadata should be replicated in the image dictionary's **Metadata** stream in XMP format (see 14.32, "Metadata Streams").

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

- **Width** and **Height** shall match the corresponding width and height values in the JPEG2000 data.

- **ColorSpace** shall be optional since JPEG2000 data contain colour space specifications. If present. it shall determine how the image samples are interpreted, and the colour space specifications in the JPEG2000 data shall be ignored. The number of colour channels in the JPEG2000 data shall match the number of components in the colour space; a conforming writer shall ensure that the samples are consistent with the colour space used.

- Any colour space other than **Pattern** may be specified. If an **Indexed** colour space is used, it shall be subject to the PDF limit of 256 colours. If the colour space does not match one of JPX's enumerated colour spaces (for example, if it has two colour components or more than four), it should be specified as a vendor colour space in the JPX data.

- If **ColorSpace** is not present in the image dictionary, the colour space information in the JPEG2000 data shall be used. A JPEG2000 image within a PDF shall have one of: the baseline JPX colorspaces; or enumerated colorspace 19 (CIEJab) or enumerated colorspace 12 (CMYK); or at least one ICC profile that is valid within PDF. Conforming PDF readers shall support the JPX baseline set of enumerated colour spaces: they shall also be responsible for dealing with the interaction between the colour spaces and the bit depth of samples.

- If multiple colour space specifications are given in the JPEG2000 data, a conforming reader should attempt to use the one with the highest precedence and best approximation value. If the colour space is given by an unsupported ICC profile, the next lower colour space, in terms of precedence and approximation value, shall be used. If no supported colour space is found, the colour space used shall be DeviceGray, DeviceRGB, or DeviceCMYK, depending on the whether the number of channels in the JPEG2000 data is 1, 3 or 4.

- **SMasklnData** specifies whether soft-mask information packaged with the image samples shall be used (see 11.6.5.3, "Soft-Mask Images"); if it is, the **SMask** entry shall not be present. If **SMasklnData** is nonzero, there shall be only one opacity channel in the JPEG2000 data and it shall apply to all colour channels.

- **Decode** shall be ignored, except in the case where the image is treated as a mask: that is, when **ImageMask** is **true**. In this case, the JPEG2000 data shall provide a single colour channel with 1-bit samples.

7.4.10 Crypt Filter

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

Table 14 — Optional parameters for Crypt filters

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

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

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

If a stream specifies a crypt filter, then the security handler does not apply "Algorithm 1: Encryption of data using the RC4 or AES algorithms" in 7.62, "General Encryption Algorithm," to the key prior to decrypting the stream. Instead, the security handler shall decrypt the stream using the key as is. Sub-clause 7.4, "Filters," explains how a stream specifies filters.

7.5 File Structure

7.5.1 General

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

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

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

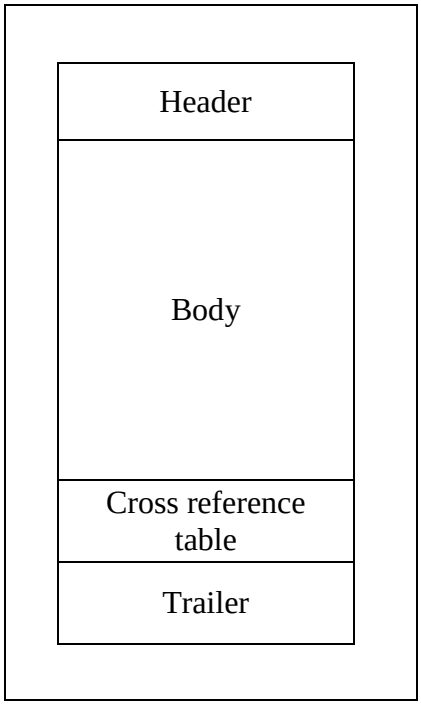

Figure 2 — Initial structure of a PDF file

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

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

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

7.5.2 File Header

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

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

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

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

NOTE This allows a conforming writer to update the version using an incremental update (see 7.56, "Incremental Updates").

Under some conditions, a conforming reader may be able to process PDF files conforming to a later version than it was designed to accept. New PDF features are often introduced in such a way that they can safely be ignored by a conforming reader that does not understand them (see I.2, "PDF Version Numbers").

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

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

7.5.3 File Body

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

7.5.4 Cross-Reference Table

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

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

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

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

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

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

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

28 5

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

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

objects that have been deleted and therefore are free. Both types of entries have similar basic formats, distinguished by the keyword **n** (for an in-use entry) or **f** (for a free entry). The format of an in-use entry shall be:

*nnnnnnnnn ggggg **n** eol*

where:

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

ggggg shall be a 5-digit generation number

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

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

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

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

nnnnnnnnn ggggg **f** *eol*

where:

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

ggggg shall be a 5-digit generation number

f shall be a keyword identifying this as a free entry

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

There are two ways an entry may be a member of the free entries list. Using the basic mechanism the free entries in the cross-reference table may form a linked list, with each free entry containing the object number of the next. The first entry in the table (object number 0) shall always be free and shall have a generation

number of 65,535; it is shall be the head of the linked list of free objects. The last free entry (the tail of the linked list) links back to object number 0. Using the second mechanism, the table may contain other free entries that link back to object number 0 and have a generation number of 65,535, even though these entries are not in the linked list itself.

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

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

EXAMPLE 2 The following shows a cross-reference section consisting of a single subsection with six entries: four that are in use (objects number 1.2, 4, and 5) and two that are free (objects number 0 and 3). Object number 3 has been deleted, and the next object created with that object number is given a generation number of 7.

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

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

xref
0 1

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

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

7.5.5 File Trailer

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

trailer
<< key_1 $value_1$,
key_2 $value_2$
…
key_n $value_n$
>>
startxref
Byte_offset_of_last_cross-reference_section
%%EOF

Table 15 lists the contents of the trailer dictionary.

Table 15 — Entries in the file trailer dictionary

Key	Type	Value
Size	integer	*(Required; shall not be an indirect reference)* The total number of entries in the file's cross-reference table, as defined by the combination of the original section and all update sections. Equivalently, this value shall be 1 greater than the highest object number defined in the file. Any object in a cross-reference section whose number is greater than this value shall be ignored and defined to be missing by a conforming reader.
Prev	integer	*(Present only if the file has more than one cross-reference section; shall be an indirect reference)* The byte offset in the decoded stream from the beginning of the file to the beginning of the previous cross-reference section.
Root	dictionary	*(Required; shall be an indirect reference)* The catalog dictionary for the PDF document contained in the file (see 7.7.2, "Document Catalog").
Encrypt	dictionary	*(Required if document is encrypted; PDF 1.1*) The document's encryption dictionary (see 7.6, "Encryption").
Info	dictionary	(*Optional; shall be an indirect reference)* The document's information dictionary (see 14.33, "Document Information Dictionary").
ID	array	*(Required if an **Encrypt** entry is present; optional otherwise; PDF 1.1)* An if array of two byte-strings constituting a file identifier (see 14.4, "File Identifiers") for the file. If there is an **Encrypt** entry this array and the two shall be direct objects and shall be unencrypted. NOTE 1 Because the **ID** entries are not encrypted it is possible to check the **ID** key to assure that the correct file is being accessed without decrypting the file. The restrictions that the string be a direct object and not be encrypted assure that this is possible. NOTE 2 Although this entry is optional. its absence might prevent the file from functioning in some workflows that depend on files being uniquely identified. NOTE 3The values of the **ID** strings are used as input to the encryption algorithm. If these strings were indirect, or if the **ID** array were indirect, these strings would be encrypted when written. This would result in a circular condition for a reader: the **ID** strings must be decrypted in order to use them to decrypt strings, including the **ID** strings themselves. The preceding restriction prevents this circular condition.

NOTE Table 19 defines an additional entry, **XRefStm**, that appears only in the trailer of hybrid-reference files, described in 7.584, "Compatibility with Applications That Do Not Support Compressed Reference Streams."

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

```
<< /Size 22
    /Root 2 0 R
    /Info 1 0 R
    /ID[ <81b14aafa313db63db6f981e49f94f4>
         <81b14aafa313db63db6f981e49f94f4>
       ]
>>
startxref
18799
%%EOF
```

7.5.6 Incremental Updates

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

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

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

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

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

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

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

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

In versions of PDF 1.4 or later a conforming writer may use the **Version** entry in the document's catalog dictionary (see 7.72 "Document Catalog") to override the version specified in the header. A conforming writer may also need to update the Extensions dictionary, see 7.12, "Extensions Dictionary", if the update either deleted or added developer-defined extensions.

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

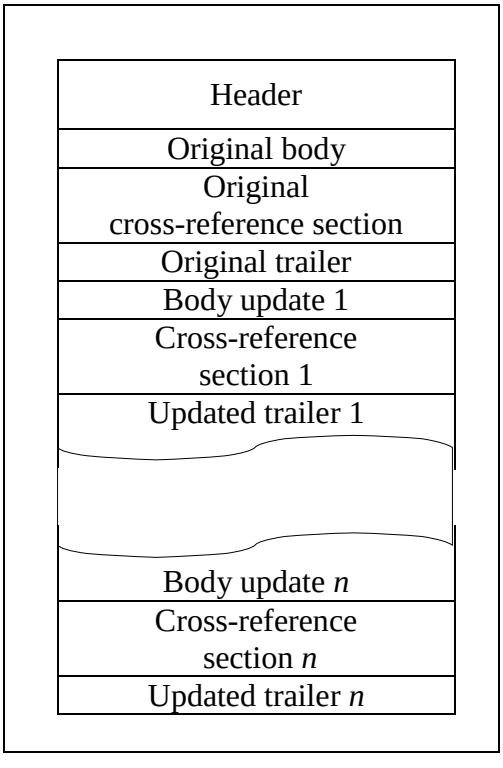

Figure 3 — Structure of an updated PDF file

7.5.7 Object Streams

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

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

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

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

- Stream objects
- Objects with a generation number other than zero
- A document's encryption dictionary (see 7.6, "Encryption")
- An object representing the value of the **Length** entry in an object stream dictionary
- In linearized files (see Annex F), the document catalog, the linearization dictionary, and page objects shall not appear in an object stream.

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

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

Table 16 — Additional entries specific to an object stream dictionary

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

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

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

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

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

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

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

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

- The value of the **First** entry in the stream dictionary shall be the byte offset in the decoded stream of the first object.

- The **N** objects are stored consecutively. Only the object values are stored in the stream: the **obj** and **endobj** keywords shall not be used.

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

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

EXAMPLE 2 3 0 R

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

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

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

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

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

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

12 0 obj
   <</Type/IFontDescriptor
      /Ascent 891
      ...other entries...
      /FontFi1e2 22 0 R
   >>
endobj

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

xref
0 32
0000000000 65535 f
...cross-reference entries for objects 1 through 10...
0000001434 00000 n
0000001735 00000 n
0000002155 00000 n
```

```
…cross-reference entries for objects 14 and on…
trailer
   << /Size 32
      /Root…
   >>
```

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

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

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

```
150 obj                 % The object stream
  <</Type /ObjStm
     /Length 1856
     /N 3               % The number of objects in the stream
     /First 24          % The byte offset in the decoded stream of the first object
% The object numbers and offsets of the objects relative to the first are shown on the first line of
% the stream (i.e., 11 0 12 547 13665).
  >>
stream
  11 0 12547 13665
  << /Type Font
     /Subtype/TrueType
     ...other keys...
     FontDescnptor 12 0 R
  >>
```

```
  << /Type/FontDescriptor
     /Ascent 891
     ...other keys...
     /FontFile2 22 0 R
  >>
  << /Type Font
     /Subtype/Type0
     ...other keys...
```

```
        /ToUnicode 10 0 R
      >>
    …
    endstream
    endobj

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

startxref
   54321
%%EOF
```

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

7.5.8 Cross-Reference Streams

7.5.8.1 General

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

- A more compact representation of cross-reference information
- The ability to access compressed objects that are stored in object streams (see 7.57, "Object Streams") and to allow new cross-reference entry types to be added in the future

Cross-reference streams are stream objects (see 7.3.8, "Stream Objects"), and contain a dictionary and a data stream. Each cross-reference stream contains the information equivalent to the cross-reference table (see 7.5.4, "Cross-Reference Table") and trailer (see 7.5-5, "File Trailer") for one cross-reference section.

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

```
…objects…

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

… more objects…

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

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

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

7.5.8.2 Cross-Reference Stream Dictionary

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

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

- Other cross-reference stream entries not listed in Table 17 may be indirect: in fact, some (such as **Root** in Table 15) shall be indirect.

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

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

key	type	description
Type	name	*(Required)* The type of PDF object that this dictionary describes; shall be **XRef** for a cross-reference stream.
Size	integer	*(Required)* The number one greater than the highest object number used in this section or in any section for which this shall be an update. It shall be equivalent to the **Size** entry in a trailer dictionary.
Index	array	*(Optional)* An array containing a pair of integers for each subsection in this section. The first integer shall be the first object number in the subsection; the second integer shall be the number of entries in the subsection The array shall be sorted in ascending order by object number. Subsections cannot overlap: an object number may have at most one entry in a section. Default value: [0 **Size**]
Prev	integer	*(Present only if the file has more than one cross-reference stream: not meamngful in hybrid-reference files: see 7.5.8.4. "Compatibility With Applications That Do Not Support Compressed Reference Streams")* The byte offset in the decoded stream from the beginning of the file to the beginning of the previous cross-reference stream. This entry has the same function as the **Prev** entry in the trailer dictionary (Table 15).

key	type	description
W		*(Required)* An array of integers representing the size of the fields in a single cross-reference entry. Table 18 describes the types of entries and their fields. For PDF 1.5. **W** always contains three integers; the value of each integer shall be the number of bytes (in the decoded stream) of the corresponding field. EXAMPLE [1 2 1] means that the fields are one byte, two bytes, and one byte, respectively. A value of zero for an element in the **W** array indicates that the corresponding field shall not be present in the stream. and the default value shall be used. if there is one. If the first element is zero, the type field shall not be present. and shall default to type 1. The sum of the items shall be the total length of each entry; it can be used with the **Index** array to determine the starting position of each subsection. Different cross-reference streams in a PDF file may use different values for **W**

7.5.8.3 Cross-Reference Stream Data

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

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

Table 18 — Entries in a cross-reference stream

Type	Field	Description
0	1	The type of this entry, which shall be 0. Type 0 entries define the linked list of free objects (corresponding to **f** entries in a cross-reference table).
	2	The object number of the next free object.
	3	The generation number to use if this object number is used again.
1	1	The type of this entry, which shall be 1. Type 1 entries define objects that are in use but are not compressed (corresponding to n entries in a cross-reference table).

Type	Field	Description
	2	The byte offset of the object, starting from the beginning of the file.
	3	The generation number of the object. Default value: 0.
2	1	The type of this entry, which shall be 2. Type 2 entries define compressed objects.
	2	The object number of the object stream in which this object is stored. (The generation number of the object stream shall be implicitly 0.)
	3	The index of this object within the object stream.

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

7.5.8.4 Compatibility with Applications That Do Not Support Compressed Reference Streams

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

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

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

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

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

The **Size** entry of the trailer shall be large enough to include all objects, including those defined in the crossreference stream referenced by the **XRefStm** entry. However, to allow random access, a main cross-reference section shall contain entries for all objects numbered 0 through **Size** - 1 (see 7.54. "Cross-Reference Table"). Therefore, the **XRefStm** entry shall not be used in the trailer dictionary of the main cross-reference section but only in an update cross-reference section.

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

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

There are limitations on which objects in a hybrid-reference file can be hidden without making the file appear invalid to readers designed only to support versions of PDF before PDF 1.5. In particular, the root of the PDF file and the document catalog (see 7.72 "Document Catalog") shall not be hidden, nor any object that is *visible from the root*. Such objects can be determined by starting from the root and working recursively:

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

In general, the objects that may be hidden are optional objects specified by indirect references. A conforming reader can resolve those references by processing the cross-reference streams. In a reader designed only to support

versions of PDF before PDF 1.5, the objects appear to be free, and the references shall be treated as references to the null object.

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

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

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

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

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

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

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

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

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

endobj

12 0 obj
…
endobj
99 0 obj
…
endobj

%The main xref section. at offset 2664 is next with entries for objects 0-99.
%Objects 2 through 11 are marked free and objects 12, 13 and 99 are marked in use.
xref
0 100
0000000002 65535 f
0000000023 00000 n
0000000003 65535 f
0000000004 65535 f
0000000005 65535 f
0000000006 65535 f
0000000007 65535 f
0000000008 65535 f
0000000009 65535 f
0000000010 65535 f
0000000011 65535 f
0000000000 65535 f
0000000045 00000 n
0000000179 00000 n
…cross-reference entries for objects 14 through 98…
0000002201 00000 n
trailer
<< /Size 100
/Root 1 0 R
/ID …
>>
%The main xref section starts at offset 2664.
startxref
2664
%%EOF

2 0 obj %The object stream, at offset 3722
<< /Length …
/N 8 % This stream contains 8 objects.
/First 47 % The stream-offset of the first object
>>
stream
3 0 4 50 5 72… the numbers and stream-offsets of the remaining 5 objects followed by dictionary objects 3-5 …

```
            << /Type/StructTreeRoot
               /K 4 0 R
               /RoleMap 5 0 R
               /ClassMap 6 0 R
               /ParentTree 7 0 R
               /ParentTreeNextKey 8
            >>
            << /S/Workbook
               /P 8 0 R
               /K 9 0 R
            >>
            << /Workbook/Div
               /Worksheet/Sect
               /TextBox/Figure
               /Shape/Figure
            >>
            …definitions for objects 6 through 10…
         endsteam
         endobj

         11 00b                          %The cross-reference stream. at offset 4899
            << /Type /XRef
               /Index [2 10]             %This stream contains entries for objects
                                         2 through 11
               /Size 100
               /W [1 2 1]                %The byte-widths of each field
               /Filter/ASCIIHexDecode    %For readability only
               …
            >>>
         stream
            01 0E8A 0
            02 0002 00
            02 0002 01
            02 0002 02
            02 0002 03
            02 0002 04
            02 0002 05
            02 0002 06
            02 0002 07
            01 1323 0
         endstream
         endobj
%The entries above are for: object 2 (0x0E8A = 3722), object 3 (in object stream 2. index 0),
%object 4 (in object stream 2, index 1)… object 10 (in object stream 2, index 7),
%object 11 (0x1323 = 4899).
```

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

The previous example illustrates several other points:

- The object stream is unencoded and the cross-reference stream uses an ASCII hexadecimal encoding for clarity. In practice, both streams should be Flate-encoded. PDF comments shall not be included in a crossreference table or in cross-reference streams.

- The hidden objects. 2 through 11, are numbered consecutively. In practice, hidden objects and other free items in a cross-reference table need not be linked in ascending order until the end.