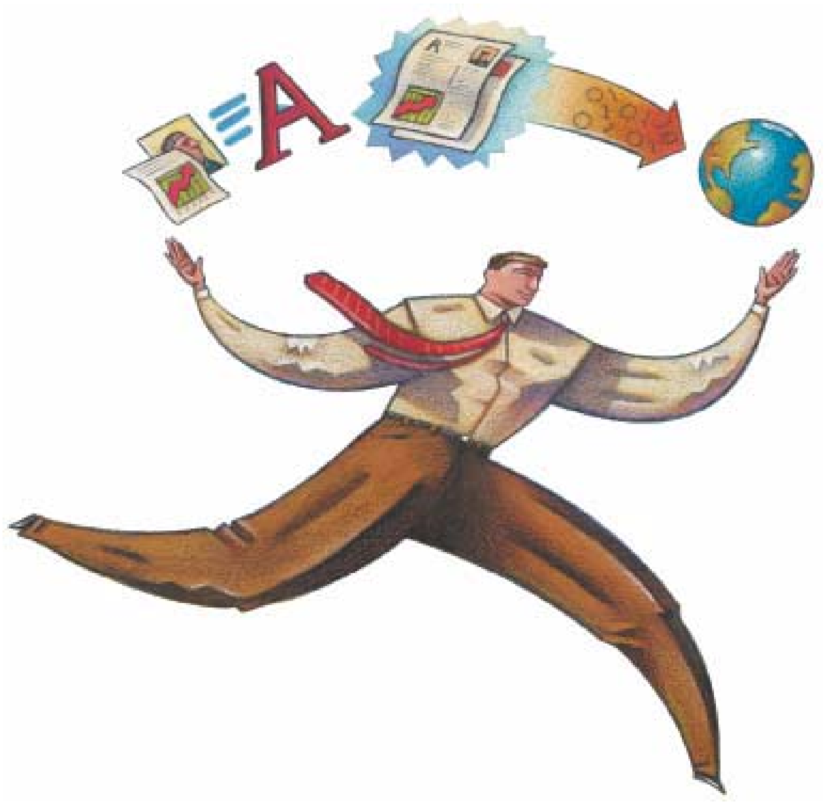
Portable Document Format: Changes from Version1.3 to1.4

Technical Note #5409

Preliminary



ADOBE SYSTEMS INCORPORATED

Corporate Headquarters

345 Park Avenue

San Jose, CA 95110-2704

(408) 536-6000

http://partners.adobe.com

June 11, 2001

Copyright 2000, 2001 Adobe Systems Incorporated. All rights reserved.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the Adobe Systems Incorporated.

PostScript is a registered trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Except as otherwise stated, any reference to a "PostScript printing device," "PostScript display device," or similar item refers to a printing device, display device or item (respectively) that contains PostScript technology created or licensed by Adobe Systems Incorporated and not to devices or items that purport to be merely compatible with the PostScript language.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Acrobat Capture, Acrobat Catalog, Acrobat Exchange, Acrobat Search, Distiller, Illustrator, Photoshop, PostScript, and the PostScript logo are trademarks of Adobe Systems Incorporated.

Apple, Macintosh, and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. PowerPC is a registered trademark of IBM Corporation in the United States. ActiveX, Microsoft, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third party rights.

Contents

Introduction

This document is a preliminary draft of the
language extensions for PDF version 1.4.

The information in this document is subject to change; it will become final when the *PDF Reference, Third Edition, Version 1.4* is published, which will be after the Adobe Acrobat 5.0 product ships in early 2001.

The information in this document is organized by topic, in the approximate order of the current PDF Reference (PDFR), Second Edition. Most sections contain a reference (in square brackets) to the chapter or section to which the new text applies.

The second release of this document, dated May 22, 2001, contained only a few updates to the Encryption and Accessibility chapters.

This second update, dated June 11, 2001, adds a few additional changes to the chapter on Encryption.

JBIG2 Compression Filter

[Applies to Section 3.3, “Details of Filtered Streams”]

1 Introduction

PDF 1.4 adds the ability to compress single-bit images using the JBIG2 image compression filter. For single-bit images, JBIG2 compression provides considerably better compression than the existing CCITT standard. In lossy mode, compression ratios can be as high as three to seven times that achieved by CCITT G4 for single-page images, with higher ratios possible for multi-page documents.

JBIG2 is currently approaching standards approval. Current information, and the latest version of the working draft, are available from:

http://www.jpeg.org/public/jbigpt2.htm

The JBIG2 standard defines a compliant bitstream, but not an explicit encoder design. Encoder capability and performance are differentiators for competitive implementations.

JBIG2 is based on symbol matching, and works well for images of text pages in any language. The encoder constructs a table of symbols found in the image (called a “symbol dictionary”), and attempts to match symbols found later in the document with the existing symbol dictionary. If no acceptable match is found, a new symbol is added to the symbol dictionary. If an acceptable match is found, JBIG2 has a lossy mode, where the occurrence of the symbol is replaced with an index into the symbol dictionary, and a lossless mode, where the difference between the dictionary symbol and the matched symbol is also stored.

A similar dictionary mechanism is used for dithered halftone regions of the image, where a halftone dictionary is constructed, and used to encode the halftone region of the image. Regions of the image which are neither halftone nor symbols (called *generic* regions – typically drawings or illustrations) are encoded using arithmetic or Huffman coding.

The JBIG2 bitstream includes a header, and segments containing either a dictionary (symbol or halftone), or an encoded region.

Note: *The use of the terms "halftone" and "halftone dictionary" in this section are unrelated to the Halftone resources defined in Section 6.4 on page 356 in PDF Reference, Second Edition.*

1.1 Multi-page images

JBIG2 supports compression of multiple page-images as a single JBIG2 stream. If the pages belong to a single document, they will tend to use only a few typefaces, and the number of matches found on later pages, and hence the compression ratios achieved, are significantly higher than for single-page images.

To allow PDF representation of multi-page JBIG2-encoded images, the JBIG2 bitstream is split into page-specific parts and an off-page part. The page-specific parts will form the content of Image XObjects referred to from individual page descriptions. These XObjects will contain JBIG2 segments which are page-specific, and can optionally refer to a PDF stream object which will contain global JBIG2 segments, typically symbol and halftone dictionaries.

2 PDF Reference Manual changes

The following additions will be made to the PDF Reference, Second Edition.

2.1 Changes to Section 3.3, ' "Filtered Streams"

Note: *[The following entry will be added to Table 3.5, "Standard Filters"]*

TABLE 1 Addition to Table 3.5, Standard Filters

KEY	TYPE	VALUE
JBIG2Decode	dictionary	*(Optional)* Decompresses single-bit sampled image data using a symbol-based compression algorithm, the JBIG2 standard,

Note: *[The following section will be added as Section 3.3.7, "JBIG2Decode"]:*

This filter decodes single-bit image data encoded using the JBIG2 (Joint Bi-level Image experts Group) standard. Encoding an image using JBIG2 always results in binary data.

JBIG2 provides for both lossy and lossless compression of single-bit images. It is useful only for single-bit images, not for color images, grayscale images, or text. The compression achieved using JBIG2 depends strongly on the nature of the image. Images of pages containing text in any language will compress particularly well, with typical compression ratios of 20:1 to 50:1 for a page full of text. The JBIG2 encoder builds a table of unique symbol bitmaps found in the image, and other symbols found later in the image are matched against the table. Matching symbols are replaced by an index into the table, and symbols which fail to match are added to the table. The table itself is compressed using other means. This results in high compression ratios for documents where the same symbol repeats often, as is typical for images created by scanning text pages. This method also results in high compression of whitespace in the image, as whitespace regions will contain no symbols, and will not need to be encoded.

While best compression is achieved for images of text, the JBIG2 standard also includes algorithms for compressing regions of the image which contain dithered halftone images, or figures.

The algorithms used by the encoder, and the details of the format, are not described here, but can be obtained from the JBIG2 committee homepage at:

http://www.jpeg.org/public/jbighomepage.htm

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

The filter addresses both single-page and multi-page JBIG2 bitstreams, by representing each JBIG2 “page” as a PDF image. This is achieved by using the following mechanism:

- The filter uses the Embedded file organization of JBIG2 (see appendix G of the JBIG2 standard). The optional 2-byte combination (marker) described in the description of the organization is not used in PDF. JBIG2 bitstreams in random-access organization should be converted to this organization. Bitstreams in Sequential organization need no reorganization, except for the mappings described below.
- The JBIG2 file header, End of Page segments, and End of File segment are not stored in PDF. These should be removed before creating the PDF objects described below.
- The Image XObject to which the JBIG2Decode filter is applied contains all segments which are associated with the JBIG2 “page” represented by that image i.e., segments whose “segment page association” field contains the page number of the JBIG2 “page” represented by the image. In the XObject, however, the segment’s page number should always be 1, that is. when these segments are written to the XObject, the value of their “segment page association” field should always be set to 1.
- If the bitstream contains global segments (segments whose “segment page association” field contains zero), these should be placed in a separate PDF stream, and the DecodeParms for the **JBIG2Decode** filter for the image should contain a **JBIG2Globals** entry, whose value is a stream object containing the JBIG2 global segments.

Note: *[The following will be inserted as* **“***Table 3.11 Parameters for JBIG2Decode”]:*

TABLE 3.11 Key Type Semantics

KEY	TYPE	VALUE
JBIG2Globals	stream	*(Optional).* Stream containing the JBIG2 global (page 0) segments.

Note: *[The following will be inserted as “Example 3.4 JBIG2-encoded image”]:*

```
5 0 obj
<<
/Type /XObject
/Subtype /Image
/Width 24
/Height 23
/BitsPerComponent 1
/ColorSpace /DeviceGray
/Filter [/ASCII85Decode /JBIG2Decode]
/DecodeParms [null <</JBIG2Globals 6 0 R>>]
/Length 113
>>
stream
J..)6T`?p&<!J9%_[umg"B7/Z7KNXbN'S+,*Q/&"OLT'F
LIDK#!n`$"<Atdi`\Vn%b%)&'cA*VnK\CJY(sF>c!Jnl@
RM]WM;jjH6Gnc75idkL5]3k
endstream
endobj
6 0 obj
<<
/Filter /ASCII85Decode
/Length 1587
>>
stream
,p>`rDKJj'E+LaU0eP.@+AH9dBOu$hFD55nC
...omitted data...
endstream
endobj
```

As shown in the above example, the value for **BitsPerComponent** must be 1, and the value for **ColorSpace** must be **DeviceGray**.

3 Inline images

Since PDF inline image object format is useful for small images, and most JBIG2-encoded images will be of relatively large scanned pages, usage of the **JBIG2Decode** filter is not allowed for inline images.

Encryption

[Applies to Section 3.5, “Encryption”]

Note: *This chapter is a complete replacement for Section 3.5, Encryption, in the PDF Reference, Second Edition. References in this section – to tables and page numbers that are not in this chapter – are references to the PDF Reference, Second Edition.*

3.5 Encryption

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

Note: *When a PDF stream object (see Section 3.2.7, “Stream Objects”) refers to an external file, the stream’s contents are not encrypted, since they are not part of the PDF file itself. However, if the contents of the stream are embedded within the PDF file (see Section 3.10.3, “Embedded Stream Objects”), they are encrypted like any other stream in the file.*

Encryption is controlled by an *encryption dictionary*, which is the value of the **Encrypt** entry in the document’s trailer dictionary (see Table 3.11 on page 61 of the PDF Reference). If this entry is absent from the trailer dictionary, the document is not encrypted. The entries shown in Table 3.12 are common to all encryption dictionaries.

TABLE 3.12 Entries common to all encryption dictionaries

KEY	TYPE	SEMANTICS
Filter	name	*(Required)* The name of the security handler for this document; see below. Default value: **Standard**, for the built-in security handler. (Names for other security handlers can be registered using the procedure described in Appendix E.).
V	number	(*Optional, value defaults to 0 if not present*) A code specifying the algorithm to be used in encrypting and decrypting the document: Values indicating decryption algorithm 0 An alternate algorithm that is undocumented and no longer supported, and whose use is strongly discouraged. 1 Algorithm 3.1 (see below). 2 [PDF 1.4] Algorithm 3.1 (see below), but allowing key lengths greater than 40 bits. 3 [PDF 1.4] An unpublished algorithm allowing key lengths up to 128 bits. This algorithm is unpublished as an export requirement of the U.S. Department of Commerce. The default value of this entry if it is omitted is 0, but a value of 1 or greater is strongly recommended.
Length	integer	*(Optional, defaults to 40)* Defined only when **V** is 2 or 3. Length of key, in bits, used for encryption and decryption. The size must be a multiple of 8, with a minimum value of 40 and maximum value of 128.

The encryption dictionary's **Filter** entry identifies the file's *security handler*, a software module that implements various aspects of the encryption process and controls access to the contents of the encrypted document. PDF specifies a standard security handler that all viewer applications are expected to support, but applications may optionally substitute alternate security handlers of their own. The remaining contents of the encryption dictionary are determined by the security handler, and may vary from one handler to another. Those for the standard security handler are described below in Section 3.5.2, "Standard Security Handler."

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

Note: *If the standard encryption methods provided by PDF are not sufficient to their needs, document creators have two choices: they can provide an alternate, more secure security handler or they can encrypt whole PDF documents themselves, bypassing PDF security entirely.*

3.5.1 General Encryption Algorithm

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

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

The encryption of data in a PDF file is based on the use of an *encryption key* computed by the security handler. Different security handlers can compute the key in a variety of ways, more or less cryptographically secure. The key length can vary dependent on the algorithm version that is specified by the **V** entry in the encryption dictionary. The algorithm version is specified by the security handler that is used to encrypt the document. The PDF 1.3 specification defined algorithms up to version 1, allowing 40-bit key lengths. The PDF 1.4 specification defines up to version 2 which allows variable-length keys. PDF's **Standard** encryption handler can use algorithm versions 1 and 2.

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

1. Obtain the object number and generation number from the object identifier of the string or stream to be encrypted (see Section 3.2.9, "Indirect Ob-

jects”). If the string is a direct object, use the identifier of the indirect object containing it.

2. Treating the object number and generation number as binary integers, extend the original N-byte key to N+5 bytes by appending the low-order 3 bytes of the object number and the low-order 2 bytes of the generation number in that order, low-order byte first. N is the key length in bytes, which is always 5 (40 bits) for algorithm version 1, but may be as large as 16 (128 bits) for other algorithm versions.
3. Initialize the MD5 hash function and pass the result of step 2 as input to the MD5 hash function.
4. Use the first N+5 bytes of the output from the MD5 function as the key for the RC4 encryption function, along with the string or stream data to be encrypted. If N+5 is greater than 16 then only the first 16 bytes are used for the RC4 key. The output is the encrypted data to be stored in the PDF file.

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

3.5.2 Standard Security Handler

PDF’s **Standard** security handler allows two passwords to be specified for a document: an *owner password* and a *user password*. Correctly supplying either password allows a user to open the document, decrypt it, and display it on the screen. Correctly supplying the owner password will give full access to a document. Correctly supplying the user password will potentially provide a reduced set of operations that can be performed on the document. Access information in the document’s encryption dictionary specifies which of these operations, if any, may be performed. The owner password must be supplied in order to change these restrictions or the passwords themselves. The following operations are optionally permitted for **Standard** security handler 2 when a correct user password is provided:

- Modifying the document’s contents
- Copying text and graphics from the document
- Adding or modifying text annotations (see Section 7.4 “Annotations”) and interactive form fields (Section 7.6 “Interactive Forms”)
- Printing the document

The following additional operations are optionally permitted for **Standard** security handler 3 when a correct user password is provided:

- Form fill-in and sign document
- Text inspection for accessibility
- Document assembly, including insertion, rotation, and deletion of pages and creation of bookmarks and thumbnails
- Allow only printing that does not allow perfect digital copies, but which may also result in degradation of output quality

Note: *PDF cannot enforce the document access privileges specified in the encryption dictionary. It is up to the implementors of PDF viewer applications to respect the intent of the document creator by restricting access to an encrypted PDF file according to the passwords and permissions contained in the file.*

Note: *If the owner and user passwords are the same, the document is always opened with user access privileges. It is therefore impossible in these circumstances to obtain owner privileges for the document.*

Encryption Dictionary

Table 3.13 shows the encryption dictionary entries for the standard security handler (in addition to those in Table 3.12). The values of the **O** and **U** entries are used to determine whether a password string supplied by the user is the correct owner password, user password, or neither. If the user password is supplied, the **P** entry determines which operations are to be permitted. A document is encrypted if an owner password, user password, or any access restriction was specified when the document was created. However, the user is prompted for a password on opening the document only if the document has a user password; this can be determined by testing the empty string as the user password (see Algorithm 3.5 below).

The value of the encryption dictionary's **P** entry is an unsigned 32-bit integer containing a set of flags specifying which access privileges should be granted when the document is opened with the user password. Table 3.14 shows the meanings of these flags. Bit positions within the flag word are numbered from 1 (low-order) to 32 (high-order); a 1-bit in any position enables the corresponding access privilege. Bits 17 through 21 are new to the revision 3 implementation of the **Standard** security handler.

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

TABLE 3.13 Additional encryption dictionary entries for the standard security handler

KEY	TYPE	VALUE
R	*number*	(*Required*) The revision number of the **Standard** security handler that is required to interpret this dictionary. The revision number should be set as follows: 2: for documents that do not require the new encryption features of PDF 1.4, meaning documents encrypted with **V** entry value of 1 and using permission bits 1–6 3: for documents requiring the new encryption features of PDF 1.4, meaning documents encrypted with **V** entry value of 2 or greater or that use the extended permission bits 17–21.
O	string	*(Required)* A 32-byte string used in determining whether a valid owner password was entered. Contains an encrypted version of the padded user password (see step 1 of Algorithm 3.2, below).
U	string	*(Required)* A 32-byte string used in determining whether a valid user password was entered. Contains an encrypted version of the fixed padding string shown in step 1 of Algorithm 3.2 below.

P	*integer*	(*Required*) A set of flags specifying which operations are permitted when the document is opened with the user password (see table 3.14).

Implementation note *Acrobat version 4.0 and PDF version 1.3 and earlier support* **Standard** *security handler revision* **R** *with a value of 2. Acrobat 4.05 and PDF 1.4 supports revision 3.*

TABLE 3.14 User password access privileges

BIT POSITION	ENABLED CAPABILITY
1–2	*Reserved; must be 0.*
3	Print document.
4	Modify contents of document (other than text annotations, but including interactive form fields).
5	Copy text and graphics from document.
6	Add or modify text annotations, modify interactive form fields and, if bit 4 is also set, allow authoring of interactive form fields and signatures.
7–8	*Reserved; must be 1.*
9	*(Revision 3 only)* Fill in existing interactive form fields (including signature fields), even if bit 6 is clear.
10	*(Revision 3 only)* Extract text and graphics for sole purpose of making the contents of the document acccessible through assistive technologies such as screen readers or Braille output devices.
11	*(Revision 3 only)* Assemble the document (insert, rotate, or delete pages and create bookmarks or thumbnails), even if bit 4 is clear.
12	*(Revision 3 only)* Print the document to a representation from which a faithful digital copy of the PDF content could be generated. When this bit is clear (and bit 3 is set), printing is limited to a low-level representation of the appearance, possibly of degraded quality. (See implementation note 118 in Appendix H.)
13–32	*(Revision 3 only) Reserved; must be 1.*

Key Generation Algorithms

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

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

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

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

   That is, if the password string is *n* bytes long, append the first (32 – *n)* bytes of the padding string to the end of the password string. If the password is omitted, treat it as an empty (zero-length) string and substitute the entire padding string in its place.
2. Initialize the MD5 hash function and input the result of step 1 to this function.
3. Input the value of the encryption dictionary's **O** entry to the hash function. (Algorithm 3.3 shows how the **O** value is computed.)
4. Treat the value of the **P** entry as an unsigned 4-byte integer and pass these bytes to the MD5 hash function, low-order byte first.
5. Pass the first element of the file's file identifier to the MD5 hash function (see Section 8.3, "File Identifiers").
6. Finish the MD5 hash and get the output of the hash. If revision 3, then loop 50 additional times, taking the output of the hash and using this as input into a new MD5 hash. The first N bytes of the output from the final MD5 hash constitute the encryption key. N is the key length as defined by the **Length** attribute in the encryption dictionary. For revision 2, N is always 5.

This algorithm, when applied to the user password, produces the encryption key used to encrypt or decrypt string and stream data according to Algorithm 3.1 on page 11. Parts of this algorithm are also used in the algorithms described below.

In addition to the encryption key, the standard security handler must provide the contents of the encryption dictionary (Tables 3.12 and 3.13). The values of the **Filter**, **R**, **P**, and **V** entries are straightforward, but the computation of the **O** (owner password) and **U** (user password) entries requires further explanation. Algorithms 3.3 and 3.4 show how to compute the values of these entries.

Algorithm 3.3 ***Computing the O (owner) value in the encryption dictionary***

1. Pad or truncate the owner password string as described in step 1 of Algorithm 3.2. If there is no owner password, use the user password instead. (See implementation note 16 in Appendix H.)
2. Pass the result of step 1 as input to the MD5 hash function. If the **R** entry is 3, then loop 50 additional times, taking the output of the hash and using this as input into a new MD5 hash.
3. Create an RC4 key using the first N bytes of the MD5 output. N is the key length as defined by the **Length** entry in the encryption dictionary. When **R** is 2, N is always 5.
4. Pad or truncate the user password string as described in step 1 of Algorithm 3.2.
5. Encrypt the padded user password string with the RC4 algorithm, using the key obtained in step 3. If **R** is 3, then loop an additional 19 times over the RC4 algorithm with the data output of each iteration being the data input for the next iteration. For each iteration (1 through 19), each byte of the key obtained in step 3 is XOR'd with the single-byte value of the loop counter.
6. Store the result of step 5 as the value of the **O** entry in the encryption dictionary.

Algorithm 3.4 ***Computing the U (user) value in the encryption dictionary n (Revision 2)***

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

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

1. Create an encryption key based on the user password string, as described in Algorithm 3.2.
2. Initialize the MD5 hash function and pass the encryption key from step 1 as input to this function.
3. Pass the first element of the file's file identifier array (the value of the **ID** entry in the document's trailer dictionary; see Table 3.12 on page 67) to the hash function and finish the hash.
4. Encrypt the 16-byte result of the hash, using the RC4 algorithm with the encryption key from step 1.
5. Do the following 19 times: Take the output from the previous RC4 encryption and pass it as input into a new RC4 encryption; use an encryption key generated by taking each byte of the original encryption key (obtained in step 1) and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 1 to 19).
6. Store the output from the final RC4 encryption as the value of the **U** entry in the encryption dictionary.

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

Algorithm 3.6 ***Checking the user password***

1. Compute an encryption key from the supplied password string, as described in Algorithm 3.2.
2. Decrypt the value of the encryption dictionary's U entry, using the RC4 algorithm with the encryption key computed in step 1.
3. (Revision 2) If the result of step 2 is identical to the fixed padding string shown in step 1 of Algorithm 3.2, the password supplied is the correct user password.

   (Revision 3) Do the following 19 times: Take the output from the previous RC4 encryption and pass it as input into a new RC4 encryption; use an encryption key generated by taking each byte of the original encryption key (obtained in step 1) and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 19 to 1). If the output from the final RC4 encryption is identical to result of step 4 of Algorithm 3.5, the password supplied is the correct user password.

Once it is determined that the password supplied is the correct user password, the key obtained in step 1 of the above algorithm can be used to decrypt the document using Algorithm 3.1.

Algorithm 3.7 ***Checking the owner password***

1. Compute an encryption key from the supplied password string, as described in steps 1 to 4 of Algorithm 3.3.
2. Decrypt the value of the encryption dictionary's **O** entry, using the RC4 algorithm with the encryption key computed in step 1.
3. *(Revision 3 only)* Do the following 19 times: Take the output from the previous RC4 encryption and pass it as input into a new RC4 encryption; use an encryption key generated by taking each byte of the encryption key obtained in step 1 and performing an XOR (exclusive or) operation between that byte and the single-byte value of the iteration counter (from 19 to 1).
4. Use Algorithm 3.2 to compute an encryption key from the decrypted value obtained as output from the last RC4 encryption.
5. Decrypt the value of the encryption dictionary's **U** entry, using the RC4 algorithm with the encryption key computed in step 4.
6. If the result of step 5 is identical to the fixed padding string shown in step 1 of Algorithm 3.2, the password supplied is the correct owner password.

Once it is determined that the password supplied is the correct owner password, the key obtained in step 4 of the above algorithm can be used to decrypt the document using Algorithm 3.1.

Adding Language Version to the Catalog

[Applies to Section 3.6.1 “Document Catalog;”]

1 Introduction

PDF 1.4 allows the addition of an optional **Version** key to the catalog dictionary to indicate the PDF language version number used in the file. This makes it possible to perform an incremental update of a PDF file when it is necessary to specify an updated version number of the PDF language used in the document.

1.1 Terminology

Version number – The phrase *version number* refers to the version of the PDF language specification that a PDF document conforms to i.e. the PDF language version as currently specified in the document’s header.

Incremental update – As described in section 3.4.5 of the PDF 1.3 language specification, a PDF document can be altered by appending information to the end of the file without altering the existing file contents. This is known as an incremental update to the file (and also known as an append save or incremental save).

2 Background

It is currently impossible to perform an incremental update of a PDF file if the version number must be updated. The version number is only specified in the header, the very first line of the file; to modify the version number you must at least alter the first few bytes of the file (and at most re-write the file entirely).

Note that the header is the only portion of a PDF file that cannot be updated incrementally.

Note: An incremental update strategy should allow the update to be completely reversed by simply removing the data appended to the file during the update. The current specification prevents such “roll-back” if the version number must be updated; updating the version number is guaranteed to completely obliterate the previous version number. It’s desirable to find a solution which allows a complete version number history to be maintained for documents which are incrementally updated.

2.1 Digital signatures

The digital signature design was based on the notion that idea that once a file was signed, all further modifications to the file would be performed using incremental updates to ensure that existing bytes are not altered. Altering any existing bytes in the file will invalidate any existing digital signatures in the file. Without a language change to either the version numbering scheme or the digital signatures mechanism, it will continue to be impossible to update the version number of a document without invalidating existing signatures. It is too late to change the existing digital signature definition at this point because of the number of signed documents already authored.

2.2 Saving files to non-traditional file systems

Current Acrobat implementations never directly modify existing bytes in a file. When saving a file, Acrobat either performs an incremental update or writes out a brand new file from scratch and then renames the new file to replace the old file. This follows accepted practices for minimizing the risk of data loss.

This methodology assumes that a temporary file can be written out and then renamed to replace the existing file. These assumptions cannot be met in certain contexts, such as when viewing a file across an HTTP connection or working with a file using OLE embedding (a Windows-specific technology). In particular one cannot overwrite the contents of the original file using a renaming scheme in these contexts. Indeed, there appears to be no one scheme for performing a full rewrite of the existing file that works across all of these contexts; a separate strategy must be developed and implemented in each case (with a good deal of effort involved for each). Extending the existing incremental update logic to these contexts is relatively easy by comparison, but currently this strategy fails when the document's version number must be changed.

3 Adding Language Version to the Catalog

PDF 1.4 allows the addition of an optional **Version** key to the catalog dictionary to indicate the PDF language version number used in the file.

TABLE 1 Addition to Table 3.15 *Entries in the catalog dictionary*

KEY NAME	TYPE	DESCRIPTION
Version	number	*(Optional)* The version of the PDF specification the file conforms to (e.g. 1.3, 1.4). If not present the document is assumed to conform to the version specified in the file's header. If present the document conforms to either the version specified by this key or in the header, whichever is greater.

In contexts where only incremental updates can be performed, a PDF author can update the version number by inserting a **Version** key into the catalog with the new version number, leaving the header intact.

When the version number specified in the file header and the **Version** key disagree as to the version number, the greater of the two values is taken as the effective version number. This is to accommodate applications that may have modified PDF 1.4 documents before full support for this change was available. It is possible for such software to roll the version number in the header up to PDF 1.4, while leaving the **Version** key in the catalog at a lower number.

3.1 Compatibility Notes

Acrobat 5.0 will avoid adding a **Version** key in the Catalog unless it has to. Once it has done so, however, it will never remove the **Version** key. For documents containing a **Version** key the implementation will attempt to ensure that the version number in the header matches the value of the **Version** key; if this is not possible it will at least ensure that the value of the **Version** key exceeds the value in the header.

Consumers which are not PDF 1.4 savvy will continue to look for the version number only at the top of the file and will not see the updated version number specified in the Catalog. Take Acrobat 4.0 as an example of an application designed to handle version numbers up to 1.3. In theory the **Version** key in the Catalog will have a value of 1.4 or greater, so the interesting cases are:

- Header specifies 1.2 or earlier, **Version** key specifies 1.4. If the user modifies the document Acrobat 4.0 will attempt to update the version number to 1.3. Since it believes the document is currently at 1.2 or earlier it will not allow an incremental update of the file when saving. Instead it will rewrite the file with a new header indicating version number 1.3. The **Version** key in the Catalog, however, will be maintained at a value of 1.4.
- Header specifies 1.3 or greater, **Version** key specifies 1.4 If the user modifies the document Acrobat 4.0 will allow an incremental update. The value of the **Version** key will be maintained at a value of 1.4.

In both of the above cases certain errors will be reported differently. If the Acrobat 4.0 viewer knew that the version number was really greater than 1.3 it would include the following string in some of its error messages:

This file contains information not understood by the viewer.

The user would also be allowed to suppress further error messages of this type. Since the viewer will not know that the version is actually 1.4 this message will not be shown and the user will not be able to suppress further errors.

Similar anomalies would occur when opening a file in earlier Acrobat viewers.

4 Web behavior

The version number would need to be available very early when opening a PDF file. The Catalog dictionary is already placed near the front of a linearized PDF file and so should be readily available even when downloading a file across an HTTP connection.

Included Data

[Applies to Section 3.6.3 “Name Dictionary”]

1 Included Data

PDF 1.3 allows PDF documents to refer to the contents of a file, which may be embedded within the document or may reside externally. In either case, the file specification associates a location in a file system with the referenced data.

PDF 1.4 allows the embedding of data independent of any file system: the data’s presence in the PDF document is all that is required. This can be done using an embedded file stream referenced by a private entry in a dictionary, but PDF 1.4 defines a standard location for the data, using the **Data** key in the document’s name dictionary (see Table 3.18 *Entries in the name dictionary*, in Section 3.6.3 “Name Dictionary”).

The mapping from user-defined strings to an embedded file stream is specified by the **Data** key in a document’s name dictionary. The specified name tree maps user-defined strings to embedded file streams containing the corresponding user-defined data.

TABLE 1 Additions to Table 3.18 *Entries in a name dictionary*

KEY	TYPE	VALUE
Data	name tree	*(Optional)* A name tree that maps strings to embedded file streams.

The relationship between the name of the data object and the embedded file is an artificial one for data management purposes. This allows embedded files to be easily identified by the author of the document in much the same way that the JavaScript name tree associates names with document level JavaScripts.

ProcSet Resource

[Applies to Section 3.72 “Resource Dictionaries,” and Table 3.20 “Entries in a resource dictionary”]

1 Procedure Sets; Obsolete for PDF 1.4

For PDF 1.4 files, the **ProcSet** resource is obsolete, though it is still required for PDF files of PDF 1.3 or earlier.

It is recommended that applications that generate PDF files specify:

```
/ProcSet [/PDF/Text/ImageB/ImageC/ImageI]
```

that is, the full complement, unless they have explicit knowledge that fewer procsets are needed.

Graphics

[Section 4.5 “Color Spaces”]

1 Support for Separation and DeviceN Is Removed

Separation and **DeviceN** colorants are not supported for additive color devices.

Glyph Widths; Predefined CMaps

[Applies to Chapter 5 Fonts; Section 5.1.3 “Glyph Positioning and Metrics”]

1 Glyph Widths Override

Implementation note:

Acrobat 5 uses the glyph widths stored in the PDF file to override the widths of glyphs in the font itself, which improves the consistency of the display and printing of the document. This addresses the situation where a document is created with a font whose widths have been intentionally altered, but the same font on the viewer’s system has not been altered.

The font with altered glyph widths might be embedded or not:

- If the font is embedded, the widths in the font should exactly match the widths in the **Widths** array in the PDF file.
- If the font is not embedded: Acrobat will override the widths in the font used in the viewer’s system, with the widths specified in the **Widths** array.

However, it is important that the glyph width override feature not be used for any purpose other than to represent an actual font that has been modified. The problem is that consumers of PDF files depend on widths in many different contexts: viewing, printing, fauxing, reflow, word-finding, and the like. Those processes may malfunction if arbitrary adjustments are made to the widths, and they don’t represent the actual glyph widths intended by the PDF producer.

It is recommended that diagnostic and preflight tools check the PDF widths against the glyph widths of an embedded font, and flag any inconsistencies. It might also be helpful if the tools could optionally check for consistency with non-embedded fonts; this is useful for checking a PDF file immediately after it is produced, when the original fonts are still available.

2 Predefined CMaps

[Applies to Section 5.6.4 "CMaps"]

2.1 Standard Character Collections for CID-keyed Fonts

Adobe Acrobat 4.0 is aware of several standard character collections, which are used as the basis for searching and copy-and-paste operations. A character collection is defined as a combination of the /Registry, /Ordering, and /Supplement fields in a CIDFont's /CIDSystemInfo dictionary. The first two fields are string objects, and the third is an integer value. The standard character collections that are recognized by Adobe Acrobat 4.0 for these purposes are as follows:

Adobe-GB1-2	(Adobe Tech Note #5079)
Adobe-CNS1-0	(Adobe Tech Note #5080)
Adobe-Japan1-2	(Adobe Tech Note #5078)
Adobe-Korea1-1	(Adobe Tech Note #5093)

Note: *The Supplement numbers indicate the highest supplement number supported by the product release. Some CMaps may be at a lower supplement number because they do not reference glyphs defined in a higher supplement.*

Adobe Acrobat 5.0 recognizes the same character collections, but with greater /Supplement values:

Adobe-GB1-4	(Adobe Tech Note #5079)
Adobe-CNS1-3	(Adobe Tech Note #5080)
Adobe-Japan1-4	(Adobe Tech Note #5078)
Adobe-Korea1-2	(Adobe Tech Note #5093)

2.2 New CMaps for PDF 1.4

Several new CMaps have been added to the core Predefined CJK CMaps for a CIDFont, as shown in Table 1.

Note: *The CMaps described in this section refer to the CMap files associated with CIDFont files, and do not refer to the "cmap" tables included in TrueType fonts. Both CMaps and cmaps specify glyph encodings, but they differ in structure and format.*

TABLE 1 Additions to Table 5.14 *Predefined CJK CMap names*

NAME	DESCRIPTION
GBKp-EUC-H	Identical to GBK-EUC-H except that the single-byte range (ASCII) uses proportional-width glyphs instead of half-width ones. Supports Microsoft's Code Page 936. Code point 0x24 is mapped to a dollar sign ($), instead of the yuan glyph.
GBKp-EUC-V	Vertical version of GBKp-EUC-H.
GBK2K-H	Supports the GB 18030-2000 character set and its mixed one-, two-, and four-byte encoding. Also supports Microsoft's Code Page 936. Code point 0x24 is mapped to a dollar sign ($), instead of the yuan glyph.
GBK2K-V	Vertical version of GBK2-H.
HKscs-B5-H	Supports the Hong Kong SCS (Supplementary Character Set), which is an extension to the Big Five character set and encoding.
HKscs-B5-V	Vertical version of HKscs-B5-H.

Note: *No new CMaps were added for the Japanese or Korean Character Collections.*

2.3 Changes to Existing CMaps

PDF 1.4 adds new Unicode entries to CID mappings for the following CMaps:

UniGB-UCS2-H
UniCNS-UCS2-H

UniJIS-UCS2-H
UniJIS-UCS2-V

These new mappings are for mapping Unicode codepoints to CID numbers for supplements that have been added to the Adobe-GB1, and Adobe-CNS1, and Adobe-Japan1 character collections. Only new mappings have been added; none of the mappings that existed in early versions of the CMaps have been changed. The UniGB-UCS2-H CMap includes mappings to CIDs in Adobe-GB1 Supplements 0 through 4; previously only supplements 0 through 2 were supported. The UniJIS-UCS2-H/V CMaps now include mappings to CIDs in Adobe-Japan1 Supplements 0 through 4; previously only supplements 0 through 2 were supported. The UniCNS-UCS2-H CMap contains mappings for Adobe-CNS1 supplements 0 through 3; previously only supplement 0 was supported.

2.4 Backward Compatibility Issues

If an Acrobat viewer version 4.05 or earlier displays a PDF that uses one of the new CMaps, an error will be displayed indicating the CMap is not found.

No characters will be displayed for the newly added code points in the modified CMaps in Acrobat versions 4.05 or earlier.

Note: *If a font is used that references one of the predefined CMaps, and its CMap has a larger /Supplement number than the standard one supported for the PDF language version being used, the CMap file should be embedded in the document, whether or not the font is embedded.*

3 PDF CMaps and Supplement Numbers

The following table lists the CMaps supported by previous versions (1.2 and 1.3) of the PDF language, and shows the current supplement number of Character Collection for each CMap.

TABLE 2 PDF 1.2 CMaps and supplement numbers

Adobe-GB1	Supplement
GB-EUC-H	0
GB-EUC-V	0

GBpc-EUC-H	0

Adobe-CNS1	**Supplement**
B5pc-H	0
B5pc-V	0
ETen-B5-H	0
ETen-B5-V	0
CNS-EUC-H	0
CNS-EUC-V	0

Adobe-Japan1	**Supplement**
83pv-RKSJ-H	1
90ms-RKSJ-H	2
90ms-RKSJ-V	2
90pv-RKSJ-H	1
Add-RKSJ-H	1
Add-RKSJ-V	1
Ext-RKSJ-H	2
Ext-RKSJ-V	0
H	1
V	0

Adobe-Japan2	**Supplement**
Hojo-EUC-H	0
Hojo-EUC-V	0

Adobe-Korea1	**Supplement**
KSC-EUC-H	0
KSC-EUC-V	0
KSCms-UHC-H	1
KSCms-UHC-V	1
KSCpc-EUC-H	0
KSC-Johab-H	1
KSC-Johab-V	1

TABLE 3 PDF 1.3 CMaps and supplement numbers

Adobe-GB1	Supplement
GB-EUC-H	0
GB-EUC-V	0
GBpc-EUC-H	0
GBpc-EUC-V	0
GBK-EUC-H	2
GBK-EUC-V	1
UniGB-UCS2-H	2
UniGB-UCS2-V	1

Adobe-CNS1	Supplement
B5pc-H	0
B5pc-V	0
ETen-B5-H	0
ETen-B5-V	0
ETenms-B5-H	0
ETenms-B5-V	0
CNS-EUC-H	0
CNS-EUC-V	0
UniCNS-UCS2-H	0
UniCNS-UCS2-V	0

Adobe-Japan1	Supplement
83pv-RKSJ-H	1
90ms-RKSJ-H	2
90ms-RKSJ-V	2
90msp-RKSJ-H	2
90msp-RKSJ-V	2
90pv-RKSJ-H	1
Add-RKSJ-H	1
Add-RKSJ-V	1
EUC-H	1
EUC-V	1
Ext-RKSJ-H	2
Ext-RKSJ-V	2
H	1
V	0
UniJIS-UCS2-H	2

UniJIS-UCS2-V	2
UniJIS-UCS2-HW-H	2
UniJIS-UCS2-HW-V	2

Adobe-Korea1	**Supplement**
KSC-EUC-H	0
KSC-EUC-V	0
KSCms-UHC-H	1
KSCms-UHC-V	1
KSCms-UHC-HW-H	1
KSCms-UHC-HW-V	1
KSCpc-EUC-H	0
UniKS-UCS2-H	1
UniKS-UCS2-V	1

TABLE 4 PDF 1.4 CMaps and Supplement numbers

(CMaps added for PDF 1.4 are marked with an asterisk)

Adobe-GB1	**Supplement**
GB-EUC-H	0
GB-EUC-V	0
GBpc-EUC-H	0
GBpc-EUC-V	0
GBK-EUC-H	4
GBK-EUC-V	1
UniGB-UCS2-H	2
UniGB-UCS2-V	1
GBKp-EUC-H*	1
GBKp-EUC-V*	0
GBK2K-H*	4
GBK2K-V*	4

Adobe-CNS1	**Supplement**
B5pc-H	0
B5pc-V	0
ETen-B5-H	0
ETen-B5-V	0

ETenms-B5-H	0
ETenms-B5-V	0
CNS-EUC-H	0
CNS-EUC-V	0
UniCNS-UCS2-H	3
UniCNS-UCS2-V	0
HKscs-B5-H*	3
HKscs-B5-V*	0

Adobe-Japan1	**Supplement**
83pv-RKSJ-H	1
90ms-RKSJ-H	2
90ms-RKSJ-V	2
90msp-RKSJ-H	2
90msp-RKSJ-V	2
90pv-RKSJ-H	1
Add-RKSJ-H	1
Add-RKSJ-V	1
EUC-H	1
EUC-V	1
Ext-RKSJ-H	2
Ext-RKSJ-V	2
H	1
V	0
UniJIS-UCS2-H	4
UniJIS-UCS2-V	2
UniJIS-UCS2-HW-H	4
UniJIS-UCS2-HW-V	4

Adobe-Korea1	**Supplement**
KSC-EUC-H	0
KSC-EUC-V	0
KSCms-UHC-H	1
KSCms-UHC-V	1
KSCms-UHC-HW-H	1
KSCms-UHC-HW-V	1
KSCpc-EUC-H	0
UniKS-UCS2-H	1
UniKS-UCS2-V	1

Output Intents for Color Critical Workflows

[Applies to Section 3.6.1 "Document Catalog" and Section 4.5.4 "CIE-Based Color Spaces"]

1 Introduction

This section describes the PDF 1.4 means for identifying the color characteristics of the intended output device associated with a PDF document. This is accomplished through an extension of the PDF Language syntax which supplements that found in the PDF Reference, version 2.0. All constructs described herein are backward compatible with the PDF 1.3 specification. As such, they may be included in compliant PDF 1.3 files.

Most sections in this document contain information that will be added to, or replace the text in the PDF Reference, version 2.0. Other sections in this document introduce new material that will be added to the PDF 1.4 specification.

2 Use of OutputIntents

The **OutputIntents** array is a repository of information about the color reproduction characteristics of one or more intended output devices. In some workflows this data is provided for informational purposes only. There is no expectation that a PDF production tool would automatically convert source ICC colorant values referring to the same base color space to the specified output space prior to generating output. Nor is it necessarily desirable that they do so when working with CMYK data which is tagged with a source ICC profile only for the purposes of characterization.

Note: *The 4-3-4 transformation performed in a CMYK conversion is likely to produce a loss of fidelity in the black component information.*

It is possible that the **DestOutputProfile** of an output intent dictionary be used as a destination profile when converting from source color spaces which are not derived from the same base colorspace. It is possible, but not required that a PDF reading application use a profile or characterization data stored in an output intent dictionary as a target profile. Acrobat 5.0 will not make use of the **DestOutputProfile** out of the box. However, a plug-in developer could develop a tool to do so.

Output intent information may vary depending on expected reproduction workflow and the tools at the production house. For instance, one print production facility may accept PDF/X-1 compliant files and have tools for processing them. Another facility may use custom Acrobat plug-ins to implement their RGB workflow for document distribution on the web. Each of these scenarios may require different sets of output characterization data. Furthermore, it is possible that one PDF file may be distributed unmodified to multiple vendors for production. This format allows for the definition of multiple output intent dictionaries that may be stored simultaneously in the **OutputIntents** array. It is expected that the purchaser of final output and service provider have prior agreement on which set of information will be used in a particular production run. The language specification intentionally does not provide a selector specifying the output intent dictionary to be used at any given time.

The output intent dictionaries supplement rather than replace information found in an **ICCBased** color space or a default colorspace. These existing mechanisms are specifically used to describe the characteristics of source color component values. Output intent information used in combination with the aforementioned source profiles will provide the capability to convert ICCBased data to that required for a specific output condition and/or enable the display and/or proofing of the intended output.

3 Defining OutputIntents

3.1 Additions to the Catalog

The **OutputIntents** array found in the catalog describes one or more possible output conditions for the entire document. Each output condition is identified with a separate dictionary entry in the **OutputIntents** array. The individual output intent dictionaries found in the array may vary in form and content. Each subordinate output intent dictionary must contain a value for the **S** key which is used to uniquely identify the form and content.

TABLE 1 Additions to Table 3.15 *Entries in the catalog dictionary*

KEY	TYPE	SEMANTICS
OutputIntents	Array	*(Optional)* Describes one or more intended output conditions applicable to the entire document.

3.2 Known forms of Output Intent Dictionaries

Below, the known forms of output intent dictionaries are described. The form and content of various output intent dictionaries will be uniquely identified by the value in the **S** key which is required. The value for the **S** key should conform to the guidelines described in Appendix E.

Definition of PDF/X Output Intent Dictionary

The table below describes one form of an output intent component dictionary. It contains the information used by a PDF/X based workflow system. These contents will be found associated with the **S** key having a value of *GTS_PDFX* in the output intent dictionary. This document describes the general syntax requirements for use of a PDF/X output intent dictionary. The PDF/X family of international standards (ISO 15930) identifies multiple conformance levels. At any level, the PDF/X standard may prescribe further restrictions on the use of certain keys and their associated semantics. Compliance with such standards requires that precedence be given to the requirements stated therein.

The PDF/X family of international standards allows PDF creators to include a specific output profile or identify a printing condition by name. Use of a profile explicitly describes the color capabilities of the intended output device insuring that the intended printing condition matches, or is appropriate for, the named condition. The ICC Characterization Data Registry of standard printing conditions describes printing conditions for which a set of characterization data is separately identified. When the **OutputConditionIdentifier** field is present in a GTS_PDFX output intent dictionary, the consuming workflow system is responsible for insuring that the intended output condition matches or is appropriate for the named condition.

When all source data in the page content is characterized and the contents of the **OutputConditionIdentifier** match an entry in an industry standard output condition registry, such as the ICC Characterization Data Registry of standard printing conditions, inclusion of the profile data in the **DestOutputProfile** key is optional. If a non-standard condition is identified, such as *Custom*, an ICC profile shall be included in the **DestOutputProfile** and a free form description is required in the **Info** key. It is recommended that creators provide a human readable description in the **OutputCondition** in order facilitate display in a user interface.

The device to PCS (AToB) transform found in a **DestOutputProfile** can be used to enable the remapping of the uncharacterized source color values having a similar base color space to some other destination color space. A typical use of this would be for screen preview or hardcopy proofing. The default behavior of the Acrobat 5.0 application does not make use of this mechanism.

TABLE 2 PDF/X OutputIntent Dictionary (New for PDF 1.4)

KEY	TYPE	SEMANTICS
Type	Name	*(Required)* Object type. The value must be ***OutputIntent***
S	Name	*(Required)* The output intent subtype. The remaining entries below apply to this type. At present, *GTS_PDFX* is the only valid type. Future extensions may introduce other types, which will most likely have a different set of additional entries.
DestOutputProfile	ICCProfile/stream	(*Optional* in presence of **OutputConditionIdentifier**. Otherwise required.) Any valid bi-directional ICC profile which describes the transformation to device colorants. This is not an ICCBased colorspace definition, but rather an ICC Profile stream as described in table 4.16. When used in this context the restriction described therein for output colorspaces are applicable.

OutputConditionIdentifier	String	*(Required)* Identification of the intended output condition. It may be the name of a printing condition maintained in an industry standard registry. This field should be human or machine readable. It may be used for presentation in the user interface when the **OutputCondition** key is not defined. The value *Custom* is recommended for situations where the condition is not an established standard. This indicates that the consuming application should use information in the Info key to further identify the intent.
RegistryName	String	*(Optional)* A string specifying the registry where the **OutputConditionIdentifier** is defined. Conventionally, this is a URI.
OutputCondition	TextString	*(Optional)* Concise identification of the output condition in the form of a text string that is human readable. It is the intended and preferred key for use in user interface presentation of this information. It may be localized.
Info	TextString	*(Optional)* Additional information or comments about the output condition.

4 Changes to restriction on ICC profile stream attributes

The existing description of table 4.16 and associated text in the PDF Reference, version 2.0, identifies a number of restrictions on the contents of an ICC profile stream which arise from the need to use it as a source color space. Those restrictions do not apply to a profile used as a DestOutputProfile.

The following tables and paragraphs describe additions and changes to the existing reference material.

TABLE 3 Modifications to Table 4.16 *Entries in an ICC profile stream dictionary*

KEY	TYPE	SEMANTICS
Alternate	Array or Name	Note: Change to *Optional* and add the following to the beginning of the description: *(Optional)* An alternate source color space to be used in case the one specified in the stream data is not supported. Note: The following will be added at the start of the second paragraph:

Note that there is no conversion of source color data when using the alternate color space.

Note: Add the following to the end of the second paragraph:

The alternate value is ignored when the stream describes an output colorspace.

Last paragraph on page 175

The first sentence of the paragraph will be reworded as follows:

When the **ICCBased** color space is being used as a source color space, the “to-CIE” profile information (AtoB in ICC terminology) is used; The “from CIE” (BToA) information is used only for destination profiles such as those found in the **OutputIntents** entry of the catalog.

Example OutputIntents array

Example 4.1 OutputIntents Array using Industry Standard Identifier

```
24 0obj % ICC Profile stream
<<
/N 4
/Length 1605
/Filter/ASCIIHexDecode
>>
stream
00 00 02 0C 61 70 ....
endstream
endobj

/OutputIntents [
<<
/Type/OutputIntent
/S  /GTS_PDFX
/DestOutputProfile 24 0 obj
/OutputConditionIdentifier(CGATS TR 001)
/OutputCondition(CGATS TR 001 \(SWOP\))
/RegistryName (http://www.color.org)
>>
]
```

Example 4.2 OutputIntents Array using Custom Identifier

```
/OutputIntents [
<<
/Type/OutputIntent
/S  /GTS_PDFX
/DestOutputProfile24 0 obj
/OutputConditionIdentifier(Custom)
/OutputCondition(Coated)
/Info   (Coated 150lpi)
>>
]
```

New Bookmark Properties

[Applies to Section 7.2.2 "Document Outline"]

1 New Bookmark Properties

PDF 1.4 adds two new entries to Table 7.4, *Entries in an outline item dictionary,* to specify font style and color for text in a bookmark.

TABLE 1 Additions to Table 7.4 *Entries in an outline item dictionary*

KEY	TYPE	DESCRIPTION
C	array	*(Optional)* An array of three numbers in the range 0.0 to 1.0 representing the components of a color in the DeviceRGB color space. The color will be used to display the text for the outline entry. Default value: [0 0 0].
F	integer	*(Optional)*A set of flags specifying various characteristics of the outline entry. Default value: 0. The defined flags are: Bit 1: If set, display the entry using an italic font. Bit 2: If set, display the entry using a bold font. All other bits are reserved and must be set to 0.

Annotations

[Applies to Section 7.4 "Annotations"]

1 Annotation naming

In order to better identify individual annotations across editing sessions, an additional attribute has been added to the description of annotations.

TABLE 1 Addition to Table 7.9 *Entries common to all annotation dictionaries*

KEY	TYPE	SEMANTICS
NM	Text	*(Optional)* A string unique among all annotations on a given page

2 Background color

For annotations of type *Line*, *Circle*, or *Square*, an additional attribute has been added to support background colors.

TABLE 2 Addition to Table 7.17 *Additional entries specific to a line annotation*, and Table 7.18 *Additional entries specific to a line annotation*

KEY	TYPE	SEMANTICS
BG	array of numbers	*(Optional)* An array of three numbers representing the fill color used by Line, Circle, and Square annotations. A color is specified as an array of

		three numbers in the range 0 to 1, representing a color in DeviceRGB.

3 Line endings

For annotations of type *Line*, the **LE** attribute has been added to describe line endings.

TABLE 3 Additions to Table 7.17 *Line annotation attributes*

KEY	TYPE	SEMANTICS
LE	array	*(Optional)* An array of two names identifying line ending styles. The first name represents the line ending which corresponds to the endpoint described by the first two coordinates in the **L** attribute. The second name represents the line ending which corresponds to the endpoint described by the last two coordinates in the **L** attribute. The following line ending styles are defined:

Square line ending	A square is drawn (and optionally filled with the color specified by **BG**) at the appropriate endpoint
Circle line ending	A circle is drawn (and optionally filled with the color specified by **BG**) at the appropriate endpoint.
Diamond line ending	A diamond shape is drawn (and optionally filled with the color specified by **BG**) at the appropriate endpoint.
Line line ending	Two short lines are drawn from the appropriate endpoint such that the line proper bisects the small angle.
Triangle line ending	Two lines are drawn as in the *Line* line ending style. These two lines are connected with a third line forming a triangle. This triangle is optionally filled with the color specified by **BG**.

4 Quadding support

For annotations of type *FreeText*, the **Q** attribute has been added to support quadding (text justification).

TABLE 4 Addition to Table 7.16 ***Additional entries specific to a free text annotation***

KEY	TYPE	SEMANTICS
Q	integer	*(Optional)* Specifies whether the text contained in **Contents** is left-justified (0), centered (1) or right-justified (2). The default value is 0.

5 Squiggly annotations

The *Squiggly* annotation is like the *StrikeOut*, *Highlight*, or *Underline* annotations. Its appearance is a jagged line underlining the quads indicated by the **QuadPoints** attribute.

ADDITION TO TABLE 7.19 ***ADDITIONAL ENTRIES SPECIFIC TO MARKUP ANNOTATIONS***

KEY	TYPE	SEMANTICS
Subtype	name	*(Required)* Annotation subtype. Always **Squiggly**.

6 Markup annotation definition

In addition to Acrobat Form fields, FDF submitted through Acrobat Forms may contain markup annotations. Markup annotations include the following annotation types: *Text*, *Sound*, *FreeText*, *Stamp*, *Line*, *Square*, *Circle*, *StrikeOut*, *Highlight*, *Underline*, *Squiggly*, *Ink*, *FileAttachment*, and *Popup*.

7 Annotation Border Style

[Applies to Section 7.4.5 “Annotation Types;” Subsection: “Link Annotations”]

Add note: The link annotation uses the **Border** key to specify the border style; the **BS** key is not recognized.

8 Markup annotation opacity

[Applies to Section 7.4.1 “Annotation Dictionaries”]

For markup annotations, a general opacity attribute **CA** has been added. This attribute does not refer to stroke opacity or fill opacity but to a high level notion of opacity for the annotation as a whole. The implicit blend mode is /Normal.

TABLE 5 Addition to Table 7.9 *Entries common to all annotation dictionaries*

KEY	TYPE	SEMANTICS
CA	number	*(Optional)* A number representing the opacity of the markup annotation. The default value is 1.

9 Trap Network Annotations

[Applies to *Trap Network Annotations* in Section 8.6.3 “Trapping Support”]

The *TrapNet* annotation dictionary has the following changes:

- The **Version** key and **AnnotStates** keys are now *optional* instead of *required*

- A new key has been added to Table 8.22, *Additional entries specific to a trap network annotation*:

TABLE 6 Addition to Table 8.22 ***Additional entries specific to a trap network annotation***

KEY	TYPE	VALUE
LastModified	date	*(Optional)* The date the TrapNet was last modified.

BACKGROUND NOTE:

The **TrapNet** annotation has required keys **Version** and **AnnotStates** whose purpose is to specify when the appearance of a trapped page has changed, so that trapping-aware software can determine that the trapping annotation is no longer valid. The PDF 1.3 specification describes an algorithm for using the **Version** key to detect that page content has changed, but use of that key or the **AnnotStates** key can produce false results

For PDF 1.4, the **LastModified** key in a **TrapNet** can also be used in conjunction with the **LastModified** key in a Page dictionary, in a workflow where that key is up-to-date. The Page **LastModified** key was designed with a similar purpose in mind, but it depends on voluntary cooperation by applications and plug-ins. Currently nothing in Acrobat insures that the Page **LastModified** is correct, but it can still be useful in a controlled workflow.

Implementation note: Distiller 5.0 will extend the capabilities of pdfmark to enable putting date strings into a dictionary that match the **CreationDate** of the PDF Distiller is making.

New Trigger Events

[Section 7.5.2 “Trigger Events”]

1 Document Level Triggers

This feature allows the execution of a JavaScript depending on the action a user performs on a PDF document. A dialog is provided so that a JavaScript can be specified to any of the existing five action triggers. The new triggers are: *Document Will Close*, *Document Will Save*, *Document Did Save*, *Document Will Print*, and *Document Did Print*. There is no *Document Open* action trigger because document-level JavaScripts already exist.

These new document triggers require that the **AA** (Additional Actions) key attribute be added to the PDF Catalog dictionary, as shown in Table 1 below.

TABLE 1 Addition to Table 3.15 *Entries in the catalog dictionary*

KEY	TYPE	SEMANTICS
AA	dictionary	*(Optional)* An additional-actions dictionary, providing actions for the entire document.

2 The Trigger Types

These are the trigger types allowed for document-level actions. There can only be one action per trigger. The action type is limited to the JavaScript type.

TABLE 2 Table 7.28 *Additional Actions attributes*

KEY	TYPE S	SEMANTICS
DC	dictionary	(*Optional*, Defined only for Document objects) The action to be executed prior to the closure of a document. The action type is always a JavaScript action.
WS	dictionary	(*Optional*, Defined only for Document objects) The action to be executed prior to saving a document. The action type is always a JavaScript action.
DS	dictionary	(*Optional*, Defined only for Document objects) The action to be executed after a document is saved. The action type is always a JavaScript action.
WP	dictionary	(*Optional*, Defined only for Document objects) The action to be executed prior to printing a document. The action type is always a JavaScript action.
DP	dictionary	(*Optional*, Defined only for Document objects) The action to be executed after a document is printed. The action type is always a JavaScript action.

Forms-Related Changes to PDF and FDF

[Applies to Section 7.6 “Interactive Forms”]

1 Multiple Selection for Fields of Type Listbox

In order to specify multiple selection, the *multiple selection* flag, available to fields of type *listbox*, can be set to indicate that the field allows multiple selection. Fields that have the multiple selection flag set to *on* may have an array of strings as a value.

Table 7.44 contains updates for the **Ff** and **V** keys for all field dictionaries.

TABLE 1 Addition to entry in Table 7.44 *Entries common to all field dictionaries*

KEY	TYPE	SEMANTICS
Ff	integer	*(Optional, inheritable)* Flags. The following flag is added for PDF 1.4: bit 22:Multiple selection flag. Indicates that the field allows multiple selection.

[Replace description of V in Table 7.44 with the following:]

(Optional, inheritable) Field value. The value of a **listbox** field is of type text if only one element is currently selected. If multiple elements are selected, then it is an array of text elements. Each element in **V** corresponds to the **Opt** array element that is currently selected. If that **Opt** array element is text,

then the value is that text. If it is an array, then the value is the text found in the first element of that array. The default value of this attribute is null.

2 File Select Control

In order to support functionality equivalent to a “file select” control, there is a new file select flag available to fields of type text, which when set indicates that the field’s value represents the pathname of a file whose contents may be submitted with the form. On form submission (see 7.6.4, *Form Actions – SubmitForm Action*), a text field that has the file select flag set to *on* causes the contents of the file that it is pointing to, to be submitted. If the chosen submission format is HTML-compatible, then the submission uses the content-type multipart/form-data. For FDF submission, the value of **V** in the FDF is a dictionary using the same format as that used by file attachment annotations when they get exported as FDF (see Table 7.23, *Additional entries specific to a file attachment annotation*).

TABLE 2 Addition to Table 7.49 *Field flags specific to text fields*

KEY	TYPE	SEMANTICS
Ff	integer	*(Optional, inheritable)* Flags. The following flag is added in PDF 1.4: bit 21: File select flag. Indicates that the field’s value represents the pathname of a file whose contents may be submitted with the form.

3 Multiple Options with the Same Value

For fields of type **listbox** or **combobox**, if two or more options have the same value, the currently selected options can be specified using the **I** key. This feature is typically needed when the item names (as shown on the form at fill-in time) are different, yet the export values are the same. The **Opt** array is composed of elements, each of which may itself be an array with two text elements. The **export** value is the first, the item **name** is the second.

TABLE 3 Addition to Table 7.52 *Additional entries specific to a choice field*

KEY	TYPE	SEMANTICS
I	array of integers	*(Optional, inheritable)* An array of integers sorted in ascending order, that represents the zero-based indices of the currently selected elements in **Opt**. The **I** key is required when two (or more) elements in **Opt** have the same export value, or when the value of the field is an array. **I** is allowed to be used, however, even if that's not the case. If the value **V** of the field does not match the option(s) that **I** points to, then **V** takes precedence.

4 Submit Using XML Flag

The following addition has been made to Table 7.54 in the PDF Reference:

TABLE 4 Addition to Table 7.54 *Additional entries specific to a submit-form action*

KEY	TYPE	SEMANTICS
Flags	integer	*(Optional)* The binary value of the integer is interpreted as a collection of flags that define various characteristics of the action. The default value is 0. The following flag is added for PDF 1.4: bit 6: XML flag. If this bit is 0, then the data is sent using the format indicated by bit 3. Otherwise it is sent in XML format.

5 Changes to FDF

[Section 7.6.6 Forms Data Format; Subsection “FDF Catalog”, page 463]

TABLE 5 Addition to TABLE 7.61 *Entries in an FDF catalog dictionary*

KEY	TYPE	VALUE
Version	name	*(Optional)* The version of the PDF specification this file conforms to (e.g. 1.3, or 1.4). If not present the document is assumed to conform to the version specified in the file’s header. If present the document conforms to either the version specified by this key or in the header, whichever is greater.

Implementation note: Due to a bug in versions of Acrobat prior to 5.0, which would prevent Acrobat from accepting anything different, the FDF header (see “FDF Header” on page 462) will remain “%FDF-1.2.”

TABLE 6 Additions to TABLE 7.62 *Entries in an FDF dictionary*)

KEY	TYPE	VALUE
Differences	stream	*(Optional)* A stream containing all the bytes in all incremental updates (i.e. append saves—see Section 3.4.5 “Incremental Updates”) made to the PDF that is being submitted from, since the time it opened until the time the FDF was submitted. An incremental save is automatically performed just before the submission takes place, in order to capture all the changes made to the document. See also *IncludeAppendSaves* in Table 7.55 (PDFR). Note that even if multiple sets of changes are made to the document, each followed by a submission, the full set of incremental updates (all the way back to when the document first opened) is included each time. It is possible for an FDF to contain both form fields (i.e. a **Fields** array) and/or annotations (i.e. an **Annots** array), as well as a **Differences** stream. In this case, the **Differences** stream will capture any changes that were made to the form fields and annotations, so the latter are seemingly redundant. This redundancy may be convenient for a server that can simply save the **Differer-**

		ences stream without looking at it and process the submitted form fields and annotations directly. Although one would normally expect the form fields and annotations in the FDF to be consistent with the ones captured in the **Differences** stream, PDF makes no guarantee of such consistency. In particular, if the **Differences** stream contains a digital signature, only the values of the form fields given in the **Differences** stream can be considered trustworthy under that signature.
Target	string	*(Optional)* Used to specify the name of a browser frame where the document pointed to by the **F** key is to be opened. This mimics the *target* attribute in HTML <href> tags.
EmbeddedFDFs	array of file specifications	*(Optional)* An FDF can be a carrier for multiple embedded FDFs within it. Each of the file specifications in the array contains an embedded file stream (see Section 3.10.3 "Embedded File Streams"). Each of the embedded FDFs may optionally be encrypted using RC4 encryption with a 40-bit key. If this is the case, then the stream dictionary describing the embedded, encrypted FDF contains the additional entry shown in Table 7 below (To follow Table 7.62, PDFR). The encryption key is computed using an MD5 hash, using as input a padded user-supplied password. This computation consists of steps 1 and the first part of 2 in *Algorithm 3.2, Computing an encryption key* on page 69 in PDFR).

TABLE 7 New Table to follow Table 7.62 ***Additional entries in an embedded, encrypted FDF file stream dictionary***

KEY	TYPE	VALUE
EncryptionRevision	integer	*(Required if the FDF is encrypted)* The number 1.

6 Using Unicode Values / Buttons with the Same Export Value

As of PDF 1.3, the value (and default value) of a field of type checkbox or radio button, if not null, is of type name. Recall that for fields of type checkbox or radio button, the value of the field is equal to the name that is used to identify the *on* appearance of the widget annotation that is currently checked. The value is null if none of the Widget annotations belonging to the field are checked. Using a name to represent the value constitutes a limitation in that Unicode strings are not allowed. In order to support non-Roman locales, the following attribute is added in PDF 1.4:

TABLE 8 New Table, (Section 7.6.3): *Additional entries to a button field*

KEY	TYPE	SEMANTICS
Opt	array of text	*(Optional, inheritable)* This key is used when the value of the field cannot be represented in PDFDoc encoding. The elements in **Opt** are text strings, where the ith string represents the export value of the ith annotation in the **Kids** array (if there is only one annotation, there may not be a **Kids** array, and instead the annotation shares the dictionary with the field). When the **Opt** array is present, the names used to represent the *on* state in each **AP** dictionary are computer-generated numbers (starting with "0" for the first annotation of the field). **Opt** is also used when two or more annotations have the same *on* value. In this case, **Opt** will include two or more entries that are equal. Since each annotation has a different computer-generated number to represent its *on* state, this allows checking a single annotation at a time while permitting duplicate export values.

7 New Field Flags

[Section 7.6.3 Field Types; Subsection: Interactive Forms; page 434]

TABLE 9 Additions to Table 7.49 *Field flags specific to text fields (page 448)*

BIT POSITION	NAME	MEANING
23	DoNotSpellCheck	If set, the field will not be spell checked. Note: this flag is also applicable to editable

		combo boxes (i.e. *Choice* fields that have the *Edit* bit set, see Table 7.51).
24	DoNotScroll	If set, the field will not scroll to accommodate more text than fits within the rectangle. For single line fields, this means horizontally. For multi-line, it means vertically. Once the field is full, no more text will be accepted.

8 JavaScript and FDF

The current FDF specification describes how an FDF can carry a new value for the actions (**A** or **AA**) of a field, and this includes JavaScript actions. The following extensions enable adding document-level scripts to the document, as well as immediate execution of scripts at FDF import time.

The following new key is added to the root object of an FDF:

TABLE 10 Addition to Table 7.62 Entries in an FDF dictionary

(KEY	TYPE	SEMANTICS
JavaScript	dictionary	*(Optional)* See Table: "JavaScript dictionary entries" 11 below.

The entries in the JavaScript dictionary are listed below (a new table to be added).

TABLE 11 JavaScript dictionary entries (New table to be added to Section 7.6)

KEY	TYPE	SEMANTICS
Before	string or stream	*(Optional)* The string or the contents of the stream contains the JavaScript to be executed just before the FDF gets imported.

After	string or stream	*(Optional)* The string or the contents of the stream contains the JavaScript to be executed right after the FDF gets imported.
Doc	array	*(Optional)* The even elements of this array are script names, and the odd ones are strings or streams containing scripts. These scripts will be added to any other document level scripts already present in the Names tree (see *JavaScript Actions* in section 7.6.4, *Form Actions*) and then executed. This happens before the *Before* scripts (if any) get executed.

9 Additional Submit Form Action Flags

Two submission formats have been added to the Submit Form action. Digital Signatures needs the ability to submit the whole PDF document to the web server. The document may contain signatures, and sending the complete PDF is one of two new ways to transmit those signatures to the server. The other is to send just the "append saves" of the PDF, embedded within an FDF. There is no other mechanism in FDF to transmit just the digital signatures.

Also, PDF 1.4 adds several submission options. The ability to include *markup* annotations inside the submitted FDF has been added (see Section *6, "Markup annotation definition* in the chapter on "Annotations" in this document). See the **Annots** key in Table 7.62 (PDFR) *Entries in an FDF dictionary* for a description of how annotations are represented inside FDF.

Some explanation is in order with respect to the *ExclNonUserAnnots* flag in Table 7.55 below. For collaboration, **T** is being christened the *owner* of the annotation. Given this assumption, *ExclNonUserAnnots* simply filters annotations based on their **T** key and collaboration's idea of the current "user." If "owner" != "user," don't include the annotation in the resultant FDF. This allows a collaboration backend to be a dumb CGI script that simply swallows and regurgitates whole FDFs. This scenario would work something like this:

> A user begins collaborating on a document. The collaboration engine does a submit to a CGI script (what it submits is not important... presumably, it submits essentially nothing.) The CGI script sends back a single FDF containing JavaScript that simply does several more submits – one per person collabo-

rating on the document. Each of those submits results in the CGI script sending back a single FDF containing the annotations for a single person, and that person alone. All of those FDFs together comprise all annotations that pertain to the document.

The user is done collaborating and wants to submit his or her annotations. The collaboration engine does another submit to a CGI script. This time, it does submit something. It submits annotations. But, it only submits the annotations owned by the current user. The CGI script just stores this FDF somewhere, byte for byte, ready to be sent to the next person who collaborates on that document. The reason for the *ExclNonUserAnnots* flag is also related to the scenario described above. It simply gives the collaboration server the ability to move around PDF files if it sees fit. Say a PDF file is being collaborated upon at http://foo.bar/doc1.pdf. People put lots of annotations on it and the server is collecting lots of FDFs and everyone is happy. At some point, someone realizes that hierarchies can be good and moves doc1.pdf to http://foo.bar/Docs/doc1.pdf. If there are no /F keys, everything just works (assuming the server knows about the move.) If those FDF files did have /F keys, as soon as you try to collaborate on the document in its new location, the FDFs the server has lying around will cause Acrobat to try to look for the document in its old location.

The encrypted/embedded FDF stream (see Table Table 6, above) addresses this issue as well. The encryption part is to give confidence to people wanting to use a collaboration server to collaborate on sensitive topics. It avoids leaving sensitive text in what is essentially plain text, especially if the server is run by a third party. If the client encrypts the FDF before it is sent, the server can store FDFs, but text cannot be extracted from them. So, collaboration works without the server ever knowing what the annotations actually are.

As for the ability to embed multiple FDFs, this can be used as an optimization so that a server could send back the annotations for all users in one single FDF. Note that with some work, a server could stitch together even encrypted FDFs while still not being able to decrypt them.

TABLE 12 Additions to Table 7.55 *Flags for submit-form actions*

BIT POSITION	NAME	MEANING
7	IncludeAppendSaves	If set, the submitted FDF includes the incremental updates (i.e. the append saves) made to the PDF (see the **Differences** key in Table Table 6 above). Only applicable if bit 3 (ExportFormat) is 0.
8	IncludeAnnotations	If set, the submitted FDF includes the annotations in the PDF. Only applicable if bit 3 (ExportFormat) is 0.
9	SubmitPDF	If set, will submit the PDF itself (i.e. MIME type application/pdf). If this bit is set, then all the other flag bits are ignored except for bit 4 (GetMethod)
10	CanonicalFormat	If set, will convert any submitted field values that represent dates to standard format (see Section 3.8.2 *Dates*). The interpretation of a form field as a date is not specified explicitly in the field itself, but only in the JavaScript that processes the field.
11	ExclNonUserAnnots	If set, will exclude any annotations that are not owned by the current user. Only applicable if bit 3 (ExportFormat) is 0 and bit 8 (IncludeAnnotations) is 1.
12	ExclFKey	If set, the submitted FDF will exclude the **F** key. Only applicable if bit 3 (ExportFormat) is 0.

Referenced PDF

[Applies mainly to Section 8.4.3 “Logical Structure” and Section 8.6.4 “Open Prepress Interface”]

1 Introduction

Referenced PDF enables the use of one PDF document to reference another PDF document for page content. The referenced PDF page might contain a logo to be used on multiple pages, or a high resolution image. The document containing the reference must contain a proxy (that which is used in place of the referenced document page), which might be an image or text. The proxy may be a low-resolution image of the referenced PDF, or, it may be a gray box, or, text referring to the referenced PDF, or, some other means of indicating that a proxy is being used.

The behavior of a referenced PDF depends upon the applications that view and operate upon it. A simple application may display the proxy only, and ignore the reference. A more sophisticated application may look for the reference at view or print time and use the referenced PDF if it is available. An application may also provide a user interface to allow editing and updating of referenced PDF links.

Any XObject Form may contain a reference to a referenced PDF. The reference object is a dictionary with the name **Ref**.

TABLE 1 XObject Form attribute for referenced PDF

KEY	TYPE	SEMANTICS
Ref	dictionary	*(Optional)* Contains information about the referenced PDF document. The document that con-

tains this XObject is the container of the referenced PDF and the XObject is the proxy object for it.

The **Ref** object contains the following attributes.

TABLE 2 Referenced PDF attributes

KEY	TYPE	SEMANTICS
F (File)	file specification	*(Required)* The file containing the referenced PDF.
Page	number or text	*(Required)* The page number or page name of the referenced PDF. If a name, it refers to the Page label of a page.
ID	array	*(Optional)* The File **ID** of the referenced PDF. This can be used to determine if the referenced PDF has changed since the **ID** was last updated. It can also be used to verify that the referenced PDF file is the same document as the one that was originally referenced.

A referenced PDF is specified by a file specification and a page number. The file **ID** should be used to detect whether the referenced PDF has been modified or rewritten. Note that the reference is weak in that the actual page could be changed or replaced which could invalidate the reference inadvertently.

When the referenced PDF replaces the proxy, it is transformed according to the matrix in the XObject Form. As for an XObject Form, the referenced PDF is clipped to the bounds of the Form's **BBox**. Note that the combination of the matrix and the Form **BBox** implicitly define the bounding box of the referenced page. This bounding box will typically be the **CropBox** or the **ArtBox**, but is not constrained to any of the known BBoxes.

When a referenced page contains annotations, those annotations that contain a printable, unhidden, visible appearance object (**AP** dict), must be included in the rendering of the referenced page. If the proxy is a snapshot of the referenced PDF, then it must also include the annotation appearances. In order to represent these appearances in an XObject Form they must be converted into part of the Form's stream, either as subsidiary XObject Form's or flattened into the Form's stream.

Note that a referenced PDF may also be an Embedded File within the containing PDF.

2 Structure

A referenced PDF page may contain structure information. This information should be ignored when using referenced PDF.

Typically, for a multi-page document, the structure tree will refer to elements on the page as part of a larger set of structure elements. In such a case it does not make sense to incorporate structure information from the referenced page into the containing document. For one-page documents, or, page-oriented documents, a structure subtree may be present which wholly contains the structure information for the referenced page. In this case it is possible to incorporate the structure tree for the proxy PDF into the structure of the containing document. At this time, it is not possible to refer to a structure subtree which is in an external document such as the referenced PDF.

3 Optimization

When streaming an XObject Form using the features of Linearized PDF, an application may display the proxy as the initial rendering of the Form. If the application supports rendering of the referenced PDF, it may defer retrieval and rendering of the referenced PDF.

All XObjects occur after the page Contents, thus are already deferred in PDF 1.3 and earlier. The use of the **Ref** object will result in an additional level of deferral. The **Ref** object and its contents should be direct objects to avoid another trip to the server for the information in the **Ref** dict.

4 Page Insertion/Deletion

There are no known effects.

5 Printing and OPI

When printing, an application may emit one of the following for the referenced PDF:

- The proxy XObject Form
- The referenced PDF page
- The OPI object for the XObject Form, if present

The selection of which object will depend upon user choices and the purpose of the print job. Simple applications may always print the proxy. More sophisticated applications will support use of the referenced page, or, an OPI high-resolution image. When generating OPI comments in a PostScript stream, an OPI producer may refer to the OPI high-resolution image, and include the proxy or the referenced page as the OPI low-resolution image.

Natural Language Specification

[Applies to Chapter 8 “Document Interchange,” Section to be added]

1 Introduction

Specification of the language used for text in a PDF document can increase the accessibility of that document for disabled users. For example, with the language correctly identified, text-to-speech engines can properly vocalize the text, either via a screen reader or some more direct invocation of a text-to-speech engine.

2 Language Specification

The language of a word in a document is determined in a hierarchical fashion by the language specification keys in the **Catalog** and in structure element objects, and by properties attached to marked content with the **Span** tag. **Lang** is the optional language specification key that indicates the language of the text in this object. The value associated with **Lang** is of type *string*, and an empty string indicates that the language is unknown.

2.1 Language Key Values

If it is not empty, the value of this key is a language identifier as defined by [IETF RFC 1766], “Tags for the Identification of Languages,” described in section 2.12 of the XML 1.0 Specification, http://www.w3.org/TR/REC-xml.

```
LanguageID  ::= Langcode (‘-’ Subcode)*
Langcode ::= ISO639Code |  IanaCode |  UserCode
ISO639Code ::= ([a-z] | [A-Z]) ([a-z] | [A-Z])
```

```
IanaCode ::= (‘i’ | ‘I’) ‘-’ ([a-z] | [A-Z])+
UserCode ::= (‘x’ | ‘X’) ‘-’ ([a-z] | [A-Z])+

Subcode   ([a-z] | [A-Z])+
```

The *Langcode* may be any of the following:

- a two-letter language code as defined by [ISO 639], “Codes for the representation of names of languages.”
- a language identifier registered with the Internet Assigned Numbers Authority [IANA]; these begin with the prefix “i-” (or “I-”).
- a language identifier assigned by the user, or agreed on between parties in private use; these must begin with the prefix “x-” or “X-” in order to ensure that they do not conflict with names later standardized or registered with IANA.

There may be any number of Subcode segments; if the first subcode segment exists and consists of two letters, then it must be a country code from [ISO 3166], “Codes for the representation of names of countries.” If the first subcode consists of more than two letters, it must be a subcode for the language in question registered with IANA, unless the Langcode begins with the prefix “x-” or “X-”.

Although these values are case insensitive, it is customary to give the language code in lower case, and the country code (if any) in upper case.

2.2 Catalog Object key:

This key specifies the language of all text in the document except where overridden by language specifications for structure element objects or marked content. This default applies both to text strings and content strings. See section 3, “Determining the Language of Text in a Document” (below) for how to override the default language in a text string.

2.3 Structure Element key:

This key specifies the language of all text in this structural element except where overridden by language specifications for nested structural elements or marked content.

2.4 Marked Content Property List key:

This key specifies the language of all text within this marked content except where overridden by language specifications for nested marked content. If the marked content is non-structural (that is, has no /MCID entry), then the tag for the marked content should be /Span.

3 Determining the Language of Text in a Document

There are two scope hierarchies for the language specification that interact in sometimes unexpected ways. There is the structure hierarchy for the document and there is the hierarchy defined by the nesting of marked content objects within the page stream.

The document default language and structure tree determine the default language for a marked content that is a structure leaf. Within that marked content, the language can be changed within a **Span** marked content by using the **Lang** property.

The following example shows the use of language specifications both in the structure tree and in marked content:

Example 1:

```
  1 0 obj        % Structure element
     << /Type /StructElem
      /Lang (en)% Language specification
      …
      /K << /Type /MCR
          /Pg …% Page containing the marked content sequence
          /MCID 0% Marked content identifier
          >>
     >>
  endobj
  2 0 obj        %Page’s content stream
  << /Length … >>
  stream
    …
 /P <</MCID 0>> BDC
 (When we go out for Mexican food, I can never resist trying the)  Tj
```

```
    /Span <</Lang (es)>> BDC (guacamole) Tj EMC
    (to see how it compares with my own.) Tj
    EMC
    …
  endstream
  endobj
```

The language of any page content text that is not in the structure tree (for example, in an unstructured PDF file) is specified by the **Lang** key of the Catalog unless the text is within a marked content containing a **Lang** property.

For an example of an admittedly contrived, though legal, interaction between these two language specification hierarchies, consider the following page where only part of the page content is contained in the structure tree.

```
Example 2:

1 0 obj        % Structure element
      << /Type /StructElem
       /Lang (en)% Language specification
       …
       /K << /Type /MCR
           /Pg …% Page containing the marked content sequence
           /MCID 0% Marked content identifier
           >>
      >>
   endobj


2 0 obj        %Page's content stream
<< /Length … >>
stream
/Span <</Lang (es)>> BDC (Hasta la vista,) Tj
/P <</MCID 0>> BDC
(Baby! Arnold won't be back for new Terminator installment.) Tj
EMC
EMC
endstream
endobj
```

Although the **Span** marked content in the content stream contains the text "Baby! Arnold won't be back for new Terminator installment", its /Lang (es) attribute does not apply to it. The text is contained within the /P element that is a node of the structure tree (indicated by the /MCID attribute). So the language of “Baby! Arnold won’t be back for new Terminator installment” is determined by that Structure Element’s language specification, /Lang (en).

Accessibility Support in PDF

[New section to be added to Chapter 8 “Document Interchange”]

1 Introduction

Because of their potential richness of visual display, PDF files have been a difficult medium for many visually handicapped computer users. These users typically use screen readers to read a document aloud. Several features of PDF need to be supported to provide a complete and meaningful vocalization of a document.

1.1 Logical Content Order

To solve one of the problems encountered by disabled users, creators of PDF files can use logical structure and Tagged PDF to indicate logical content ordering in a PDF file. The content of a document will be presented to a disabled user by walking the structure tree and presenting the contents of each node. So all information in the document must be reachable from the structure tree.

1.2 Alternative descriptions

PDF 1.3 permits a human-readable text representation to be attached to a structural element or tagged content stream with the **Alt** attribute, which is a property of the **Span** tag. PDF documents can be enhanced by providing alternate descriptions for images, graphics, or other items that don’t translate naturally into text by providing **Alt** text.

Alternative Descriptions for Annotations

All annotation may contain a **Contents** attribute (Table 7.9 “Entries common to all annotations dictionaries,” in the PDF Reference, Second Edition). This should be used as the description for all annotations. It is the natural description for a Text annotation and it is a natural container for alternate descriptions of other, non-text-oriented annotations. The **TU** attribute of interactive form fields serves the same purpose.

Using Multiple Languages in Alternative Descriptions

Alt fields are of type Text (PDF Reference, Section 3.8.1, "Text Strings") which may be encoded using either PDFDocEncoding or Unicode. Unicode defines an escape sequence to indicate the language of the text. This mechanism permits the alternate description to change from the current language, as specified in the section on Tagged PDF.

1.3 Abbreviations and Acronyms

The full expansions of abbreviations and acronyms should be provided via the **BDC** tagged content operator. The property list for the tagged abbreviation should contain the **E** property. This will permit correct pronunciation of the abbreviation.

For example, in the text "Dr. Edward Smith", the word "Dr." would be tagged as an abbreviation for "Doctor", but in the text "123 Ridgemont Dr.", the word "Dr." would be tagged as an abbreviation for "Drive". Some abbreviations or acronyms are conventionally not expanded: "see it on CBS" should be left untagged (leaving the pronunciation to the mercy of the text-to-speech engine) or should have the expansion "C B S". Use the Unicode escape sequence for expansions in other than the current language. Table 1 shows the **E** key entry.

TABLE 1 Tagged content properties

KEY	TYPE	SEMANTICS
E	text	*(Optional)* The text for pronunciation of the tagged content stream. **E** is a property of the Span tag.

***Note:** The* **/E** *key is limited to use in* **/Span** *marked content.*

Box Definitions for Pages

[Applies to Section 8.6 “Prepress Support”]

1 Introduction

The PDF 1.3 specification allowed designers of PDF documents used in production printing workflows to specify more precise definitions of page areas. The **Page** object may include one or more of the following box definitions: **ArtBox**, **TrimBox**, **BleedBox**. The art box specifies the bounding area of artwork which can be placed into other documents. The trim and bleed boxes specify object positioning to accommodate for mechanical variation in physical printing systems. See Adobe Technical Note #5188, *PDF features to facilitate ANSI CGATS.12, PDF/X*, for a detailed description.

PDF 1.4 introduces a number of additions that allow the user or a prepress application to specify viewing preferences and attributes that enhance the viewing experience. Standard viewer implementations may simply ignore this additional information.

The preferences and attributes added for PDF 1.4 include:

- Additional Viewer Preferences that allow selection of a visible area distinct from a clipping region.
- Additional Page attributes additions (Section 3.6, “Page Objects”)
- A **BoxColorInfo** dictionary
- A **BoxStyle** dictionary, which describes the on-screen appearance of guidelines associated with a particular box.

2 Viewing and Printing preferences

These features provide the ability to select a preferred visible area distinct from a preferred contents clipping region. The user should be able to choose these settings separately for viewing and printing. It is desirable for these selections to be stored as preferences for individual documents.

The purpose of the Viewer Preferences, as the name suggests, is to record the user's preferences for alternative boxes to use for the page and clip areas. These preferences might be used when printing directly from an application, or when generating a Job Ticket. On the other hand, when a PDF file is printed from Acrobat or sent directly to a PDF printer, the **MediaBox** and **CropBox** will be used, just as before. There is no intention that PDF consumers in general will heed the Viewer Preferences.

Use of Guidelines

Following are some suggestions for how the newly defined boxes could be used in an application. Guidelines might be displayed in user-selectable colors both at "box" definition time and optionally when viewing the composite document. The ability to see such guides would be dependent on the graphical content of a given page. To be useful, the lines must be distinguishable from any artwork that they intersect. In order to prevent a need to repeatedly modify the guide colors, it is necessary to allow them to be stored in association with a page.

The choice of whether or not to display the guidelines would ideally be controlled by an individual user's preference for interacting with the document. As such, this information need not be stored in the document.

3 Additions to the Viewer Preferences Dictionary

The following additions to the viewer preferences dictionary (PDF Reference, Section 7.1; Table 7.1: "Entries in a viewer preferences dictionary") will support the viewing and printing preferences mentioned above:

TABLE 1 Additions to Table 7.1 Viewer Preferences

KEY	TYPE	SEMANTICS
ViewArea	name	*(Optional)* Specifies the name of the box which represents the area of the page the user would like to see on screen. The absence of a **ViewArea** box will result in use of the default as specified in Technical Note #5188, which states: If **ViewArea** is not specified, the **CropBox** will be used as the default. If the **CropBox** is also missing, the visible area will be defined by the **MediaBox**.
ViewClip	name	*(Optional)* Specifies the box which represents the contents-clipping area of the page the user would like to see on screen. The absence of a definition for the named box for any given page, will result in use of the default as specified in Technical Note #5188. In absence of this definition the clip region will be the **CropBox**.
PrintArea	name	*(Optional)* Specifies the box which represents the area of the page the user would like to see on printed output. The absence of a definition for the named box for any given page, will result in use of the default as specified in Technical Note #5188. In absence of this definition the printed area will be the **CropBox**.
PrintClip	name	*(Optional)* Specifies the box which represents the contents clipping area of the page the user would like to see on printed output. The absence of a definition for the named box for any given page, will result in use of the default as specified in Technical Note #5188. In absence of this definition the clip region for the printed content will be the **CropBox**.

4 Page Attributes additions

The following additions to the Page Attributes (PDF Reference, Section 3.6, Table 3.17: “Entries in a page object”) will support the user-selectable guideline colors as described above.

TABLE 2 Addition to Table 3.17 *Entries in a page object*

PARAMETER	TYPE	SEMANTICS
BoxColorInfo	dictionary	*(Optional; may not be inherited)* This dictionary specifies the user-selected guideline colors for

one or more of the box definitions. Where values are not present, the current application default settings will be used instead. The **BoxColorInfo** dictionary is described below.

5 BoxColorInfo Dictionary

BoxColorInfo is a dictionary that describes the colors to be used on screen when providing guidelines in a user interface which shows the bounds of various boxes. These attributes are intended merely as an adjunct to UI controls. If a user requests that the **TrimBox** guideline be shown, then the **TrimBox** attribute determines how it will appear. A **BoxColorInfo** dictionary has the attributes described in Table 3.

Absence of a page specific color record for a given box means that the current application default will be used instead. If a color record is defined, but no box definition exists, the information is simply ignored.

TABLE 3 BoxColorInfo Dictionary (New dictionary to follow Table 3.17 in PDFR)

PARAMETER	TYPE	SEMANTICS
CropBox	dictionary	*(Optional)* A dictionary which specifies several attributes related to the appearance of the guideline. In absence of a **CropBox** definition associated with the page object, this value will be ignored.
TrimBox	dictionary	*(Optional)* A dictionary which specifies several attributes related to the appearance of the guideline. In absence of a **TrimBox** definition associated with the page object, this value will be ignored.
BleedBox	dictionary	*(Optional)* A dictionary which specifies several attributes related to the appearance of the guideline. In absence of a **BleedBox** definition associated with the page object, this value will be ignored.
ArtBox	dictionary	*(Optional)* A dictionary which specifies several attributes related to the appearance of the guideline. In absence of a **ArtBox** definition associated with the page object, this value will be ignored.

Other box parameters may be defined in the future.

Each value in the **BoxColorInfo** dictionary is itself a dictionary that describes the on screen appearance of guidelines associated with a particular box. Table Table 4 describes the entries common to all dictionaries in the **BoxColorInfo** dictionary.

TABLE 4 Entries common to all Box Style Dictionaries

PARAMETER	TYPE	SEMANTICS
S (Style)	name	*(Optional)* One of the following names: **S** (Solid) The box draws as a solid line. This is the default. **D** (Dashed) The box is drawn as a dashed line. The dash pattern is specified by the **D** attribute. (See below) Other styles may be defined in the future.
C (Color)	array	*(Required)* An array of colorant values in **DeviceRGB** colorspace.
D (Dash array)	array	*(Optional)* If the value of the **S** entry is **D,** this array contains numbers representing *on* and *off* stroke lengths for drawing dashes, in the same format as the **d** marking operator. The default for this key is [3].
W (Width)	number	*(Optional)* The line width in points. The default is 1.

PrinterMark Annotations

[Applies to Section 8.6 “Prepress Support”]

1 Introduction

Printer’s marks are a set of graphic symbols and/or strings used by press operators to identify components of a multiple-plate job and maintain consistent output during the printing of a job. Targets are used to register the alignment of each plate. Color bars allow measurement of colors and ink density across multiple printing environments. Crop Marks show where the paper will be trimmed.

Such marks typically appear outside the area containing the graphical content of a page and are logically separate entity from those contents. Despite this, creation applications traditionally include such marks in the contents stream of the document. The **TrimBox** and **BleedBox** added to the page’s dictionary in PDF 1.3 provide a mechanism to logically separate the two regions.

PDF 1.4 includes a new annotation type, **PrinterMark**, to record, display, and print printer’s marks. Each page may contain multiple **PrinterMark** annotations, each of which contains a single symbol (for example, target, gray ramp, color bar).

Since printer’s marks typically fall outside the content clip and surround the page contents area, it is necessary to define each mark in a separate annotation. Otherwise, the default annotation handler, which is invoked for any user interaction that occurs within the bounds of an annotation, would control user interaction for the entire page. Implementing separate **PrinterMark** annotations facilitates an application’s ability to implement a drag-and-drop user interface for specifying such marks.

2 PrinterMark Annotation Attributes

Printer's marks are represented as Form XObjects which occur as entries in the Appearance Dictionary of **PrinterMark** annotations. **PrinterMark** annotations that are to be printed must occur in the **N** (Normal) sub-dictionary of the annotation **AP** dictionary. Entries may be provided for the **R** (Rollover) or **D** (Down) keys in the **AP** dictionary, but cannot be printed.

More than one appearance may be specified for a given Printer's Mark, based on different regional requirements or production facility requirements. Multiple sub-appearances in the **Appearance** dictionaries are the mechanism for specifying the alternatives. The sub-keys used inside the **N** (Normal), **R** (Rollover) and **D** (Down) dictionaries are arbitrary. The existing **AS** mechanism is used for selecting among them.

TABLE 1 Additions to Table 7.9 *Entries common to all annotation dictionaries*

KEY	TYPE	SEMANTICS
Subtype	name	*(Required)* Annotation subtype. Always **PrinterMark**.
MN (Mark Name)	name	(Optional) Name identifying the type of printer mark. For instance /RegistrationMark or /ColorBar.
F (Flags)	integer	*(Required)* The flag field must be present; bit values specified below: bit 1Invisible flag; must be 0 bit 2Hidden flag; must be 0 bit 3Print flag; must be 1 bit 4NoZoom flag; must be 0 bit 5NoRotate flag; must be 0 bit 6NoView flag; must be 0 bit 7ReadOnly flag; must be 1 bit 8ClickThrough; must be 0 (Note: this is a new bit intended to allow any annotation to always pass mouse clicks to the next lower level. A value of 0 indicates that the annotation handler will receive the clicks and is the standard behavior of the Acrobat product.)
AP (Appearance)	dictionary	*(Required)* Holds a stream containing the graphical operations required to draw a single **PrinterMark**.

AS (Appearance State)	name	*(Required if more than one appearance is present)* This value specifies which trap network is 'current' and will be displayed and printed.

2.1 PrinterMark Appearance Attributes

Each **PrinterMark** appearance contains the marking operations needed to render the desired graphic. The attributes are listed in Table 2.

TABLE 2 Additions to Table 7.12 *Entries in an appearance dictionary*

KEY	TYPE	SEMANTICS
MarkStyle	text	*(Optional)* Text which an application can use to describe a **PrinterMark** to users.
Colorants	dictionary	*(Optional)* Identifies the colorants which were identified in this mark. For instance a color bar that only includes CMYK vs. one with CMYK + Green. The contents of this dictionary are identical to the optional Colorants dictionary associated with a **DeviceN** colorspace.

New Metadata Architecture for PDF 1.4

[Applies to Chapter 8 "Document Interchange"]

1 Introduction

A means for defining Metadata is defined in PDF 1.3, using an **Info** dictionary to store the Metadata. PDF 1.4 introduces an alternate method that offers significant enhancements in how applications can access the data. The motivation for changing the way metadata is stored includes:

- PDF-based workflows often embed metadata-bearing artwork as components within larger documents. PDF 1.4 provides a standard way of preserving the metadata of these components for examination downstream. PDF-aware applications should be able to derive a list of all metadata-bearing document components from the PDF document itself.
- PDF documents are often made available on the World Wide Web, where many tools routinely examine, catalog, and classify documents. These tools should be able to understand the self-contained description of the document even if they do not understand PDF 1.4.

The additions for PDF 1.4 address the above needs.

2 Metadata streams

Metadata, both for the entire document and for components within a document, is stored in PDF streams having dictionary entries as given in Table 1.

The contents of a metadata stream are the metadata represented in XML. This information will be visible as plain text to non-PDF-aware tools only if the metadata stream is both unfiltered and unencrypted.

The format of the XML representing the metadata will be defined in a new document: *Adobe XAP Metadata Framework*. The XAP (eXtensible Authoring and Publishing) Metadata Framework provides a way to use XML to represent metadata describing documents and their component assets. XAP is intended to be adopted by a wider class of applications than just those that process PDF. XAP includes a method to embed XML data within non-XML data files in a platform-independent format that can be easily located and accessed by simple scanning rather than having to parse the document file. This document will be published by Adobe Systems in the general timeframe of the Acrobat 5.0 release.

TABLE 1 Metadata streams

KEY	TYPE	SEMANTICS
Type	name	*(required)* Must be **Metadata**.
Subtype	name	*(required)* Must be **XML**.

3 Document metadata

3.1 Attaching metadata to the document

There will be a new reserved key **Metadata** in the **Catalog** dictionary whose value is an indirect reference to a metadata stream as described in Section 2, “Metadata streams. The current **Info** dictionary will be preserved for backwards compatibility (see Section 5, “Compatibility).

TABLE 2 Additional key for Catalog dictionary

KEY	TYPE	SEMANTICS
Metadata	stream	*(Optional)* Contains the metadata for the PDF document. The metadata is represented as RDF in conformance with the XAP Adobe Standard Metadata schema. The stream must satisfy the

requirements of a metadata stream as given in Table 1.

4 Component metadata

4.1 How metadata is associated with components

A PDF document component represented as a dictionary or stream may have an additional key **Metadata**. The value associated with this key is an XML stream just as for the document-level metadata.

TABLE 3 Addition to any dictionary for components having metadata

KEY	TYPE	SEMANTICS
Metadata	stream	A metadata stream (as described in Table 1) containing the metadata for the component.

4.2 What components can have metadata

There are many kinds of objects represented by streams and dictionaries in PDF. Most, but not all, of these can have metadata attached as specified in Table 3.

In general, a PDF stream or dictionary may have metadata attached as long as the stream or dictionary represents an actual information resource, as opposed to serving as an implementation artifact. Table 4 enumerates the PDF constructs that are considered implementational (and hence cannot have associated metada-ta).

For the remaining PDF constructs there is sometimes ambiguity about exactly which dictionary or stream should bear the **Metadata** key. Such cases are to be resolved so that the metadata is attached as close as possible to the object that ac-tually stores the data resource described. For example, metadata describing a til-ing pattern should be attached to the pattern stream's dictionary, but a shading should have metadata attached to the shading dictionary itself, rather than to the shading pattern dictionary that refers to it.

This scheme for attaching metadata to component objects will not work for in-line images. If it necessary to attach metadata to an image, the image must be included as an image XObject.

TABLE 4 PDF constructs that do not take metadata

Nodes in name and number trees	The class map
Pages objects	The role map
The **Catalog** dictionary	File specification dictionaries
The **Names** dictionary	Font encodings
The trailer dictionary	Font dictionaries[1]
The **Info** dictionary	Font descriptors
The encryption dictionary	Color spaces[2]
Resources dictionaries	DeviceN dictionaries
Actions	SeparationInfo dictionaries
Destinations	Functions
The structure tree root	Extended graphics states
Marked content references	OPI dictionaries
Object references	**Group** dictionaries
Objects belonging to the various Web Capture databases	Property lists (when stored inline in content streams)

4.3 Finding component metadata from document metadata

The relationship between the document metadata and component metadata is maintained entirely within the metadata itself, and is therefore outside the scope of this specification.

1. By the principle that metadata should be attached to the closest object to the information being described, metadata for fonts should be attached to font file streams rather than to font dictionaries.

2. By the principle that metadata should be attached to the closest object to the information being described, metadata describing an ICC profile should be attached to the stream containing the ICC profile data.

4.4 Enumerating components with metadata

In order to find the actual component objects that have component metadata, an application must be PDF-aware. It is up to an application to decide which dictionaries and streams it will inspect for the presence of attached metadata.

5 Compatibility

PDF 1.3 documents read by PDF a 1.4-aware application

PDF 1.3 **Info** dictionaries remain syntactically valid in PDF 1.4. Any PDF 1.4-aware application should also understand PDF 1.3 **Info** dictionaries as a repository of metadata.

PDF 1.3 documents having metadata altered by PDF 1.4-aware application; PDF 1.4 documents read by PDF 1.3 application

In order to provide a measure of backward compatibility, applications that create PDF 1.4 documents should also write properties-value pairs having an equivalent in PDF 1.3 into the **Info** dictionary as well as into the **Metadata** stream.

PDF 1.4 documents read by PDF 1.4-aware application

Applications that support PDF 1.4 should check for the existence of a **Metadata** stream. If one is present, the application should ignore the **Info** dictionary.

Note: *Rationale: It might seem that the* ***Metadata*** *stream should override the* ***Info*** *dictionary on a property-by-property basis, but this is unnecessarily complicated. Any application that knows how to create a* ***Metadata*** *stream at all should make sure that it saves documents with the* ***Info*** *dictionary in sync with the* ***Metadata*** *stream. Imposing this condition allows applications that know to examine the* ***Metadata*** *stream to be confident that there's no different information to be found in the* ***Info*** *dictionary.*

PDF 1.4 document having metadata altered by PDF 1.3 application

An application that only understands PDF 1.3 that changes the metadata in a PDF 1.4 document would not know to write out a copy in the XAP stream. This means that a PDF file can have metadata in the **Info** dictionary that should override corresponding values in the XAP metadata stream.

It's not possible to modify existing PDF 1.3 applications, but it *is* possible to detect the situation in a PDF 1.4-aware application. The XAP property xap:MetadataDate records the last time the XAP metadata was modified. PDF 1.4-aware applications can check whether the modification date of a PDF document is more recent than the date given by xap:MetadataDate; if it is, the application can update the XAP metadata from the values in the **Info** dictionary.

Tagged PDF

[Applies mainly to Section 8.4.2 "Marked Content." Also Section 4 of this chapter will be added to Section 3.6.1,"Document Catalog."]

1 Introduction

Tagged PDF is a stylized use of PDF that allows reliable recovery of text, graphics, and images in PDF documents, with no ambiguity about the contents or the ordering of the contents. Tagged PDF is a firm basis on which to build tools for reflowing text; extraction of contents, extraction and conversion of PDF-to-HTML, PDF-to-RTF, and PDF-fragment-to-clipboard; access (in particular, PDF-to-speech); and other uses including full-text indexing of PDF documents, reliable search, table extraction, spell-checking, etc.

A Tagged PDF is page oriented. For each page of a Tagged PDF document, the rendering stream for that page contains the text, graphics, and images in PCOrder, as determined by the authoring application.

A Tagged PDF is a Logical Structured PDF. *Logical Structure* is used to carry information necessary to support tagging for access and content extraction, as well as styling properties needed for access, reflow so, and content extraction. It also provides the identification of the article flows in the cross-page environment for access and content extraction.

1.1 Terminology

The following terms are used in this section:

Alternative Text	Text, supplied outside the marking operators, that can "stand in" for a page element.
Artifact	A page element that is a side effect of rendering, rather than an intrinsic part of the document or story.
Block-level Structural Element (BLSE)	A Block-level Structural Element is a *Structural Element* that is laid out in the slow-progression direction (top-to-bottom in western text) as specified by the **WritingMode** property.
Character Code	A show string is the encoded representation of a sequence of non-negative integers. Each of those integers is a Character Code. The interpretation of a show string depends on the associated font: some fonts imply a one-byte representation while others imply a multi-byte representation. Note: This is the PDF definition of character code and somewhat contrary to normal usage. The same letter, for example a capital A, can have many different character codes on the same page.
Code-point	Each Unicode code-point is derived by mapping the Character Codes through the font's **ToUnicode** table. A single Character Code may map to a sequence of 1 or more Unicode Code-Points. This provides a uniform letter code for access and for content extraction.
Decoration Annotation	A PDF Annotation that is not part of the *page content*. Examples: text annotation; link annotation.
Illustration	Illustrations are fragments of a document that constitute single compact page-regions that are units-of-layout. They are rectangular and aligned parallel to the page-edges.
Inline-level Structural Element (ILSE)	An Inline-level Structural Element is a *Structural Element* that is laid out in the fast-progression direction (left-to-right in western text) as specified by the **WritingMode** property.
Logical Structure	A PDF facility that allows the structure of a PDF file to be expressed via a *Logical Structure*. Ref: PDF Reference, section 8.4.3. The *Logical Structure* is separable from Tagged PDF, although Tagged PDF requires the use of *Logical Structure*. Every Tagged PDF is a Logical Structured PDF, but not all Logical Structured PDF files are Tagged PDFs.
Page Content Annotation	A PDF Annotation that is "part of the page content." An annotation is page content, per section 7.4, if and only if it: a) Contains an AP appearance dictionary with a N normal

	appearance and b) Does not have its Hidden flag set. The remaining flags may be investigated by individual users. For example reflowing text requires the Print flag to be set in order to consider the annotation for reflow.
PCOrder	The ordering of a page's elements defined by the sequencing of the page-content stream imaging operations.
Reference-area	A reference-area is a virtual box (closed, non-self-crossing path that is typically but not always rectangular) used by the layout application as a reference frame or guide to place content. PDF does not record these boxes, but certain measurements used by reflow and by the content extraction properties are measured from these boxes (such as StartIndent and EndIndent). The *Reference-area*s of interest are generally the column box(es) in a general text layout, the outer bounding box of a table and each table-cell, and the bounding box of a figure or floating group.
Show String	The strings that are the arguments to the page-content **Tj** operators and the strings that are the arguments to the ' and " operators and the strings that are elements of the arrays that are the arguments to the **TJ** operators.
Soft Hyphen	A character that is used to mark conditional hyphenation points. Unicode and ISO-Latin-1 *code-point* 0xAD.
Structural Element	A node in the Logical Structure tree.
Unicode Standard	The set of sixteen-bit *code-points* defined by the Unicode Consortium, 57709 of which are currently assigned and name Unicode Characters. The lowest 65536 *code-points* in ISO 10646-1 1993 are identical to the Unicode Standard and are sometimes called the Basic Multilingual Plane. Ref: http://www.unicode.org
Unspecified	If a property has a default value of *Unspecified* it is treated identically to the situation where the property is not specified on the current element.

Definition of value types

<length>	A number, interpreted in default user-space units. HTML, CSS, and XSL allow the specification of measurement units [pt (computer points), pc, px, em (fraction of current font

	size), ex, cm, or %] on any length specifier. This feature is NOT supported in Tagged PDF.
<string>	A PDF string.

2 Tagged PDF Content Streams

2.1 Page Content and Page Artifacts

Artifacts

Within a page-content, we can call-out several forms of content:

1. The *real* content of the document. The *author's content*. The content that is identifiable in the author's markup.
2. Artifacts:
   - Artifacts of the printing process. Examples: crop-box markings; document file name printed outside the crop box.
   - Artifacts of the pagination of the document. Elements that would be absent (or present in a much different form) if a document was always one very big page. Examples: Running headers; page numbers.
   - Artifacts of the layout process and typographic style. Examples: a pastel colored area under the text columns; a horizontal rule above the footnote.

The differences between these forms are not absolute, and sometimes the line is blurred. Is a vertical rule between columns a pagination artifact or a layout artifact? If a footnote is numbered “3,” is that digit an artifact of the pagination or part of the real content?

Many consumers of Tagged PDF wish to distinguish between artifacts and the real content. A speaker of Tagged PDF probably should not speak running headers at page-turns. The reflowing of Tagged PDF should not reflow an inter-column rule if the reflow-result is single column.

Page artifacts are not part of a document's *Logical Structure*.

Some creators of Tagged PDF may wish to distinguish between the several types of Artifact. Some consumers of Tagged PDF will be only interested in "Artifact or non-artifact?" But the type of artifact, if it is clear, may aid other consumers.

An Artifact marked content may be used to distinguish artifacts.

/Artifact BMC .. EMC

/Artifact **PropertyList** BDC .. EMC

The first form is used to identify a generic artifact. The second form is when properties are attached to the Artifact.

Note: *It is recommended that the properties be attached to an Artifact whenever possible. They aid text reflow; reflow is likely to discard Artifacts that lack a* ***BBox****.*

TABLE 1 Artifact Property List

KEY	TYPE	SEMANTICS
Type	name	*(optional)* The type of page-content artifact. One of **Page**, **Pagination**, **Layout**.
BBox	rectangle	*(optional)* The page-extent of the Artifact.
Attached	array	*(optional)* Array containing one to four names. The edges of the page (as defined by the **CropBox**), if any, that the Pagination Artifact is "attached to." The edges are named **Top**, **Bottom**, **Left**, and **Right**. The ordering of the names in the array is not material. If a Pagination Artifact is logically "attached to" an edge, that characteristic can be captured with this key-value. Multiple edge-attachments may be indicated. If both **Left** and **Right** (or **Top** and **Bottom**) are supplied, that indicates a full-width (full-height) Artifact. No meaning for Attached is defined for **Page** or **Layout** Artifacts. (It is only honored for artifacts of type **Pagination**.)

2.2 Examining and Processing the Page Content

A consumer of Tagged PDF may have its own ideas about what constitutes the *actual* page-content. (Some of this is discussed, above, under Artifacts)

The usage of Tagged PDF supports this. In general, the consumer is allowed to:

a) decide that some page-content is not of interest, and

b) treat some page elements as *terminals* that are not examined further, for example, treat an illustration as a unit for reflow purposes, and

c) substitute *Alternate Text*. Different consumers will make different decisions, as their goals are different.

Throughout this specification, such usage is supported. Phrases that talk about "processing the page-content" always condone such processing.

2.3 Page Content Ordering

Within a page-content, the serialization of the elements imposed by the page-content stream defines an ordering of the elements. That is the Page Content Ordering (*PCOrder*). And the ordering of the glyphs within each show-string defines an ordering within show-string elements.

This ordering is separate from the ordering of the elements of the page imposed by a depth-first traversal of the *Logical Structure*. It may or may not be a different ordering.

Note: *If the page contains Artifacts, the artifacts will be present in the PCOrder but artifacts are always absent from the Logical Structure order.*

When dealing with material on a page-by-page basis, some consumers of Tagged PDF will use the *PCOrder* of a page as reading order, rather than the Logical Structure ordering of a page's elements. The requirement for *PCOrder* is for reflow to maintain a proper reading order, so it should normally be a top-to-bottom order, with artifacts in their correct relative place.

The creator of Tagged PDF is responsible for producing an appropriate *PCOrder* of each page's content.

Note: *Both PCOrder and Logical Structure order sequence both the full page-content and any fraction of it chosen by a consumer of* Tagged *PDF.*

Annotations

In PDF, annotations are not interleaved within the page-content stream, but are placed in the **Annots** array for the page.

Logical Structure provides a mechanism for referencing the annotation from within the logical structure tree, even though it is not part of the content marking stream; see section 8.4.3, “Logical Structure,” of the PDF Reference, and page 494, “PDF Objects as Content Items.”

In order to make the location of an *Page Content Annotation* that is in the *Logical Structure* manifest within the Page Content Ordering, one applies this rule:

> For each *Structural Element* that is manifest in the page-content: The set (if any) of Page Content Annotations (on this page) that are immediate predecessor Elements in the *Logical Structure* precede this *Structural Element* in the Page Content Ordering. The set (if any) of Page Content Annotations (on this page) that are immediate successor Elements in the *Logical Structure* follow this *Structural Element* in the Page Content Ordering.
>
> (Rephrasing this mechanistically: To process the page in Page Content Ordering, process the manifest *Structural Elements* in page-content order, and, a) immediately before processing each page-content manifest *Structural Element*, process (in *Structural Element* order) predecessor leaf Elements in the *Logical Structure* as long as they are unprocessed Page Content Annotations on this page; b) immediately after processing each page-content manifest *Structural Element*, process successor leaf Elements in the *Logical Structure* as long as they are unprocessed Page Content Annotations on this page.)

Note: *A creator may introduce an empty MCID in order to have this rule order the page-content.*

Reverse Order Show String Content

Font characteristics may suggest that right-to-left text be typeset left-to-right. This is an artifact of the historic use of Type 1 and other fonts. In a font that contains Hebrew letters, for example, one might expect that the character-origins are at the right side of the glyphs and that the advance, the delta-x of the origin, is negative.

Although fonts can be created that way, usually such fonts are constructed just like an English font, with character-origins at the left and positive advance. That means that to typeset a Hebrew word in reading order using such a positive-advance font, the PDF page content must supply an explicit re-origin for every letter. This is unfortunate: a) it makes the file larger b) if the advances are utilized within a show-string, then the calculations at the rendering level can be performed in device space; but if they are instead performed by the creating application in page-space, the as-rendered inter-character spacing is less precise.

Hence, individual show-strings are allowed without interior embedded spaces to be typeset left-to-right (positive advance) even though the reading order within the show-string is thus backward.

Note: *The “without embedded spaces” is not a major limitation, because a space provides an opportunity to re-align the typography without visible effect; and it is valuable because it limits the scope of reversals for word-processing consumers of Tagged PDF.*

The **ReversedChars** marked content informs the consumer of Tagged PDF that the show string or show strings within the marked content are individually reversed in reading order.

```
/ReversedChars BMC (olleH) Tj -200 0 Td (.dlrow) Tj EMC
```

Note: *Within a multi-show-string marked content, only the individual letters within a string are reversed; the ordering of the show-strings themselves is in reading order, as in the example.*

2.4 The Text Stream at the Character Level

Within each page of a Tagged PDF, there is a sequence of show strings and therefore a sequence of *Character Codes* with associated fonts.

For each page of a Tagged PDF document, the sequence of *Character Codes* can be unambiguously turned into a sequence of Unicode values appropriate for “cut and paste” operations.

Unicode

In Tagged PDF, every *Character Code* can be mapped to a *Code-point* in the Unicode Standard. The Unicode Consortium has defined sixteen-bit values for most of the characters in the world's languages and world's character sets, and provides a *vendor space* that can be considered a *private use* area for escaping purposes.

The mapping function from a *Character Code* to a Unicode value is the following, as derived from section 5.9, *ToUnicode CMaps*, of the PDF Reference.

Note: *All objects of PDF-type NAME (items having the syntax: /xxx) within the structure tree or its associated attribute dictionaries (whether used as a key or as a value), RoleMaps and ClassMaps (used by structure or any structure attribute) must be in UTF-8 encoding.*

Obtaining a Unicode Value from a Character Code and Font

If the font contains a **ToUnicode** entry, then the *Character Code* should be converted to Unicode via the **ToUnicode** CMap.

Else if the font uses one of the PDF predefined encodings MacRomanEncoding, MacExpertEncoding, or WinAnsiEncoding (perhaps as modified by a **Differences** array in the fonts encoding resource), use the **Differences** array or Appendix D to convert the *Character Code* to an Adobe glyph name (called "NAME" in Appendix D of the PDF Reference, Second Edition). Then use the Adobe glyph name and look up the corresponding Unicode value.

Else if the font uses one of the predefined CMaps listed in Table 5.14 on page 320 of the PDF Reference, Second Edition, except **Identity-H** and **Identity-V**, convert the *Character Code* to a Unicode value via the following steps.

1. Obtain the **Registry** and **Ordering** of the predefined CMap from the **CIDSystemInfo** of the appropriate CMap.
2. Concatenate the **Registry** and the **Ordering** according to the format "<registry>-<ordering>-UCS2" to obtain a second CMap name, for example, "Adobe-Japan1-UCS2." Obtain that CMap.
3. Index into the predefined CMap, using the *Character Code*, and obtain an Intermediate Value.
4. Index into the CMap obtained in step 2, using the Intermediate Value, and

obtain a Unicode Value.

If any of these four steps fail, for example if there is no CMap of that name or the indexing value is missing or undefined in the CMap, then failure: the page is not a Tagged PDF page.

Else if the font is a Type 0 font whose descendant CIDFont uses the Adobe-Japan, Adobe-Korea, Adobe-CNS1, or Adobe-GB1 character collection, as specified in the **CIDSystemInfo** dictionary, follow the same steps as in the preceding else clause.

Else if the font is a Type 1 font whose character names are taken from the Adobe standard Latin character set and the set of named characters in the Symbol font, documented in Appendix D of the PDF Reference, Second Edition, use the corresponding Unicode value found by looking up the glyph name.

Else failure: the page is not a Tagged PDF page.

Note: *An ActualText or Alt property on a Structural Element may affect the character-stream that some consumers of* Tagged *PDF actually utilize. (For example, some consumers of* Tagged *PDF may choose to utilize the ActualText or Alt content and ignore all other content in the Structural Element and its children.)*

Note: *Several uses of* Tagged *PDF require characters that may not be available in all fonts. Appendix A of this document shows how to add additional Unicode code-points to the font-descriptor.*

Font Characteristics

In Tagged PDF, every Character Code from the text stream can be given the key-values:

Serifed	(boolean)
Proportional	(boolean)
Italic	(boolean)
Smallcap	(boolean)
Weight	(non-negative integer, per http://www.w3.org/TR/xsl#font-style)

and these key-values depend solely on the font used, not on the Character Code value.

These key-values are useful when re-purposing a Tagged PDF page and the target system has a limited repertoire of available fonts.

Note: *These key-values only exist for characters contained in the* Tagged *PDF show strings. They do not exist for ActualText replacement text nor for Alt descriptive text.*

In a Tagged PDF, the above key-values can correctly be calculated from the information in the PDF Font Descriptor for the font in use. For the base fourteen PDF fonts, the Font Descriptor may be missing; the well-known values for these fourteen fonts are used:

The *Serifed* value is equal to the value of the “Serif font” bit in the **Flags** of the Font Descriptor.

The *Proportional* value is the negation of the value of the “Fixed-width font” bit in the **Flags** of the Font Descriptor.

The *Italic* value is equal to the value of the “Italic” bit in the **Flags** of the Font Descriptor.

The *Smallcap* value is equal to the value of the “Small-cap font” bit in the **Flags** of the Font Descriptor.

The *Weight* value is stored in a PDF file via the StemV value of the Font Descriptor. To obtain the Weight from the StemV, calculate it as

SQRT (MAX(StemV,50) – 50) * 65

and, if desired, adjust the result to be in the CSS2 and XSL range of [100 .. 900].

Note: *The Weight (and StemV) value is for font-substitution, where the actual “weight” of the glyphs is the important thing. Not all fonts with “Bold” in their names have similar weights: Bell Gothic Bold is lighter than Stone Sans [Regular].*

***Note:** The Base 14 fonts may be present in a PDF without an associated Font Descriptor. However, if it is necessary to add Unicode code-points to the font, it will be necessary to provide a font descriptor in accordance with the guidelines in Appendix A in this document.*

2.5 The Text Stream at the Word Level

The text stream not only defines the characters in the running text of a page, it also defines the words.

Unlike character, the definition of word is fuzzy, and depends on the intent of the application that is examining the Tagged PDF.

***Note:** The identification of what constitutes a word is totally unrelated to how the text happens to be grouped into show strings. It is a common error for people to believe that the division into show strings has semantic significance. In particular, a space or other word-breaking character is still needed even if a word break happens to fall at the end of a show string.*

Reflow tools need to know where it can re-break the running text when it reformats; access wants to vocalize the words; spell-check and other applications all have their own idea of what “Word” means.

It is not important that we create a single definition of word that satisfies all these clients; what is important is that there is enough correct information in the stream so that each client can easily obtain the information that it needs. For example, reflow should consider that there is a re-break opportunity at the end of an em-dash that separates two letter-sequences; access software may wish punctuation to be separated from words; given a Unicode stream they can work this out on their own.

A consumer of Tagged PDF finds words by (sequentially) examining the Unicode character stream, perhaps augmented by ActualText. It does not utilize a) glyph positioning on the page b) font information or c) glyph sizes.

The main consideration is ensuring that the spacing characters that would be present in any markup (for example RTF) are present in the Tagged PDF. Some applications may discover “words” by simply separating words at every space character. Or they may utilize slightly more sophisticated algorithms, for example, treating an em dash as a word-separator.

Other applications, such as reflow, may be interested in ascertaining possible line-break opportunities; they may use an algorithm similar to the one at:

< http://www.unicode.org/unicode/reports/tr14/ >

Note: *The above specification implies that the lines of text for western languages usually end with a trailing space.*

2.6 Alternative Texts and Other Textual Key-Values

Since Tagged PDF enables many consumers, including accessibility, to recover the contents of PDF pages, a Tagged PDF file should use the alternative text (**Alt**), actual text (**ActualText**), abbreviation expansion text (**E**), and language tagging (**Lang**) facilities described in this document.

The guidelines at:

< *http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505* >

and the documents that it points to should be used to understand the support that is required.

Expansion Text Instream Marking

The page-content marking stream may contain an abbreviation that is customarily spoken in its full form.

Sometimes, a reader-application can divine the full pronunciation without aid: appeal to a dictionary will probably reveal that “Blvd” is pronounced “boulevard” and “Ave.” is pronounced “avenue.”

But some abbreviations are difficult for the Tagged PDF consumer to resolve: Consider “St. Peter lives at 123 Main St.” An authoring application can place an Abbreviation Expansion alternative text into the Tagged PDF in order to provide a “long form” for an abbreviation. Abbreviation Expansions are defined in the Accessibility section in this document.

They are marked with an instream marking of the form:

```
/Span << /E (replacement string) >> BDC (abbrev.) Tj EMC
```

Note: *The Expansion Text (E) property is treated as if it were a word or phrase substitution for the BDC...EMC content. Thus if 2 objects each having an Expansion Text (E) property are encountered in sequence, they should be treated as if a word break were present between them.*

Language Tagging

If the authoring application knows the language of the running text, it is strongly encouraged to include this information in the PDF document. Knowledge of the language aids many clients of the Tagged PDF, including clients interested in pronunciation, spell-checking, and hyphenation in re-layout.

Language tagging is described in the *Language* section of this document.

The language may be marked with an instream marking of the form:

```
/Span << /Lang (en-us) >> BDC (Text to be interpreted as US. English.) Tj EMC
```

Note that the expansion text and language tagging can be combined:

```
/Span << /Lang (en-us) /E (replacement) >> BDC (abbrev.) Tj EMC
```

Note: *ISO 639 language codes can be found at*
< http://lcweb.loc.gov/standards/iso639-2 >
and IANA registered language codes can be found at
< http://www.isi.edu/in-notes/iana/assignments/languages >.

2.7 Textual Artifacts

There are elements and relationships in a page's running text that are not really part of the articles in the document, but rather artifacts associated with turning the articles into a document. Since these elements would not be present in an SGML or other purely-markup representation of the document, they hence do not appear as elements in the Logical Structure of a Tagged PDF document.

Hyphenation

Among the artifacts introduced by typesetting is the hyphen at the end of a line signifying that a line-break was introduced into the middle of a word.

Such a soft hyphen is identified by a character that the **ToUnicode** mapping translates to the Unicode value of U+00AD, 173 decimal.

A “regular” or “hard” hyphen is U+002D. In order to differentiate between “soft” and “hard” hyphens, the creator of a Tagged PDF chooses a character that the **ToUnicode** mapping translates to U+002D or U+00AD appropriately. The consumer of Tagged PDF does not have to guess what those characters are.

Word continuations

The running-text of a page may contain locations where the normal progression of article text suffers a discontinuity. An example is a page that contains the beginnings of two different stories, each story being continued onto later pages in the document.

A consumer of Tagged PDF needs to be able to recognize this case, so that, for example, the last phrase of the first story fragment is not part of the first sentence of the second story. Examining the *Logical Structure* is the way to recognize this case.

Text Clipping and Other Hidden Page Elements

The text of a document, present in the page-content, may be invisible for many reasons. It may be clipped, its ink-color may match the background, it may be over-printed.

In Tagged PDF, the text is all of the text without regard for any such display anomalies.

Similarly, illustrations on the page may overlap, one obscuring the other. For the purposes of Tagged PDF, each Illustration is complete, and may include elements that are not visible on the original page.

2.8 XObject Forms

An XObject Form participates in the *Logical Structure* and Tagged PDF of the page that it is imaged into via the **Do** operator.

For all of the purposes of Tagged PDF, it is as though the form's content stream was bodily inserted into the page, replacing the **Do**. (Actually performing such a replacement involves fix-ups of the Graphics State and of Resource identifiers.)

In *Logical Structure* (and therefore in Tagged PDF), no XObject Form within a document can be incorporated into the document's pages more than once unless the XObject Form contains no *Logical Structure* markings at all.

Note: *A multiply-referenced XObject Form can participate in Logical Structure, just as a multiply-referenced XObject Image can, by being all of the content of a leaf element or part of the content of a leaf element. But the Form's content-stream cannot itself contain any Logical Structure BDC, BMC, or EMC markers.*

3 Use of *Logical Structure* in Tagged PDF

Logical Structure is used in Tagged PDF to carry the structure and properties needed for access, content extraction, and reflow.

The *Logical Structure* is also used to resolve reading-order when document components cross page boundaries.

3.1 Standard Roles and Standard Properties

This section describes the standard roles. These roles are necessary so that reflow can determine a standard formatting paradigm for a given object; access can provide predictable presentation on alternate media; and content extraction can provide for standard translations to various common file formats (HTML, HTML with basic CSS, XML with basic CSS, RTF, etc.)

The user may either:

- Specify the role as the value of the subtype (**S**) key of the *Structural Element*, or
- Specify an arbitrary value as the subtype (**S**) then provide a RoleMap entry which maps the arbitrary subtype (**S**) to one of these roles.

Use of the Standard Roles in Tagged PDF

Use of the *Structural Element's Subtype (S) and the Logical Structure's RoleMap*

The *Structural Element*'s subtype (**S**) value will be used as the *Structural Element*'s tag for any XML output and possibly as the stylesheet-name for RTF output. In most other cases, the mapped role is used to determine the tag to be issued.

Note: *For proper processing by the various content extraction and access tools, it is strongly recommended that the specified subtype (***S***) or the mapped role be one of the "Standard" roles described later in this section.*

The **rolemap** is not required if all subtype (**S**) values are members of the "Standard" roles described in this document. However, if the subtype (**S**) value is not one of the Standard Roles, the **Role** is derived by taking the subtype (**S**) value and remapping it via the **RoleMap**. In the absence of a **RoleMap**, the subtype (**S**) value will be used.

Top Structural Element of the Logical Structure Tree

For most content extraction formats, the document must be a tree with a single root node. Therefore, the **StructRoot** node may have only one child in its kids (**K**) array. If the PDF file is a complete document, the use of the **Document** role is recommended. If the PDF file is a well-formed document fragment, then **Part**, **Art**, **Sect**, or **Div** may be used as the top node.

TABLE 2 Additional Role for the Top Structural Element of the Logical Structure

STANDARD ROLE	DESCRIPTION
Document	This is the root element of any structure tree that contains multiple parts or multiple articles. (Comparable to <BODY> in HTML.)

Block-level Grouping Elements

TABLE 3 Roles that provide grouping of Block-level Structural Elements

STANDARD ROLE	DESCRIPTION
Part	A Part is a large division of an entire document. Parts are appropriate for grouping Articles or for grouping sections. (Comparable to <DIV Class="Part"> in HTML.)
Art	An Article is a relatively self-contained body of text considered to be a single narrative or exposition. An Article should not contain any other Article, so that applications that examine *Logical Structure* may assume that Articles are disjoint. (Comparable to <DIV Class="Art(icle)"> in HTML.)
Sect	A Section is the most general text container type, comparable to Division <DIV Class="Sect"> in HTML.
Div	A Division is a generic block-level element or a group of block-level elements.
BlockQuote	A block quote is a block-level element containing one or more paragraphs of text attributed to someone other than the author of the text surrounding it.
Caption	A Caption is a brief portion of text that describes a table or figure. It is different from a **Lbl** (Label) in that it describes rather than merely identifying. Elements that have a child of role Caption may also have a child of role **Lbl**.
Index	An **Index** contains a sequence of entries that contain both identifying text and Reference elements that point out the occurrence of the text in the main text body of a document.
TOC	A Table of Contents element contains a **List** of **ListItems** (see Table 5), each of which may contain either a **ContentItem** or another such List along with a Reference to a text body (or other) *Structural Element*. A **TOC** is thus potentially hierarchical. It is desirable for such a hierarchy to correspond in structure to the structure hierarchy of the text body of the document.

	Lists of figures and tables, as well as bibliographies, can be treated as tables of contents for the purpose of standard *Structural Element* types.
TOCI	(ContentItem) A **TOCI** contains some identifying text, along with one or more References to a text body *Structural Element*. (The Reference entries can be used to show the **Label**, **Title**, and/or page-number of the item in the text body.)

Please see the section(s) describing the properties owned by **Layout** to determine the specific usage of properties by **Div**, **Part**, **Art**, **Sect**, **BlockQuote**, **Caption**, **Index**, **TOC**, and **TOCI**.

Paragraph-like Block-level Structural Elements

TABLE 4 Paragraph-like Block-level Structural Elements

STANDARD ROLE	DESCRIPTION
H	(Heading) A Heading is a label for a subdivision of a document's content. ***Note:*** **A** *Heading should appear as the first child of the division that it heads.*
H1 through **H6**	(Heading with specific level) A Heading is a label for a subdivision of a document's content. Headings with specific levels are used in applications that can't hierarchically nest their sections, thus can't determine the heading level from the level of nesting.
P	(Paragraph) A Paragraph is a low-level division of text. (A preliminary version draft of this document required that **P** elements should not contain any other **P** element. This requirement has been relaxed.)

Please see the section(s) describing the properties owned by **Layout** to determine the specific usage of properties by **P**, **H**, **H1**, **H2**, **H3**, **H4**, **H5**, **H6**.

Block-level Structural Elements

As used herein, a *BLSE* is any region of text that is laid out paragraph-like: heads, list-items, footnotes are all *BLSE*s.

All *BLSE*s are elements of the document's *Logical Structure*. A *Structural Element* is a *BLSE* if (and only if) it is an *Structural Element* role-mapped to one of { **P**, **H**, **ListItem**, **Caption**, **Document**, **Div**, **Part**, **Art**, **Sect**, **BlockQuote**, **Caption**, **Index**, **TOC**, **TOCI**, **H1**, **H2**, **H3**, **H4**, **H5**, **H6**, **L**, **LI**, **Lbl**, **LBody**}. All other elements are treated as inline-level elements, except for:

a.) **TR**, **TH**, & **TD** which have Table-specific layout strategies and are considered neither ILSE nor BLSE.

b.) Elements with a **Placement** property, where the property value overrides the default placement of Inline.

The document's Logical Structure completely describes the relationship between *BLSE*s and page-content. In Tagged PDF, all of a page's running text is within *BLSE*s. There is no text layout outside of *BLSE*s.

Note: *Text may exist outside of BLSEs. It is either an Artifact or it is a design Structural Element in an Illustration.*

Fragmented BLSEs

In many cases, a *BLSE* appears as one compact piece of page-content. In other cases, a *BLSE* is discontinuous in (one or more) page-contents. Examples: a *BLSE* that extends across a page boundary; a *BLSE* that is interrupted (in the *PCOrder*) by a nested *BLSE* or directly-included footnote.

Note: *Nothing in* Tagged *PDF suggests that the physical placement of a footnote at the bottom of a page implies a late location in the PCOrder, nor does it preclude placing it later in PCOrder.*

Appeal to the *Logical Structure* is the way to recognize fragmented *BLSE*s and reassemble them into a single *BLSE* when necessary.

Baseline Alignment Within BLSEs

The typesetting of the *BLSE*'s text on the Tagged PDF page may be irregular. This may be done to create a special typographic effect, for example text that "bounces" on an irregular baseline, or that has a curved baseline.

For reflow and content extraction, text will ordinarily be re-typeset according to the ordinary **Layout** of the paragraph. And speech programs will ignore the special typographic effect variation.

However, reflow and content extraction should not lose the visual appearance of subscripting and superscripting. And speech programs should be able to distinguish subscripting and superscripting. The **BaselineShift** property is used to record superscript/subscript deviations from the defined baseline or reference path.

Most authoring applications that create special typographic effect text use explicit operations on the graphic matrix (and/or XY-moves with adjustments to the text matrix) on a character-by-character basis to place each character within the special typographic effect text. The application's equivalent to the **BaselineShift** property is used to record superscript/subscript deviations from the defined reference path. Tagged PDF capitalizes on this by ignoring positioning adjustments (since we now require characters that the **ToUnicode** mapping translates to spaces to be retained in the text stream and other breaks to be explicitly marked) and using **BaselineShift** to identify superiors and inferiors.

Special Block-level Structural Elements for Structuring Lists

TABLE 5 Special Block-level Structural Elements for Structuring Lists

STANDARD ROLE	DESCRIPTION
L	(List) A List can be any sequence of items of like meaning and importance. ***Note:*** *The immediate children of a List element should be ListItem elements.*
LI	(List Item) A List Item is one member of a List. ***Note:*** *List Items may contain an* ***Lbl*** *and an* ***LBody****.*

LBL	(Label) A Label is a name or number that distinguishes an element from others in the same list or other group of like items. (Contains the term in dictionary lists.)
LBody	(LBody) A wrapper surrounding the list item's descriptive content. (Contains the definition in dictionary lists.) Contains either inline text or may contain multiple block-level elements (including nested lists).

Please see the section(s) describing the properties owned by **Layout** and **List** to determine the specific usage of properties by **List** components.

Special Structural Elements for Structuring Tables

TABLE 6 Special Structural Elements for Structuring Tables

STANDARD ROLE	DESCRIPTION
Table	A Table represents a two-dimensional layout of rectangular data cells, possibly having a complex substructure. A Table contains **TR**s (table rows) as children, and may have a **Caption** as its first or its last child. The substructure of *Structural Element* types that may be included within a **Table** is designed to mimic the highly prevalent structure of HTML tables. A Table may be placed as specified by the Placement property.
TR	(Table Row) A Table Row is one row of headings or data in a table. A table row may contain TableHeaderCells and TableDataCells.
TH	(Table Header Cell) A Table Header Cell is a table cell that contains header text describing one or more rows or columns of a table.
TD	(Table Data Cell) A Table Data Cell is a table cell that it is intended to contain (non-header) data. The association of headers with rows and columns of data is to be determined heuristically by applications and is not defined here.

Please see the section(s) describing the properties owned by **Layout** to determine the specific usage of properties by **Table** and **TR**.

Please see the section(s) describing the properties owned by **Layout** and **Table** to determine the specific usage of properties by **TD** and **TH**.

Inline-level Structural Elements

An *Inline-level Structural element* is used to identify a span of text having specific styling or specific behaviors. They are handled in exactly the same manner as all other *Structural Elements* and *Structural Element* properties.

TABLE 7 Inline-level Structural Elements

STANDARD ROLE	DESCRIPTION
Span	A **Span** is any inline segment of text. One use of a **Span** is to delimit the text associated with a given set of styling properties. ***Note:*** *Not all inline style changes need to be marked as a Span. It is not necessary to mark text color and font changes (including bold, italic, and small-caps modifiers) since these can be derived from the PDF content. However, it is necessary to use a Span to mark TextDecorationType, LineHeight, or BaselineShift or to mark any of these 3 properties on the BLSE that directly contains the text. Certain other properties, such as Lang and E may be marked using the instream span marking "/Span <</Lang (en-us)>> BDC ... EMC".*
Quote	A **Quote** is an inline portion of text attributed to someone other than the author of the text surrounding it. (This is a quote that is inline within a paragraph, whereas **BlockQuote** is a quote which is a whole paragraph or multiple paragraphs in length.)
BibEntry	A **BibEntry** gives information on where some cited information can be found. A **BibEntry** may contain a **Lbl** as a child *Structural Element*. It is likely that a **BibEntry** will have substructure identifying author, work, publisher, and so forth; but standard *Structural Element* types are not provided at this level of detail.

Note: *Inline-elements have in common that, although they form part of a text sequence, they are not themselves BLSEs. They may, however, be contained in or contain BLSEs.*

Please see the section(s) describing the properties owned by **Layout** to determine the specific usage of properties by the common Inline components.

Special Inline-level Structural Elements

TABLE 8 Special Inline-level Structural Elements

STANDARD ROLE	DESCRIPTION
Code	**Code** represents part of computer program text embedded within a document.
Note	A **Note** is some explanatory text, such as a footnote or an endnote, that is referred to from body text. A **Note** may have a **Lbl** as a child *Structural Element*. A **Note** may be included as a child in the body text *Structural Element* that refers to it, or may be included elsewhere (such as in an endnotes section) and accessed via a **Reference**.
Form	A **Form** *Structural Element* designates an *Inline-level Structural Element* as a PDF Form Annotation—something that can be or has been filled out. A Form *Structural Element* refers to its underlying annotation via an object reference (OBJR). A form element does not have any children other than the OBJR, its rendering is defined by the appearance entry of the form annotation.
Link	A hypertext (go to) link. (A discussion of Link attributes will be supplied in a future revision.) In a manner similar to the Form Structural Element, the Link Structural Element designates an Inline-level Structural Element as associated with a PDF Link Annotation—something that goes to another location in this document or goes to another document. A Link Structural Element refers to its underlying annotation via an object reference (OBJR) in the Structural Element's Kids array. In the case of the link annotation, it may have children in addition to the OBJR which represent the displayed form of the link. In the absence of any child Structural Element, the text content of the Link Structural Element is treated as if it were a Span element. (See section below on Simple Links.)
Reference	A **Reference** element contains a citation to data elsewhere in the document. (A discussion of **Reference** attributes will be supplied in a future revision.)
Formula	A **Formula** is a mathematical formula.

	Note: *This is only useful for navigation: the encoding would need to be represented in order to be able to reuse a formula. From a formatting viewpoint, it is handled in a manner similar to a Figure element.*
Figure	A Figure is graphic material associated with text which may be placed as specified by the **Placement** property.

Please see the section(s) describing the properties owned by **Layout** to determine the specific usage of properties by all Special Inline components.

Simple Links

It is important to make link annotations that are comparable to HTML links accessible in Tagged PDF, since links are a particularly important navigation aide to the blind or motion disabled. This section describes how to incorporate link annotations in the structure tree to make it possible to provide this level of accessibility.

What are simple links?

A simple link is a **Link** *Structural Element* in the logical structure tree that has an action associated with it. There is only one action associated with this *Structural Element*, and the action is associated with all of the content of the *Structural Element*.

Examples of simple links are links that are associated with a span of text or with an image.

Creating a simple link

A Structural Element with the subtype (S key) of Link is inserted in the structure tree to mark the location in the tree to be associated with a PDF link annotation. This Structural Element is treated as a special form of an ILSE and may contain any content allowed in an ILSE except another link element. Rather than associating styling with the content of the element, a link Structural Element associates the action of the link annotation.

A Link *Structural Element* in the logical structure tree is a simple link when it has one or more children that are OBJRs for Link Annotations (PDF 1.3, section 7.4.5). All of the content of the Link *Structural Element* must be covered by one of the child Link Annotations. A simple link may contain several Link Annotations because the geometry of the object requires it; for instance, if a span of text wraps from the end of one line to the beginning of another, separate Link Annotations may be required to cover the portions of the simple link on different lines. If a simple link contains multiple Link Annotation children, they must all have the same action and target.

When an OBJR to a Link Annotation appears in the logical structure tree as a child of a Link structure element, that Link Annotation is associated with the content of that Link *Structural Element*. The Link Annotation rectangle may cover a subset of the content, or it may extend beyond the ranges of the content, but it should not overlap visible content in any other part of the logical structure tree.

If a Link Annotation in the structure tree is not a child of a Link *Structural Element*, it may not be properly processed by Accessibility and content extraction services.

Simple links may not be nested, that is, one simple link may not be a descendant of another simple link.

How does Accessibility use simple links?

When the MSAA plug-in encounters a link annotation in a document as it is passing it to the screen reader, it needs to provide certain attributes. It must identify it as a link, it must activate the link if requested, and it must describe what the link will do if activated. The plug-in can obtain this information from the link annotation.

It must also provide the name of the link, used to identify the link to the user, and it must provide a value that is used to determine when two links are really the same. The link annotation can't provide this information directly.

If a link annotation is part of a simple link, the link name can be generated from the content of the Link. If the content is text, that text can be used as the name. If the content or the Link itself has an Alt tag, that can be used as the name. In either case, the name is likely to be meaningful to the user as he reads the document or as he tabs to the next link and needs to know where he is.

If two link annotations are siblings within a simple link, they will be given the same value so the screen reader will only present the link to the user once, as matches the logical structure.

Examples of simple links

Example 1

Consider the following snippet of html which produces a line of text containing a link:

```
<html>
<body>
<p>
Here is some text <a href=http://www.adobe.com>with a link </a>
inside.
</body>
</html>
```

If we represented the equivalent in PDF and represent the link as a simple link, the content stream might contain

```
/P <</MCID 0 >>BDC
q
0 18 612 756 re
W* n
BT
/T1_0 1 Tf
14 0 0 14 10 753.9756 Tm
(Here is some text )Tj
ET
EMC
/Link <</MCID 1 >>BDC
0 0 1 RG                          // blue
0.7056 w 10 M 0 j 0 J []0 d       // line width, etc.
111.094 751.8587 m                // move to beginning of underline
174.486 751.8587 l                // lineto end
S                                 // stroke
0 0 1 rg
BT
/T1_0 1 Tf
14 0 0 14 111.094 753.9756 Tm
```

```
(with a link )Tj
ET
EMC
/P <</MCID 2 >>BDC
0 0 0 rg
BT
/T1_0 1 Tf
14 0 0 14 174.486 753.9756 Tm
(inside. )Tj
ET
EMC
```

The following is a illustration of the structure tree, approximately as shown in the Tags view, with the children of a structure tree element indented beneath the element:

```
/P
    MCID 0 Reference
    /Link
        MCID 1 reference
        OBJR to link annotation
    MCID 2 reference
```

Example 2

A Link *Structural Element* may contain several link annotations if the linked information is discontinuous. In this case, all the link annotations must have the same target. For instance, the paragraph

This is a two sentence paragraph and this is
a six word link. This is the second sentence.

contains a simple link that needs to be covered by two link annotations, since it splits across a line break. It would be represented in the Tags view as follows:

```
<P>
    MCID reference for “This is a two sentence paragraph and”
    <Link>
        MCID reference for “this is a six word link.”
        OBJR to link annotation covering “this is”
        OBJR to link annotation covering “a six word link”
    MCID reference for “This is the second sentence.”
```

Illustrations

In Tagged PDF, an Illustration (defined in Terminology section) is an *Structural Element* of the *Logical Structure* that is role-mapped to a member of the set { Figure Form Formula }.

Illustrations are considered to be PDF graphical objects. Therefore, an *Illustration* cannot occur within a Text Object (BT-ET sequence).

Clipping is part of an *Illustration* if and only if it is supplied by a contained marked clipping sequence as defined by the rules of section 8.4.2. In Tagged PDF, the tag associated with each marked clipping sequence is required to be **Clip**.

Note: *The required **BMC** tag does not alter the specification of section 8.4.2 at all. It merely adds a redundant marking on a marked clipping sequence so that a consumer of* Tagged *PDF does not have to perform the difficult semantic analysis of section 8.4.2 in order to ascertain whether a given marked-content is, in fact, a marked clipping sequence.*

For the purposes of reflow, *Illustrations* are considered as units-of-reflow and are moved (and perhaps resized) as a unit, without examination of their interior.

For access, a **Figure** that is used as a general *Illustration* should always have an **Alt** attribute, a **Figure** used in place of a character (for instance, a dropped cap) should always have the **ActualText** attribute.

In addition to common inline- and/or block-level properties (the **Placement** property determines if inline-level properties, or block-level properties, or both apply to the *Structural Element*), an *Illustration* may have **BaselineShift**, **BBox**, **Height**, **Placement** & **Width** properties, carried in the *Logical Structure* as part of the Attributes (**A**) list in a dictionary owned by **Layout** or can be set indirectly through the use of a dictionary in **ClassMap** with the same owner and selected via the class (**C**) key.

The following list identified additional restrictions caused by certain special uses of an illustration:

1. When a **Figure** element has a **Placement** of **Block**, the value of **Height** must be <length> and cannot be **Auto**. The value is the advance amount in the block-progression-direction (depending on **WritingMode**, this is either **AdvanceX** or **AdvanceY** specified in page units [points]).

2. When a **Figure** element has a **Placement** of **Inline**, the value of **Width** must be a <length> and cannot be **Auto**. This value is used to determine the advance width for reflow.The value is the advance amount in the inline-progression-direction (depending on **WritingMode**, this is either **AdvanceX** or **AdvanceY** specified in page units [points]).
3. When a **Figure** element has a **Placement** of **Inline**, **Start** or **End**, the value of **BaselineShift** is used to determine the position of the after-edge (bottom) of the character relative to the text-line's baseline. **BaselineShift** is ignored for all other values of **Placement**.(A Figure element with the **Placement** value of **Start** can be used to create a dropped-cap. The Placement value of Inline can be used to create a raised cap.)

Illustrations may be nested in the *Logical Structure*.

Note: *Many consuming applications will treat an illustration as an elementary object, and will not examine its internal structure.*

All *Illustrations* that are contained in their entirety within a page-region are required to have a **BBox** property.

Illustrations Referred To By Paragraphs

It is often the case that an *Illustration* will logically be part of, or at least attached to, a paragraph or other document entity.

Any such containment or attachment is represented within the *Logical Structure* through the use of the **Figure** element. The **Figure** element is inserted in the *Logical Structure* at the point of attachment and the **Placement** property is used to describe the nature of the attachment.

Tables

Tagged PDF incorporates the table-in-PDF definition found in Adobe Technical Note 5402, *Use of PDF Logical Structure by Acrobat Web Capture*.

Table elements are handled in the same manner as **Figure** elements and make identical use of the **Placement**, **Height**, **Width**, **BaselineShift**, and ***BBox*** properties.

Tables and the components within a table need to be marked in *Logical Structure* with **Table**, **TR**, **TH**, & **TD** elements for access and for content extraction.

Recommended Organization of the Logical Structure Tree

Note: *The Recommended Content Rules in this section are expressed in XML syntax. They are recommendations, since the PDF applications have no way to enforce these.*

Recommended Content Rule for Document, Part, Art(icle), Sect(ion), and Div

The **Document**, **Part**, **Art**(icle), **Sect**(ion), and **Div** elements are all block-level grouping elements. That is they are used to provide the high-level structure of a document.

1. Weakly structured:
   ((**H** | **H1** | **H2** | **H3** | **H4** | **H5** | **H6** | **P** | **BibEntry** | **BlockQuote** | **Code** | **Figure** | **Form** | **Formula** | **List** | **Note** | **Table** | **Sect** | **Part** | **Article** | **Div** | **TOC** | **Index**)*)
2. More-strongly structured:
   (**H**, (**P** | **BibEntry** | **BlockQuote** | **Code** | **Figure** | **Form** | **Formula** | **List** | **Note** | **Table**)*,**TOC**?, (**Sect*** | **Part*** | **Art*** | **Div***), **Index**?)

TOC and **Index** are very similar to **Sect**.

Recommended Content Rule for H(eading), H1 – H6, P, L(ist), BlockQuote

The **H**(eading), **H1 – H6**, **P**, **L**(ist), and **BlockQuote** elements are block-level components. These components may contain other block-level components, but would normally contain text or inline-level elements.

1. Many export file formats allow only text and inline elements within these objects. The Recommended Content Rule is:
   (**MCD** | **BibEntry** | **Code** | **Figure** | **Form** | **Formula** | **Lbl** | **Note** | **Table** | **Link** | **Quote** | **Reference** | **Span**)*
2. MCD is “Marked-Content Data.” It is equivalent to #PCDATA in XML/HTML.
   For some applications, block-level elements may be intermingled with inline level elements. In these environments, the Recommended Content

Rule becomes:
(**MCD** | **H** | **H1** | **H2** | **H3** | **H4** | **H5** | **H6** | **P** | **BibEntry** | **BlockQuote** | **Code** | **Figure** | **Form** | **Formula** | **Lbl** | **List** | **Note** | **Table** | **Div** | **Link** | **Quote** | **Reference** | **Span**)*

LI (List Item) elements are block-level components of the **List** elements. **Lbl** and **LBody** are block-level elements of **LI**.

Recommended Content Rule for L(ist):

L(ist) may contain: ((**P** | **Caption**), **LI***).

Recommended Content Rule for LI (ListItem):

LI may contain: (**Lbl***, **LBody***).

Recommended Content Rule for LBody:

LBody may contain: another **L**(ist) element, multiple block-level elements, or may have the same Recommended Content Rule as **P**.

Recommended Content Rule for Inline-level Structural Elements:

Only text and inline elements are allowed in an inline element. The recommended content rule is:

(**MCD** | **BibEntry** | **Code** | **Figure** | **Form** | **Formula** | **Lbl** | **Note** | **Table** | **Link** | **Quote** | **Reference** | **Span**)*

Link, **Quote**, **Reference**, & **Span** are inline components, thus must be contained in a block-level component.

BibEntry, **Code**, **Figure**, **Form**, **Formula**, **Lbl**, **Note**, and **Table** may be used either as block-level or inline-level elements, but may be treated only as block-level by some formatting applications.

TR & **Caption** are block-level components of **Table**.

TD & **TH** are block-level components of **TR**.

3.2 Use of Standard Properties in Tagged PDF

In addition to standard roles, this document identifies standard styling properties to be used with these roles. Again, this allows for predictable formatting and translation.

Many of these properties are placed in individually owned dictionaries within the *Structural Element's* attribute (**A**) key or can be set indirectly through the use of a ClassMap and corresponding class (**C**) key. These *owned* properties are for the use of specific downstream processes such as reflow or individual content extraction filters. The specific semantics for these properties are as defined by the individual owner. Multiple owners may define like-named properties, but assign differing value-types, value names, and/or semantics. These owned properties have no effect on the PDF rendering. Tagged PDF defines specific semantics for some of these "owned" properties (called *Standard Properties* in this document) that may be shared by multiple downstream processes.

Note: *Names used by the Class (C) key to identify a member of a ClassMap are local to a document and do not need to be registered with Adobe (see "Third class" names in the PDF Reference, Version 1.3, 2nd Edition -- Appendix E).*

Owned properties are placed in a list of dictionaries within the **StructElem**'s **A** key. Each dictionary has an owner specified by a key-value pair in the dictionary with the key **O** or in an identical manner within the **ClassMap**. The following table identifies owners that have assigned uses:

TABLE 9 Standard Owners of Attribute Dictionaries

OWNER	DESCRIPTION
Layout	Properties that are shared by reflow, access, and Repurposing. These properties may be converted to the appropriate form for use by reflow or access or when converting to: HTML-3.20, HTML-4.01, OEB-1.0, RTF, and other file formats.
Link	Identifies that this element has an attached link or reference.
List	Properties controlling the numbering of a list.
Table	The table cell's **RowSpan** and **ColSpan**.
XML-1.00	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to XML.

HTML-3.20	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to HTML-3.20.
HTML-4.01	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to HTML-4.01.
OEB-1.00	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to OEB-1.0.
RTF-1.05	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to Microsoft Rich Text Format.
CSS-1.00	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to a format using CSS-1.00.
CSS-2.00	Additional properties (other than those owned by **Layout**, **Link**, **List**, and **Table**) that will be output when repurposing to a format using with CSS-2.00.

If the same property is found in both the selected **ClassMap** and under the **A** key of the **SE**, then the one found under the **A** key in the one that will be used.

The Standard Properties fall into five categories:

1. Properties specified directly in the *Structural Element* obj dictionary.
   Properties in this category are encoded in the manner shown in the following example:

```
15 0 obj
<<
/Type /StructElem
...
/Lang (en-us)
/Alt (Alternative text)
/ActualText (Substitute text)
>>
endobj
```

2. Properties chosen through the **C** attribute in the *Structural Element*.
   Entries in the *Structural Element's* attribute (A) list owned by **Layout**, **Link**, **List**, and **Table**.

3. Properties in the *Structural Element's* attribute (A) dictionary owned by **Layout**, **Link**, **List**, and **Table** are encoded in the manner shown in the following example (**Layout** and **Link** are used as the sample owners):

```
15 0 obj
<<
   /Type /StructElem
   ...
   /A  [
      <<
         /O /Layout
         /SpaceBefore 12.0
         ...
      >>
      <<
         /O /Link
         /S /L
         ...
      >>
      ]
>>
endobj
```

4. Entries in the *Structural Element's* A attribute's dictionary owned by **XML-1.00**, **HTML-3.20**, **HTML-4.01**, **OEB-1.00**, **RTF-1.05**, **CSS-1.00**, **CSS-2.00**, or any other specific export format that are applied only when that file format is the active export format. This list is open-ended and is not specified in this document. Properties in the *Structural Element's* attribute (**A**) dictionary owned by **XML-1.00**, **HTML-3.20**, **HTML-4.01**, **OEB-1.0**0, **RTF-1.05, CSS-1.00**, **CSS-2.00** are encoded in the manner shown in the following example (**XML**-1.00 and **HTML-3.20** are used as the sample owners):

```
15 0 obj
   <<
/Type /StructElem
...
/A  [
   <<
      /O /XML-1.00
      /Class (my-class)
      ...
   >>
   <<
      /O /HTML-3.02
```

```
            /Class (Sect)
            ...
        >>
        ]
    >>
    endobj
```

5. Properties that are marked directly in instream markings such as BMC...EMC or DP.

3.3 Definitions of Standard Properties used by Tagged PDF

The following two subsections contain the definitions and usage guidelines for all Standard Properties used in Tagged PDF.

Each description consists of:

- The key (name) of the property.
- The general value type (data type).
- An overview definition of the property's semantics.

  1.) The list of values or value types that may be assigned to the property;

  2.) Whether the property optional or required (optional unless otherwise specified); -and-

  3.) the default value. (One should note that even required values have a default, however the default on a required property is designed to produce a safe value that is inaccessible/unusable, thus effectively disables the use of the property on the associated element.)
- A table describing the specific meaning of each value.
- Additional notes or additional description of the values, if needed.

Keys used in the *Structural Element's* obj Dictionary

TABLE 10 Keys used in the Structural Element's obj Dictionary

KEY	TYPE	SEMANTICS
ActualText	string	*(Optional)* The **ActualText** property is a string that is an exact text replacement for the *Structural Element*. It is used to provide the actual individual letters for text glyphs that represent a non-standard ligature, a custom charac-

ter, a logo, etc. or for text that is replaced by an inline graphic such as the initial letter in an illuminated manuscript. If the **ActualText** string is non-empty, then an aural reader will read the actual text and will not process any content nor children of the *Structural Element*. This property has the following values: <string>. The default value is *Unspecified*.

Value	Description
<string>	The text to be used as an alternate representation of this *Structural Element* and its children.

Note: *Any ActualText should surround as small a piece of the page-content as possible. It is not possible to match the characters of the ActualText with the glyphs within the ActualText's marked-content. As a result, consumers of Tagged PDF such as reflow programs cannot perform, for example, word-breaking within a ActualText marked-content.*

Note: *The Alt ActualText property is treated as if it were a character substitution for the current Structural Element. Thus if 2 objects each having an ActualText property are encountered in sequence, they should be treated as if no word break is present between them.*

Alt	string	*(Optional)* The **Alt** property is used by the access reader to provide a description of a figure or formula for aural readers. This property has the following values: <string>. The default value is *Unspecified*. For a complete definition of this property, see the PDF Reference. ***Note:*** *The Alt property is treated as if it were a word or phrase substitution for the current Structural Element. Thus if 2 objects each having an Alt property are encountered in sequence, they should be treated as if a word break is present between them.*
Lang	string	*(Optional)* The language of a word in a document is determined in a hierarchical fashion by the language specification dictionary keys in the **Catalog** and in *Structural Element* objects and by properties attached to marked content with the span tag. The key is optional. The value is of type string and the empty string indicates that the language is unknown. This property has the following values: **<IETF RFC-1766-language-code>**. If this property is not present, the hierarchy must be looked up to determine the language. ***Note:*** *ISO 639 language codes can be found at http://lcweb.loc.gov/standards/iso639-2*

and IANA registered language codes can be found at http://www.isi.edu/in-notes/iana/assignments/languages.

Keys Owned by Layout

The following list summarizes the usage of properties owned by Layout:

1. Properties on any *Block-level Structural Element*:

   SpaceAfter

   SpaceBefore

   EndIndent

   StartIndent

   WritingMode

Note: *If a Structural Element has a placement property value other than Inline (and does not default to Inline), then the above 5 properties apply to the Structural Element.*

2. Additional properties that apply to *Block-level Structural Elements* that contain text:

   TextAlign

   TextIndent

3. Properties on any *Inline-level Structural Element*:

   LineHeight

   BaselineShift

   TextDecorationType

Note: *The above 3 properties, though they apply to any ILSE, may be specified on any BLSE and will apply to any content of the BLSE that is not wrapped in a Structural Element nested within this BLSE.*

Note: *If a Structural Element has a placement property value of Inline (or defaults to Inline), then the above 3 properties apply to the Structural Element.*

4. Properties on **Figure**, **Form**, **Formula**, **Code**, **Note**, or **Table** *Structural Element*s:

   Height

   Width

Placement

BBox

5. Properties on **TD**, or **TH** *Structural Elements*:

Height

Width

BlockAlign

InlineAlign

Property definitions in the following table are listed in alphabetical order.

TABLE 11 Keys Owned by Layout

KEY	TYPE	SEMANTICS
BaselineShift	number	*(Optional)* Used to create inferiors (subscripts) and superiors (superscripts). This property is also used to position inline figures relative to the baseline. This property has the following values: <length>. The default value is 0 (zero). The **BaselineShift** property allows repositioning of the baseline relative to the baseline of the parent element. The shifted object might be a subscript, a superscript, or an inline graphic. Within the shifted object, the whole baseline-table is offset; not just a single baseline. When a baseline shift is applied to an element, the content of the element and all its children are shifted by the amount specified. Any **BaselineShift** properties that are applied to a child of this element are measured relative to the shifted result of this (parent) element. This is not the full behavior of a *Reference-area* (for example, it does not define bounds for measuring indents), thus is called a reference-frame. Value — Description <length> — The baseline is shifted in the shift-direction (positive value, toward superscript) or opposite to the shift-direction (negative value, toward subscript) of the parent element's baseline by the <length> value.
BBox	rectangle	Required if a **Table**, **Form**, **Formula**, or **Figure** element appears in its entirety on a single page. The extent of the Table, Form, or Figure on the page, in page-space (points).

Value	Description
<rectangle>	The page-extent of the Figure, Form, or Table element.

BlockAlign name *(Optional)* Controls placement of content within a table-cell. (This is equivalent to “valign” in HTML.) This property specifies the alignment, in the block-progression-direction, of the areas that are the children of a *Reference-area*. This property has the following values: Before | Middle | After | Justify. The default value is Before.

Value	Description
Before	The before-edge of the allocation-rectangle of the first child area is placed coincident with the before-edge of the content-rectangle of the *Reference-area*.
Middle	The child areas are placed such that the distance between the before-edge of the allocation-rectangle of the first child area and the before-edge of the content-rectangle of the *Reference-area* is the same as the distance between the after-edge of the allocation-rectangle of the last child area and the after-edge of the content-rectangle of the *Reference-area*.
After	The after-edge of the allocation-rectangle of the last child area is placed coincident with the after-edge of the content-rectangle of the *Reference-area*.
Justify	Equal spacing is inserted between all children of this area, such that the first child has the alignment **Before** and the last child has the alignment **After**. If there is only one child, it will be placed as if **Before** had been specified.

EndIndent number *(Optional)* Distance from the end-edge of the *Reference-area* to the end-edge of all lines in the *BLSE*; larger values make the line shorter. This property has the following values: <length>. The default value is 0.0 points.

Value	Description
<length>	For each block-area generated by this *Structural Element*, specifies the distance from the end-edge of the content-rectangle of that block-area to the end-edge of the content-rectangle of the containing *Reference-area*. This property may have a negative value, which indicates a hanging indent (outdent), but various export formats may have implementation-specific limits.

Height name or number

(Optional) Desired-height of the *Reference-area* to be used to lay out this object (a form, formula, table, figure), measured in the block-progression-direction as established by the current **WritingMode**. This property applies to graphics and tables, it does not apply to text (when the "line-height" property applies). This property has the following values: <length> | Auto. The default value is Auto.

Value	Description
Auto	No constraint is imposed by this property. The block-progression-dimension is determined by the intrinsic size of the object.
<length>	Specifies an explicit block-progression-dimension.

InlineAlign name *(Optional)* Controls "lateral" placement of content within a table-cell. (This is equivalent to "halign" in HTML.) This property specifies the alignment, in the inline-progression-direction, of the areas that are the children of a *Reference-area*. This property has the following values: **Start** | **Center** | **End** | **Justify**. The default value is **Start**.

Value	Description
Start	The start-edge of the allocation-rectangle of the first child area is placed coincident with the start-edge of the content-rectangle of the *Reference-area*.
Center	The child areas are placed such that the distance between the start-edge of the allocation-rectangle of the first child area and the start-edge of the content-rectangle of the *Reference-area* is the same as the distance between the end-edge of the allocation-rectangle of the last child area and the end-edge of the content-rectangle of the *Reference-area*.
End	The end-edge of the allocation-rectangle of the last child area is placed coincident with the end-edge of the content-rectangle of the *Reference-area*.
Justify	Equal spacing is inserted between all children of this area, such that the first child has the alignment **Start** and the last child has the alignment **End**. If there is only one child, it will be placed as if **Start** had been specified.

LineHeight name or number

(Optional) Preferred height of an *Inline-Level Structural Element*. The

largest **LineHeight** for any fragment of any *Inline-level Structural Element* in a given text line is used as the height of the line. This property has the following values: **Normal** | **Auto** | <length>. The default value is **Normal**.

Note: *This is an inline-level property, not a Block-level property. However, this property may be placed on a BLSE and will apply to any content within the BLSE that is not wrapped in an element nested within the BLSE as if the unwrappped content had been wrapped in a span element bearing this LineHeight.*

Value	Description
Normal	Tells user agents to set the height of the *Inline-level Structural Element* to a reasonable value based on the font size for the *Structural Element*. In calculating the height of the line, the minima and maxima are adjusted to include any non-zero **BaselineShift**.
Auto	Tells user agents to set the height of the *Inline-level Structural Element* to a reasonable value based on the font size for the *Structural Element*. In calculating the height of the line, the maxima and minima are calculated as if **BaselineShift** had been set to zero.
<length>	The height is set to this length. Negative values are illegal.

Note: Both **Normal** and **Auto** include the term *reasonable*. Absent any other property to modify this definition, *reasonable* is assumed to be approximately 1.2 times the font size. The coefficient may differ from the assumed value of 1.2 in each of the content extraction formats. The value Normal or Auto is the value actually issued by content extraction if available in the output format, hence the reason these are defined in terms of *reasonable*.

Note: If the **LineHeight** property is set to Normal or Auto, one reasonable method to calculate the font size to be used in calculating a numeric line height from the information in a PDF file is to take the maximum (most positive) value across all characters of the applicable span of the result of multiplying the current font’s Ascent by the current composite of the GraphicState (CTM) and TextState (Tm) matrices and the size operand of the current **Tf** operator (as described in the section Text Rendering Matrix in section 5.3.2 of the PDF Reference) then subtract the minimum (most negative) value calculated in a similar manner using the font's Descent value. One should note that if the font, size, or the Tm matrix changes then this calculation should be treated as if text string was broken into two ILSEs at the point where the change occurs.

Placement name *(Optional)* This property specifies whether a box should float to the top of page, left side of the column, right side of the column, be placed as a nested *BLSE*, or be placed inline (as a character-level object). Placement may have the following values: Inline | Block | Start | End | Before | After. The default value is: Inline.

Value	Description
Inline	Specifies that the areas created by this *Structural Element* shall be laid out as if they were a baseline-resting character in the text line.
Block	Specifies that the areas created by this *Structural Element* shall be laid out as if it were a nested *BLSE*.
Start	Specifies that the areas created by this *Structural Element* may float and shall be placed so that the start-edge of the floating area coincides with the start-edge of the nearest containing *Reference-area*. This is created as a runaround.
End	Specifies that the areas created by this *Structural Element* may float and shall be placed so that the end-edge of the floating area coincides with the end-edge of the nearest containing *Reference-area*. This is created as a runaround.
Before	Specifies that the areas created by this *Structural Element* may float and shall be placed so that the before-edge of the floating area coincides with the before-edge of the nearest containing *Reference-area*. This is created as a full-width block and pushes any normal content of the *Reference-area* to begin at the after-edge of the float's area.

If multiple floats are encountered, they may be stacked adjacent to one-another against the specified edge of the *Reference-area* in the order the figure elements are encountered. (The first one encountered is placed against the *Reference-area*'s edge, the second is placed adjacent to and inboard of the first, and so on.)

Note: *StartIndent is applied to Structural Elements with a Placement of Block or Start and EndIndent applies to Structural Elements with a Placement of Block or End.*

Note: *If a Structural Element that is placed adjacent to one with a Placement of Start has a StartIndent, the actual starting indent for each of its lines of* *text will be the maximum of its StartIndent property and the ending edge position on the floating elements. This actual starting indent of the line may then be further adjusted by the TextIndent property, if applicable to the line.*

SpaceAfter number *(Optional)* Specifies the space after the last line of this *BLSE* in the block-progression-direction. If the subsequent object has a **SpaceBefore** specification, the greater of this object's **SpaceAfter** and the subsequent object's **SpaceBefore** is to be used. This property has the following values: <length>. The default value is 0.0 points.

Value	Description
<length>	Specifies the space before the first line of this *BLSE* in the block-progression-direction.

This value is disregarded if this is the last item placed in the *Reference-area*. This value is added to any adjustments induced by the **LineHeight** property.

SpaceBefore number *(Optional)* Specifies the space before the first line of this *BLSE* in the block-progression-direction. If the prior object has a **SpaceAfter** specification, the greater of this object's **SpaceBefore** and the prior object's **SpaceAfter** is to be used. This property has the following values: <length>. The default value is 0.0 points.

Value	Description
<length>	Specifies the space before the first line of this *BLSE* in the block-progression-direction.

This value is disregarded at the if this is the first item placed in the *Reference-area*. This value is added to any adjustments induced by the **LineHeight** property.

StartIndent number *(Optional)* Distance from the start-edge of the *Reference-area* to the start-edge of all lines in the *BLSE*; larger values make the line shorter. This property has the following values: <length>. The default value is 0.0 points.

Value	Description
<length>	For each block-area generated by this *Structural Element*, specifies the distance from the start-edge of the content-rectangle of the containing *Reference-area* to the start-edge of the content-rectangle of that block-area.

This property may have a negative value, which indicates a hanging indent (outdent), but various export formats may have implementation-specific limits.

TextAlign	name	*(Optional)* The justification of the lines within a paragraph. This property has the following values: Start | Center | End | Justify. The default value is Start.

Value	Description
Start	Specifies that the content is to be aligned on the start-edge in the inline-progression-direction.
Center	Specifies that the content is to be centered in the inline-progression-direction.
End	Specifies that the content is to be aligned on the end-edge in the inline-progression-direction.
Justify	Specifies that the contents is to be expanded to fill the available width in the inline-progression-direction. The last (or only) line of any block with **TextAlign**=Justify specified will be aligned as if **TextAlign**=Start had been specified.

TextDecorationType name *(Optional)* Specifies if text is to have underline, strike-through, etc.

This property has the following values:

None |

Underline | Overline | LineThrough

The default value is **None**.

This property describes decorations that are added to the text of an *Structural Element*. If the property is specified for a *Block-level Structural Element*, it affects all *Inline-level* descendants of the *Structural Element*. If it is specified for (or affects) an *Inline-level Structural Element*, it affects all boxes generated by the *Structural Element*. If the *Structural Element* has no content or no text content (for example, the IMG *Structural Element* in HTML), user agents must ignore this property.

All children of an element with this property should be formatted with the same decoration (for example, they should all be underlined). The element and its children should all be formatted as if on a single line, then the decoration applied to the formatted result as if the formatted result was a single item. The decorated result can then be broken into lines if necessary. The color of decoration should remain the same even if the child elements have different color values. The position and thickness of the decoration should be uniform across all the children.

Value	Description
None	Produces no text decoration.
Underline	Each line of text is underlined.
Overline	Each line of text has a line above it.
LineThrough	Each line of text has a line through the middle

TextIndent number *(Optional)* Additional **StartIndent** for the first line of the paragraph. Algebraically adds to **StartIndent**. This property has the following values: <length>. The default value is 0.0 points.

Value	Description
<length>	Specifies the additional indent to be applied on the start-edge of the first line of a paragraph or Block-level Structural Element. This property may have a negative value, which indicates a hanging indent (outdent), but various export formats may have implementation-specific limits.

Width name or number

(Optional) Desired-width of the *Reference-area* to be used to lay out this object (a form, formula, table, figure), measured in the inline-progression-direction as established by the current **WritingMode**. This property applies to graphics and tables, it does not apply to text. This property has the following values: <length> | **Auto**. The default value is **Auto**.

Value	Description
Auto	No constraint is imposed by this property. The inline-progression-dimension is determined by the intrinsic size of the object.
<length>	Specifies an explicit inline-progression-dimension.

WritingMode name *(Optional)* Specifies the direction of character/inline layout progression in each line (inline-progression-direction) and the direction of layout of subsequent lines/blocks in a column (block-progression-direction) for the *Structural Element* and all its children. This property has the following values: LrTb | RlTb | TbRl. The default value is LrTb.

The **WritingMode** is specified as a pair of directions-specifiers such as LrTb. For the basic writing modes, the first component specifies the inline-progression-direction (Lr=left-to-right, Rl=right-to-left, Tb=top-to-bottom, and Bt=bottom-to-top), the second component specifies the block-progression-direction.

When **WritingMode** is applied to an object that produces multiple columns, it defines the column-progression within each region. The inline-progression-direction is used to determine the stacking direction for columns (and the default flow order of text from column-to-column).

When **WritingMode** is applied to a table, it controls the layout of the rows and columns. Table-rows use the block-progression-direction as the row-stacking direction. The inline-progression-direction is used to determine the stacking direction for columns (and cell order within the row).

Value	Description
LrTb	Inline components and text within a line are written left-to-right. Lines and blocks are placed top-to-bottom. Note(s): Typically, this is the writing-mode for normal alphabetic text. Establishes the following directions: • inline-progression-direction to left-to-right – Start is set to left, end is set to right. If any right-to-left reading characters are present in the text, the inline-progression-direction for glyph-areas may be further modified by the Unicode bidi algorithm. • block-progression-direction to top-to-bottom; Before is set to top, after is set to bottom • shift-direction to bottom-to-top; A positive value moves toward top, a negative value moves toward bottom.
RlTb	Inline components and text within a line are written right-to-left. Lines and blocks are placed top-to-bottom. ***Note:*** *Typically, this writing mode is used for Arabic and Hebrew text.* Establishes the following directions:

- inline-progression-direction to right-to-left; Start is set to right, end is set to left. If any left-to-right reading characters or numbers are present in the text, the inline-progression-direction for glyph-areas may be further modified by the Unicode bidi algorithm.
- block-progression-direction to top-to-bottom; Before is set to top, after is set to bottom.
- shift-direction to bottom-to-top; A positive value moves toward top, a negative value moves toward bottom.

TbRl Inline components and text within a line are written top-to-bottom. Lines and blocks are placed right-to-left.

Note: Typically, this writing mode is used for Chinese and Japanese text.

Establishes the following directions:

- inline-progression-direction to top-to-bottom; Start is set to top, end is set to bottom.
- block-progression-direction to right-to-left; Before is set to right, after is set to left
- shift-direction to left-to-right; A positive value moves toward right, a negative value moves toward left

Keys Owned by List

The **ListNumbering** property is specified on the **L** (List) element, but controls the interpretation of the **Lbl** element within **LI** (List Item) elements within that List.

TABLE 12 Properties Owned by List

KEY	TYPE	DESCRIPTION
ListNumbering	name	*(Optional)* For auto-numbered lists, specifies the numbering system to be used. For un-numbered lists, specifies the symbol to be used to identify each list-item. This property has the following values: **None | Disc | Circle | Square | Decimal | LowerRoman | UpperRoman | LowerAlpha | UpperAlpha**. The default value is **None**. ***Note:*** *The ListNumbering property is used to allow a content extraction tool to auto-number a list. The* ***Lbl*** *element should*

contain the resultant number so that the document can be re-flowed or printed without full reformatting.

Value	Description
None	List is not auto-numbered. The Lbl *Structural Element* can be provided to contain manual numbering or the term in a dictionary list. If Lbl is absent, it is an unnumbered list.
Disc	Solid circular bullet (filled circle)
Circle	Open (unfilled) circular bullet.
Square	Solid square bullet.
Decimal	1, ..., 9, 10, ..., 99, etc. The User agent may optionally follow the number with a period or a closing parenthesis.
LowerRoman	i, ii, iii, iv, ...
UpperRoman	I, II, III, IV, ...
LowerAlpha	a, b, c, ... (Alphabet chosen depends on the current Lang property)
UpperAlpha	A, B, C, ... (Alphabet chosen depends on the current Lang property)

Note: This list may be expanded as Unicode identifies additional numbering systems.

Keys Owned by Table

These properties apply only to table cells (**TD** & **TH**).

TABLE 13 Properties Owned by Table

KEY	TYPE	DESCRIPTION
ColSpan	number	*(Optional)* Defines the number of columns in a table that are spanned by the current cell. The cell expands by adding columns in the column-progression-direction (inline-progression-direction) specified for the table.This property has the following values: a positive-integer. The default value is 1.

Value	Description
positive-integer	Number of columns spanned by the cell.

RowSpan	number	*(Optional)* Defines the number of rows in a table that are spanned by the current cell. The cell expands by adding rows in the row-progression-direction (block-progression-direction) specified for the table.This property has the following values: a positive-integer. The default value is 1.

Value	Description
positive-integer	Number of rows spanned by the cell.

(Section 4 below — To be added to Section 3.6.1 Document Catalog)

4 Additional Requirements for Tagged PDF Acceptance

The following new entry in the **Catalog** indicates that the document is a Tagged PDF document.

TABLE 14 Additions to the Catalog

KEY	TYPE	DESCRIPTION
MarkInfo	dictionary	(*Required* for PDF files conforming to the Tagged PDF specification) Contains information related to the Tagged PDF properties of this document.

TABLE 15 Contents of the MarkInfo dictionary

KEY	TYPE	DESCRIPTION
Marked	boolean	*(Optional)* A value of true indicates that the document is a Tagged PDF. Default value: **false**.

Transparency in PDF

February 5, 2001

1 Introduction

This document describes how the Adobe® imaging model is extended to include transparency and specifies how transparency is represented in the Adobe Portable Document Format (PDF). Under the transparency model, graphics objects do not necessarily obey a strict opaque painting model; instead, they can blend in interesting ways with other objects that overlap.

Support for transparency is first introduced in PDF version 1.4, which will be the native file format for Adobe Acrobat® 5. The first Adobe application to support PDF transparency is Adobe Illustrator® 9, which supports PDF 1.4 as a native file format.

The purpose of this document is to provide preliminary information that will be useful to developers working with Illustrator 9. This document describes only the transparency features of PDF 1.4. Other features of PDF 1.4 will be published following the introduction of Acrobat 5.

1.1 About This Document

The transparency extensions to the Adobe imaging model are very general. This document describes the general model to the extent necessary to understand the PDF extensions for transparency. However, it does not describe the realization of the transparency extensions in applications such as Adobe Illustrator or Adobe Photoshop®.

This document also does not cover how the transparency model is to be implemented. We use implementation-like descriptions at various points to describe how things work, but this is only for the purpose of elucidating the behavior of the model. Please bear in mind that the actual implementation will almost certainly be different from what might be implied in these descriptions.

This document is organized as follows:

- Section 2, “Overview,” introduces the basic concepts of the transparency model and its associated terminology, including *shape*, *opacity*, *alpha*, *blend mode*, *stack*, *backdrop*, and *group*.
- Section 3, “Color Compositing Computations,” describes the mathematics of computing a result color as a function of source and backdrop colors, alphas, and blend mode.
- Section 4, “Shape and Opacity Computations,” continues in the same vein and covers the related shape and opacity computations.
- Section 5, “Groups,” introduces the concept of groups and covers their properties and semantics.
- Section 6, “Soft Masks,” covers the creation and use of masks to specify position-dependent shape and opacity.
- Section 7, “Color Space and Color Rendering Issues,” describes the interactions between the transparency model and other aspects of color specification and rendering in the existing Adobe imaging model.
- Section 8, “Overview of PDF Extensions,” gives a brief overview of how the transparency extensions to the Adobe imaging model are represented in PDF.
- Section 9, “PDF Specification,” gives a detailed description of the PDF extensions, organized according to *PDF Reference*.
- Section 10, “Terminology Summary,” is an alphabetized summary of terminology used in this document.
- Section 11, “Compatibility,” discusses compatibility with PDF 1.3 and with the PostScript® language.
- Section 12, “Overprinting, Erasing, and Transparency,” presents details of the existing overprinting and erasing rules in PDF 1.3 and their equivalent representation in PDF 1.4 as a form of transparency.
- There is a revision history at the end.

1.2 Related Documents

The PDF and PostScript specifications mentioned below are available on the Adobe Solutions Network (ASN) Developer Program web site, located at:

< http://partners.adobe.com/asn/developer/ >

The official specification for PDF is now the book *PDF Reference, second edition*, published in July 2000 by Addison-Wesley (and also available on-line). That book is a self-contained reference for PDF, incorporating all information about the Adobe imaging model that formerly was documented only in *PostScript Language Reference, third edition*. Additionally, the PDF material has been extensively reorganized and revised. *PDF Reference, second edition* supersedes the former *Portable Document Format Reference Manual, version 1.3*.

This document refers to the PDF specification as, for example, *PDF Reference*, Section 1.4 [1.7]. The first number refers to a section in the new *PDF Reference, second edition*. The second number (in brackets) refers to a section in the old *Portable Document Format Reference Manual, version 1.3*, where the closest equivalent material can be found.

PDF Reference, second edition (PDF version 1.3)
Addison-Wesley, June 2000

Portable Document Format Reference Manual, version 1.3
March 11, 1999
Superseded by *PDF Reference, second edition*.

PostScript Language Reference, third edition
Addison-Wesley, February 1999

Compositing Digital Images
T. Porter & T. Duff, Computer Graphics (ACM), vol. 18 no. 3, July 1984

1.3 Intellectual Property

The information in this document is subject to the copyright permissions stated in *PDF Reference*, Section 1.4 [1.7]. Additionally, developers should be aware that many of the transparency extensions to the Adobe imaging model are the subject of patents and patents pending by Adobe Systems. The permission to use the copyrighted material in the PDF specification does not include the right to use any Adobe patents, except as may be permitted by an official Adobe Patent Clarification Notice (published at Adobe's web site or elsewhere).

2 Overview

This section introduces the general concepts behind the transparency extensions to the Adobe imaging model. Subsequent sections go into the model in greater detail. The technical terms introduced here are summarized in Section 10, "Terminology Summary."

2.1 Basic Concepts

The existing Adobe imaging model paints objects (fills, strokes, text, images), possibly clipped by a path, opaquely onto a page. One can think of the objects on a page as forming a *stack*, where the stacking order is defined to be the order in which the objects are specified, bottommost object first. At any given point on the page, the color of the page is defined to be the color of the topmost enclosing object, disregarding any overlapping objects lower in the stack. This effect can be—and often is—realized simply by rendering objects directly to the page in the order in which they are specified.

Under the transparency imaging model, all of the objects in a stack can potentially contribute to the result. At any given point, the color of the page is defined to be the result of combining the colors of all enclosing objects in the stack, according to some *compositing* rules that the transparency model defines.

Note: *The order in which objects are specified determines the stacking order, but not necessarily the order in which objects are actually painted onto the page. In particular, the model does not require the implementation to rasterize objects immediately or to commit to a raster representation at any time prior to rendering the entire stack onto the page. This is important, since rasterization often causes significant loss of information and precision that is best avoided during intermediate stages of the transparency computation.*

A given object is composited with a *backdrop*. Ordinarily, the backdrop consists of the stack of all objects that have been specified previously; the result is then treated as the backdrop for compositing the next object. However, within certain kinds of groups (see below), a different backdrop is chosen.

When an object is composited with the backdrop, the color at each point is modified by a function called the *blend mode*, which is a function of both the object's color and the backdrop color. The blend mode determines how colors interact; different blend modes can be used to achieve a variety of useful effects. A single blend mode is in effect for compositing all of a given object, but different blend modes can be applied to different objects.

Compositing of an object with the backdrop is mediated by two scalar quantities called *shape* and *opacity*. Conceptually, for each object, these quantities are defined at every point in the plane, just as if they were additional color components. (In actual practice, they are often obtained from auxiliary sources, rather than being intrinsic to the object itself.)

Both shape and opacity vary from 0 (no contribution) to 1 (maximum contribution). At any point where either the shape or the opacity is 0, the color is undefined. At any point where the shape is 0, the opacity is also undefined. The shape and opacity themselves are subject to compositing rules, so that the stack also has a shape and opacity at each point.

An object's opacity, in combination with the backdrop's opacity, determines the relative contributions of the backdrop's color, the object's color, and the blended color to the computed composite color. An object's shape then determines the degree to which the composite color replaces the backdrop color. Shape values of 0 and 1 identify points that lie "outside" and "inside" a familiar sharp-edged object, but intermediate values are useful in defining soft-edged objects.

Shape and opacity are very similar concepts. In fact, in most situations, they can be combined into a single value, called *alpha*, which controls both the color compositing computation and the fading between the object and the backdrop. However, there are a few situations in which they must be treated separately; see Section 5.5, "Knockout Groups." Moreover, raster-based implementations must maintain a separate shape parameter in order to do anti-aliasing properly; it is therefore convenient to have it be an explicit part of the model.

One or more consecutive objects in a stack can be collected together into a *transparency group*, hereafter referred to simply as *group*. The group as a whole can have various properties that modify the compositing behavior of objects within a group and their interactions with the backdrop of the group. Additionally, an additional blend mode, shape, and opacity can be associated with the group as a whole and used when compositing the group with its backdrop. Groups can be nested within other groups, so that the group hierarchy forms a tree structure.

Note: *The concept of transparency group is independent of the existing notions of "group" or "layer" in applications such as Adobe Illustrator. Those groupings reflect logical relationships among objects that are meaningful when editing those objects, but they are not part of the imaging model.*

The color result of compositing a group can be converted to a single-component luminosity value and treated as a *soft mask*, or just mask for short. Such a mask can then be used as an additional source of shape or opacity values during subsequent compositing operations. When the mask is used as shape, this technique is known as *soft clipping*; it is a generalization of the clipping path in the existing Adobe imaging model.

The *current page* is generalized to be a group consisting of the entire stack of objects placed on the page, composited with a backdrop that is white and fully opaque. Logically, this entire stack is then rasterized, determining the actual pixel values that are to be transmitted to the output device.

Note: *In contexts where a PDF "page" is to be treated as a piece of artwork to be placed on some other page, such as an Illustrator artboard or an encapsulated PostScript (EPS) file, we do not treat it as a page but as a group, whose backdrop may be defined differently from a page.*

2.2 Notation

The following are conventions for variable names used in this document. In general, a lowercase letter represents a scalar quantity, such as an opacity. An uppercase letter represents an *n*-tuple of scalar values, such as a color.

In the descriptions of the basic color compositing computations, color values are generally represented by the letter *C*, with a mnemonic suffix that indicates which of several color values is being referred to; for instance, *Cs* stands for "source color." Shape and opacity values are represented with the letters *f* (for "*f*orm factor") and *q* (for "opa*q*ueness"), with a mnemonic suffix, such as *qs* for "source opacity." The symbol α (alpha) stands for the product of the corresponding shape and opacity values.

In the descriptions of group transparency, the basic formulas are recast as recurrence relations and augmented with other formulas specifying group behavior. Here, variables have a numeric subscript indicating the position in the stack that the quantity is associated with, with the bottommost object numbered 0. Thus, Cs_i stands for "source color of the *i*th object in the stack."

In certain computations, one or more variables may have undefined values; for instance, when opacity is zero, the corresponding color is undefined. A value can also be undefined if it results from a division by zero. In any formula that uses the undefined value, the value has no effect on the ultimate result because it is subsequently multiplied by zero or otherwise cancelled out.

The important point is that any arbitrary value can be chosen for an undefined value, but the computation must not malfunction due to exceptions caused by overflow or division by zero. Additionally, it is convenient to adopt the convention that 0 / 0 = 0.

3 Color Compositing Computations

The primary change in the imaging model that comes with adding transparency is in how colors are painted. In the transparent model the result of painting, the *result color*, is a function of both the color being painted, the *source color*, and the color it is being painted over, the *backdrop color*. Both of these colors may vary as a function of position on the page, but for the purposes of this section we will concentrate our attention on some fixed point in the page and assume a fixed source and backdrop color.

Other parameters in this computation are the *alpha*, which specifies the relative contributions of the source and backdrop colors, and the *blend mode*, which allows one to customize how the source and backdrop colors are combined in the painting operation. This *color compositing function*, or just *compositing function* for short, determines the color result of a painting operation:

$$Cr = \left(1 - \frac{\alpha s}{\alpha r}\right) \cdot Cb + \frac{\alpha s}{\alpha r} \cdot ((1 - \alpha b) \cdot Cs + \alpha b \cdot B(Cb, Cs))$$

where the variable definitions are given in Table 1.

TABLE 1 Variables in the color compositing formula

VARIABLE	MEANING
αb	Backdrop alpha
αr	Result alpha
αs	Source alpha
$B(Cb, Cs)$	Function implementing the blend mode
Cb	Backdrop color
Cr	Result color
Cs	Source color

This is actually a simplified form of the compositing function where the shape and opacity values are combined and represented as a single alpha value. The more general form is presented later. This function is based on the Porter & Duff **over** operation, extended to include a blend mode in the region of overlapping coverage.

The following sections elaborate on the meaning and implications of these formulas.

3.1 Blending Color Space

First, note that the compositing function operates on colors. If the colors are represented by more than one scalar value then the computation treats them as vector quantities. To be precise, Cb, Cr, Cs, and $B(Cb, Cs)$ will all have n elements, where n is the number of components in the color space used for compositing. The above formula is then a vector function: the ith component of Cr is obtained by plugging in the ith components of Cs, Cb and $B(Cb, Cs)$.

Thus, the result of the computation will depend on the color space in which the colors are represented. For this reason, the color space used to represent colors for this computation is explicitly made part of the model and is called the *blending color space*. When necessary, source colors are converted to the blending color space prior to the compositing computation.

The following PDF color spaces are supported as blending color spaces: **DeviceGray**, **DeviceRGB**, **DeviceCMYK**, **CalGray**, **CalRGB**, and the equivalent **ICCBased** color spaces (including calibrated CMYK). The **Lab** space and the **ICCBased** spaces that represent lightness and chromaticity separately (such as Lab, Luv, and HSV) are not allowed, because the compositing computations in such spaces do not give meaningful results when done on a per-component basis. Additionally, an **ICCBased** color space used as a blending color space must be bidirectional; that is, the ICC profile must contain both *AToB* and *BToA* transforms.

The blending color space is consulted for blending of process colors only. Additionally, blending can be done on spot colors individually. Spot colors are never converted to a blending color space, except in the case where they first revert to their alternate color space. Instead, if an object is painted with a **Separation** or **DeviceN** color space, the specified color components are blended individually with the corresponding color components of the backdrop.

The blend mode functions assume that the range per color component is from 0 to 1, and that the color space is additive. The former is true for all of the allowed blending color spaces, but the latter is not. In particular, the **DeviceCMYK**, **Separation**, and **DeviceN** spaces are subtractive. When performing blending operations in subtractive color spaces, we assume that the color component values are complemented before the blend mode function is applied and that the results of the function are then complemented before being used. By *complemented* we mean that a color component value c is replaced with $1 - c$.

This adjustment makes the effect of the blend modes numerically consistent across all color spaces. However, the actual visual effect produced by a given blend mode still depends on the color space. Blending in a device color space produces device-dependent results. Blending in a CIE-based color space produces results that are consistent across all devices. Additional details about color space issues are given in Section 7, “Color Space and Color Rendering Issues.”

3.2 Blend Mode

The $B(Cb, Cs)$ term of the compositing function is used to customize the blending operation. This function of two colors is called the *blend mode*. Abstractly, this could be any function of the source and backdrop colors that returns another color. PDF defines a standard set of named functions for the blend mode; see Table 2 and Table 3.

All of the blend modes in Table 2 are defined by a scalar function that is applied separately to each color component, expressed in additive form:

$$cr = B(cb, cs)$$

where the lowercase *cr*, *cs*, and *cb* denote one component of the colors *Cr*, *Cs*, and *Cb*. Such a blend mode is called *separable*. This is in contrast to a function where the result for a particular component is a function of components other than the corresponding component in the backdrop and source colors. (In principle, a blend mode could have a different function for each component and yet remain separable; however, none of the blend modes listed below have that property.) A separable blend mode can be used with any color space, since it applies to any number of components. Only separable blend modes can be used when blending spot colors.

Some of the separable blend modes are defined by actual mathematical formulas; the rest are defined only by a description of their intended effects.

TABLE 2 Separable blend modes

NAME	RESULT
Normal	$\text{Normal}(cb, cs) = cs$ Replaces the backdrop color by the source color.
Multiply	$\text{Multiply}(cb, cs) = cb \cdot cs$ Multiples the backdrop and source color values. The result color is always at least as dark as either of the two argument colors. Multiplying any color with black produces black. Multiplying any color with white leaves the color unchanged. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.
Screen	$\text{Screen}(cb, cs) = cb + cs - cb \cdot cs = 1 - (1 - cb) \cdot (1 - cs)$ Multiplies the complements of the backdrop and source color values. The result color is always at least as light as either of the two argument colors. Screening with black leaves the color unchanged. Screening with white produces white. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.
Overlay	Multiplies or screens the colors, depending on the backdrop color. Source colors overlay the backdrop while preserving the highlights and shadows of the backdrop. The backdrop color is not replaced

	but is mixed with the source color to reflect the lightness or darkness of the backdrop color.
SoftLight	Darkens or lightens the colors, depending on the source color value. The effect is similar to shining a diffused spotlight on the backdrop. If the source color is lighter than 0.5, the backdrop is lightened, as if it were dodged. This is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it were burned in. Painting with pure black or white produces a distinctly darker or lighter area but does not result in pure black or white.
HardLight	Multiplies or screens the colors, depending on the source color value. The effect is similar to shining a harsh spotlight on the backdrop. If the source color is lighter than 0.5, the backdrop is lightened, as if it were screened. This is useful for adding highlights to a scene. If the source color is darker than 0.5, the backdrop is darkened, as if it were multiplied. This is useful for adding shadows to a scene. Painting with pure black or white produces pure black or white.
ColorDodge	Brightens the backdrop color to reflect the source color. Painting with black produces no change.
ColorBurn	Darkens the backdrop color to reflect the source color. Painting with white produces no change.
Darken	$\mathrm{Darken}(cb, cs) = \min(cb, cs)$ Selects the darker of the backdrop and source colors. The backdrop is replaced with the source where the source is darker; otherwise it is left unchanged.
Lighten	$\mathrm{Lighten}(cb, cs) = \max(cb, cs)$ Selects the lighter of the backdrop and source colors. The backdrop is replaced with the source where the source is lighter; otherwise it is left unchanged.
Difference	$\mathrm{Difference}(cb, cs) = \lvert cb - cs \rvert$ Subtracts the source color from the backdrop color or the backdrop color from the source color, depending on which has the greater

	brightness value. Painting with white inverts the backdrop color; painting with black produces no change.
Exclusion	Produces an effect similar to but lower in contrast than the Difference mode. Painting with white inverts the backdrop color; painting with black produces no change.

Table 3 lists a standard set of non-separable blend modes. Their effects are described, but no mathematical formulas are given. These modes all entail conversion to and from an intermediate hue, saturation, and luminance representation. Since the non-separable blend modes consider all color components in combination, their computation depends on the blending color space in which those colors are interpreted.

TABLE 3 Non-separable blend modes

NAME	RESULT
Hue	Creates a color with the luminance and saturation of the backdrop color and the hue of the source color.
Saturation	Creates a color with the luminance and hue of the backdrop color and the saturation of the source color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.
Color	Creates a color with the luminance of the backdrop color and the hue and saturation of the source color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images and for tinting color images.
Luminosity	Creates a color with the hue and saturation of the base color and the luminance of the blend color. This produces an inverse effect from that of the Color mode.

Note: *For illustrations of the visual effects of the blend modes, see the Adobe Photoshop User Guide.*

3.3 Interpretation of Alpha

The color compositing function (repeated below) produces a result color that is a weighted average of the source color, the backdrop color, and the $B(Cb, Cs)$ term, with the weighting controlled by the source and backdrop alphas.

$$Cr = \left(1 - \frac{\alpha s}{\alpha r}\right) \cdot Cb + \frac{\alpha s}{\alpha r} \cdot ((1 - \alpha b) \cdot Cs + \alpha b \cdot B(Cb, Cs))$$

The simplest blend mode, **Normal**, is defined by $B(Cb, Cs) = Cs$. With this blend mode, the compositing formula collapses to a simple weighted average of the source and backdrop colors, controlled by the source and backdrop alpha values.

If the blend mode is a more interesting function of the source and backdrop colors, the source and backdrop alphas control whether the effect of the blend mode is fully realized or is toned down by mixing the result with the source and backdrop colors. With any blend mode, $\alpha s = 0$ or $\alpha b = 0$ results in no blend mode effect; $\alpha s = 1$ and $\alpha b = 1$ results in maximum blend mode effect.

Mathematically, the influence of the source and backdrop colors is controlled by the source and backdrop alphas, respectively. The influence of the blend function is controlled by the product of the source and backdrop alphas.

Another variable, the result alpha, also appears in the function. This is actually a computed result, described in Section 4, “Shape and Opacity Computations.” The result color is normalized by the result alpha. This ensures that when this color and alpha are subsequently used together in another compositing operation, the color’s contribution will be correctly represented. Note that if the result alpha is zero, the result color is undefined.

The above formula is a simplification of the following one, which presents the relative contributions of backdrop, source, and blended colors in a more straightforward fashion:

$$\alpha r \cdot Cr = (1 - \alpha s) \cdot \alpha b \cdot Cb + (1 - \alpha b) \cdot \alpha s \cdot Cs + \alpha b \cdot \alpha s \cdot B(Cb, Cs)$$

The simplification requires a substitution based on the alpha compositing formula, which is presented in the next section.

4 Shape and Opacity Computations

So far, we have covered the generation of the color that results when a source color is composited with a backdrop color. This section describes the derivation of the alpha values that control the compositing process.

As indicated earlier, alpha is actually a combination of shape and opacity; it is defined simply to be their product. Thus, we define:

$$\begin{aligned}\alpha b &= fb \cdot qb \\ \alpha r &= fr \cdot qr \\ \alpha s &= fs \cdot qs\end{aligned}$$

We now describe the various shape and opacity values individually. Once again, keep in mind that conceptually these values are computed for every point on the page.

4.1 Source Shape and Opacity

The shape and opacity values can come from several sources. The transparency model defines three independent sources for each. However, the PDF representation imposes some limitations on the ability to specify all of these sources independently.

- *Object shape*. Elementary objects, such as strokes, fills, and text, have an intrinsic shape, whose value is 1 for points inside the object and 0 outside. Similarly, a masked image with a binary mask (as in PDF 1.3) has a shape that is 1 in the unmasked portions and 0 in the masked portions. The shape of a group object is the union of the shapes of the objects it contains.

  Note: *Mathematically, elementary objects have "hard" edges, with shape value either 0 or 1 at any given point. However, when such objects are rasterized to device pixels, the shape values along the boundaries may take on fractional values, representing fractional coverage of those pixels. When such anti-aliasing is performed, it is important to treat the fractional coverage as shape, not as opacity.*

- *Mask shape*. There can be an additional source of shape values varying by position, independent of the object itself. (How such a mask might be generated is discussed in Section 6, "Soft Masks.") Using such a mask to modify the shape of some object or group is called *soft clipping*. It can produce effects such as a gradual transition between an object and its backdrop, as in a vignette.

- *Constant shape*. This is a scalar value that simply modifies the source shape value at every point. The constant shape is just a convenience, since its effect could be simulated with a mask that has the same value everywhere.
- *Object opacity.* Elementary objects have an opacity of 1 everywhere. The opacity of a group object is the result of the opacity computations for all the objects it contains.
- *Mask opacity.* This is an additional source of opacity values varying by position, independent of the object itself.
- *Constant opacity.* This is a scalar value that simply modifies the source opacity value at every point. It is useful to think of this value as the "current opacity," analogous to the "current color," used when painting elementary objects.

The range of all of the above shape and opacity inputs is from 0 to 1, and the default value for all of them is 1. The intent is that any of the inputs described above will make the painting operation more transparent as it goes towards 0. If more than one input goes towards 0 then the result is compounded. This is achieved mathematically simply by multiplying the three inputs of each type, producing intermediate values called the *source shape* and the *source opacity.*

$$\begin{aligned} fs &= fj \cdot fm \cdot fk \\ qs &= qj \cdot qm \cdot qk \end{aligned}$$

where the variable definitions are given in Table 4.

TABLE 4 Variables in the source shape and opacity computations

VARIABLE	MEANING
fj	Object shape
fk	Constant shape
fm	Mask shape
fs	Source shape
qj	Object opacity

qk	Constant opacity
qm	Mask opacity
qs	Source opacity

Note: *When an object is painted with a pattern, the shape and opacity for points in the object's interior are determined by the shape and opacity of corresponding points in the pattern instead of being 1 everywhere (see Section 9.14).*

4.2 Computing the Result Shape and Opacity

In parallel with computing a result color, the painting operation also computes a *result shape* and a *result opacity* value. These values define the shape and opacity associated with the result color.

Compositing of shape and opacity values is done by the *union function*:

$$\text{Union}(b, s) = 1 - (1 - b) \cdot (1 - s) = b + s - b \cdot s$$

where b and s are the backdrop and source values to be composited. This can be thought of as an "inverted multiply": it is just a multiply with the inputs and outputs complemented. The result tends toward 1; if either input is 1 then the result is 1. This is a generalization of the conventional concept of "union" for opaque shapes.

The result shape and opacity are given by:

$$fr = \text{Union}(fb,\ fs)$$
$$qr = \frac{\text{Union}(fb \cdot qb,\ fs \cdot qs)}{fr}$$

where the variable definitions are given in Table 5.

TABLE 5 Variables in the result opacity computation

VARIABLE	MEANING
fb	Backdrop shape
fr	Result shape

fs	Source shape
qb	Backdrop opacity
qr	Result opacity
qs	Source opacity

These formulas can be interpreted as follows:

- The result shape is simply the union of the backdrop and source shapes.
- The result opacity is the union of the backdrop and source opacities, each of whose contribution is determined by its respective shape. The result is then normalized by the result shape. This ensures that when this shape and opacity are subsequently used together in another compositing operation, the opacity's contribution will be correctly represented.

Since alpha is just the product of shape and opacity, it can easily be shown that

$$\alpha r = \text{Union}(\alpha b, \alpha s)$$

This formula can be used whenever the independent shape and opacity results are not needed.

4.3 Summary of Compositing Computations

Below is a summary of all the computations from the previous section and this one. They are given in an order such that no variable is used before it is computed; also, some of the formulas have been rearranged to simplify them.

$$\text{Union}(b, s) = 1 - (1 - b) \cdot (1 - s) = b + s - b \cdot s$$

$$fs = fj \cdot fm \cdot fk$$

$$qs = qj \cdot qm \cdot qk$$

$$fr = \text{Union}(fb, fs)$$

$$\alpha b = fb \cdot qb$$

$$\alpha s = fs \cdot qs$$

$$\alpha r = \text{Union}(\alpha b, \alpha s)$$

$$qr = \frac{\alpha r}{fr}$$

$$Cr = \left(1 - \frac{\alpha s}{\alpha r}\right) \cdot Cb + \frac{\alpha s}{\alpha r} \cdot ((1 - \alpha b) \cdot Cs + \alpha b \cdot B(Cb, Cs))$$

For a list of the variables used in these formulas, see the tables in the preceding sections; the information is summarized in Table 13 on page 216.

5 Groups

A *group* is a sequence of consecutive objects in a stack that are collected together and composited to produce a single color, shape, and opacity at each point. The result is then treated as if it were a single object for subsequent compositing operations. This facilitates creating independent pieces of artwork, each composed of many objects, and then combining them, possibly with additional transparency effects during the combination. Groups can be nested within other groups; this is a strict nesting, so that the group hierarchy forms a tree structure.

The objects contained within a group are treated as a separate stack, called the *group's stack*. The objects in the stack are composited against some initial backdrop (discussed later), producing a composite color, shape, and opacity for the group as a whole. The result is an object whose shape is the union of the shapes of all constituent objects and whose color and opacity are the result of the compositing operations. This object is in turn composited with the group's backdrop in the usual way.

In addition to the computed color, shape, and opacity, the group as a whole can have several additional attributes:

- All of the input variables that affect the compositing computation for an object can also be applied when compositing the group with its backdrop. These include mask and constant shape, mask and constant opacity, and blend mode.
- The group can be *isolated* or *non-isolated*, determining the initial backdrop against which the group's stack is composited.
- The group can be *knockout* or *non-knockout*, determining whether the objects within the group's stack are composited with one another, or only with the group's backdrop.
- An isolated group can specify its own blending color space, independent of the blending color space of the group's backdrop.
- Instead of being composited onto the current page, a group's results can be used as a source of shape or opacity values for creating a *soft mask*, described in Section 6, "Soft Masks."

The next section introduces some new notation for dealing with group compositing. The following section describes the group compositing function for a non-isolated, non-knockout group. Subsequent sections describe the special properties of groups having the isolated and knockout attributes.

5.1 Notation

Since we are now dealing with multiple objects at a time, it is useful to have some notation for distinguishing among them. Accordingly, we alter the variables used earlier to include a subscript that indicates the associated object's position in the stack. The subscript 0 indicates the initial backdrop; subscripts 1 through n indicate the bottommost through topmost objects in an n-element stack; subscript i indicates the object that is currently of interest. Additionally, we drop the b and r suffixes from the variables αb, Cb, fb, qb, αr, Cr, fr and qr; other variables retain their suffixes.

This convention permits the compositing formulas to be restated as recurrence relations among elements of a stack. For instance, the result of the color compositing computation for object *i* is denoted by C_i (previously *Cr*). This computation takes as one of its inputs the immediate backdrop color, which is the result of the color compositing computation for object $i - 1$; this is denoted by C_{i-1} (previously *Cb*).

The revised formulas for a simple stack (not including any groups) are:

$$fs_i = fj_i \cdot fm_i \cdot fk_i$$

$$qs_i = qj_i \cdot qm_i \cdot qk_i$$

$$f_i = \text{Union}(f_{i-1}, fs_i)$$

$$\alpha s_i = fs_i \cdot qs_i$$

$$\alpha_i = \text{Union}(\alpha_{i-1}, \alpha s_i)$$

$$q_i = \frac{\alpha_i}{f_i}$$

$$C_i = \left(1 - \frac{\alpha s_i}{\alpha_i}\right) \cdot C_{i-1} + \frac{\alpha s_i}{\alpha_i} \cdot ((1 - \alpha_{i-1}) \cdot Cs_i + \alpha_{i-1} \cdot B_i(C_{i-1}, Cs_i))$$

where the variable definitions are given in Table 6. Compare these with the formulas summarized in Section 4.3, "Summary of Compositing Computations."

TABLE 6 Revised variables for basic compositing computations

VARIABLE	MEANING
αs_i	Source alpha
α_i	Result alpha (after compositing object *i*)
$B_i(C_{i-1}, Cs_i)$	Function implementing the blend mode
Cs_i	Source color
C_i	Result color (after compositing object *i*)
fj_i	Object shape
fk_i	Constant shape
fm_i	Mask shape
fs_i	Source shape

f_i	Result shape (after compositing object *i*)
qj_i	Object opacity
qk_i	Constant opacity
qm_i	Mask opacity
qs_i	Source opacity
q_i	Result opacity (after compositing object *i*)

5.2 Group Structure and Nomenclature

As indicated earlier, the elements of a group are treated as a separate stack, the group's stack. Those objects are composited (against a selected initial backdrop, to be described), and the resulting color, shape, and opacity are then treated as if they belonged to a single object. The resulting object is in turn composited with the group's backdrop in the usual way.

This manipulation entails interpreting the stack as a tree. For an *n*-element group that begins at position *i* in the stack, it treats the next *n* objects as an *n*-element substack, whose elements are given an independent numbering of 1 through *n*. Those objects are removed from the object numbering in the containing stack. They are replaced by the group object, numbered *i*, followed by the remaining objects that are painted on top of the group, renumbered starting at $i + 1$. This operation applies recursively to any nested groups. Henceforth, the term *element* (denoted E_i) refers to a member of some group; it can itself be either an object or a group.

From the perspective of a particular element in a nested group, there exist three different backdrops that are interesting to talk about:

- *Group backdrop*—the result of compositing all elements up to but not including the first element of the group. (This definition is altered if the parent group is a knockout group; see Section 5.5, "Knockout Groups.")
- *Initial backdrop*—a backdrop that is selected for compositing the group's first element. This is either the same as the group backdrop (non-isolated group) or a fully transparent backdrop (isolated group).
- *Immediate backdrop*—the result of compositing all elements of the group up to but not including the current element of interest.

When all elements of a group have been composited, the result is treated as if the group were a single object, which is then composited with the group backdrop. (Note that this operation occurs regardless of whether the group backdrop or a transparent backdrop was chosen as the initial backdrop for compositing the elements of the group. There is a special correction to ensure that the backdrop's contribution to the overall result is applied only once.)

5.3 Group Compositing Computations

The color and opacity of a group are defined by the *group compositing function*:

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

where the variable definitions are given in Table 7.

TABLE 7 Arguments and results of group compositing function

VARIABLE	MEANING
α_0	Alpha of the group's backdrop
α	Computed alpha of the group, to be used as the object alpha when the group itself is treated as an object
C_0	Color of the group's backdrop
C	Computed color of the group, to be used as the source color when the group itself is treated as an object
f	Computed shape of the group, to be used as the object shape when the group itself is treated as an object
G	The group: a compound object consisting of all the elements $E_1 \ldots E_n$ of the group—the n constituent objects' colors, shapes, opacities, and blend modes

Note that the opacity is not given explicitly as an argument or result of this function. When needed, the opacity can be computed by dividing the alpha by the associated shape. Almost all of the computations use the product of shape and opacity rather than opacity by itself, so it is usually convenient to keep track of shape and alpha, rather than shape and opacity.

The result of calling the Composite function is then treated as if it were an object, which is composited with the group's backdrop according to the normal formulas. In those formulas, the returned color C is treated as the source color Cs; the returned shape and alpha, f and α, are treated as the object shape and alpha, fj and αj.

The definition of the Composite function (for a non-isolated, non-knockout group) is as follows:

Initialization:

$$fg_0 = \alpha g_0 = 0$$

For each group element E_i in G, $i \in [1, n]$:

$$\langle Cs_i, fj_i, \alpha j_i \rangle = \begin{cases} \text{Composite}(C_{i-1}, \alpha_{i-1}, E_i) \text{ if } E_i \text{ is a group} \\ \text{intrinsic color, shape, and (shape} \cdot \text{opacity) of } E_i \text{ otherwise} \end{cases}$$

$$fs_i = fj_i \cdot fk_i \cdot fm_i$$

$$\alpha s_i = \alpha j_i \cdot (fk_i \cdot qk_i) \cdot (fm_i \cdot qm_i)$$

$$fg_i = \text{Union}(fg_{i-1}, fs_i)$$

$$\alpha g_i = \text{Union}(\alpha g_{i-1}, \alpha s_i)$$

$$\alpha_i = \text{Union}(\alpha_0, \alpha g_i)$$

$$C_i = \left(1 - \frac{\alpha s_i}{\alpha_i}\right) \cdot C_{i-1} + \frac{\alpha s_i}{\alpha_i} \cdot ((1 - \alpha_{i-1}) \cdot Cs_i + \alpha_{i-1} \cdot B_i(C_{i-1}, Cs_i))$$

Result:

$$C = C_n + (C_n - C_0) \cdot \left(\frac{\alpha_0}{\alpha g_n} - \alpha_0\right)$$

$$f = fg_n$$

$$\alpha = \alpha g_n$$

where the variable definitions are given in Table 8 (in addition to the ones in Table 7).

TABLE 8 Variables in the group compositing function

VARIABLE	MEANING
αs_i	Source alpha
αg_i	Group alpha: the accumulated source alphas of group elements E_1 through E_i only, excluding the initial backdrop α_0
αj_i	Object alpha for E_i,—the product of the object shape and object opacity. This is an intrinsic attribute of an elementary object (one that isn't a group); it is a computed result for a group.
α_i	Accumulated alpha (after compositing object *i*), including the initial backdrop α_0
$B_i(C_{i-1}, Cs_i)$	Function implementing the blend mode for E_i
Cs_i	Source color for E_i. This is an intrinsic attribute of an elementary object; it is a computed result for a group.
C_i	Accumulated color (after compositing object *i*), including the initial backdrop
E_i	Element *i* of the group. This is a compound variable representing the color, shape, opacity, and blend mode parameters that either are intrinsic to the object or are associated input variables.
fj_i	Object shape for E_i. This is an intrinsic attribute of an elementary object (one that isn't a group); it is a computed result for a group.
fk_i	Constant shape for E_i
fm_i	Mask shape for E_i
fs_i	Source shape
fg_i	Group shape: the accumulated source shapes of group elements E_1 through E_i only, excluding the initial backdrop
qk_i	Constant opacity for E_i
qm_i	Mask opacity for E_i

As stated above, E_i is a compound variable representing an element of a group. If the element is itself a group, it represents all the elements of that group. When Composite is called with E_i as an argument, this means to pass the entire group that E_i represents. This group is represented by the G variable inside the recursive call to Composite; it is expanded and its elements are denoted by $E_1 \ldots E_n$.

Note that the elements of a group are composited onto a backdrop that includes the group's initial backdrop. This is to achieve the correct effects of the blend modes, most of which are dependent on both the source and backdrop colors being blended. (This feature is what distinguishes a non-isolated group from an isolated group, discussed in the next section.)

Special attention should be directed to the formulas at the end that compute the C, f, and α results that are returned from the Composite function. Essentially, they remove the contribution of the group backdrop from the computed results. This ensures that when the group itself is subsequently composited with that backdrop (possibly with additional shape or opacity inputs or a different blend mode), the backdrop's contribution is included only once.

For color, the backdrop removal is accomplished by an explicit calculation, whose effect is essentially the reversal of compositing with **Normal** blend mode. The formula is a simplification of the following formulas that present this operation more intuitively:

$$bf = \frac{(1 - \alpha g_n) \cdot \alpha_0}{\text{Union}(\alpha_0, \alpha g_n)}$$

$$C = \frac{C_n - bf \cdot C_0}{1 - bf}$$

where bf is the backdrop fraction, that is, the relative contribution of the backdrop color to the overall color.

For shape and alpha, backdrop removal is accomplished by maintaining two sets of variables to hold the accumulated values. The group shape and alpha, fg_i and αg_i, accumulate only the shape and alpha of the group elements, excluding the group backdrop; their final values become the group results returned by Composite. The complete alpha, α_i, includes the backdrop contribution as well; its value used in the color compositing computations. (There is never any need to compute the corresponding complete shape, f_i, that includes the backdrop contribution.)

As a result of these corrections, the effect of compositing objects as a group is the same as compositing them separately, without grouping, if all of the following conditions are true:

- The group is non-isolated and has the same knockout attribute as its parent group (see the next two sections).
- When compositing the group's results with the group backdrop, the Normal blend mode is used and the shape and opacity inputs are always 1.

5.4 Isolated Groups

An isolated group is one whose elements are composited onto a fully transparent initial backdrop rather than onto the group's backdrop. The resulting source color, object shape, and object alpha for the group are therefore independent of the group backdrop. The only interaction with the group backdrop occurs when the group's computed color, shape, and alpha are then composited with it.

In particular, the special effects produced by the blend mode of objects within the group take into account only the intrinsic colors and opacities of those objects; they are not influenced by the group's backdrop. For example, applying the **Multiply** blend mode to an object in the group will produce a darkening effect upon other objects lower in the group's stack, but it won't produce that effect on the group's backdrop.

The effect of an isolated group can be represented by a simple object that directly specifies a color, shape, and opacity at each point. This so-called "flattening" of an isolated group is sometimes useful for import and export of fully-composited artwork in applications. Additionally, a group that specifies an explicit blending color space must be an isolated group.

For an isolated group, the group compositing function is altered simply by adding one statement to the initialization:

If the group is isolated: $\alpha_0 = 0$

That is, the initial backdrop on which the elements of the group are composited is transparent, rather than being inherited from the group's backdrop. This substitution also makes C_0 undefined, but the normal compositing formulas take care of that. Additionally, the result computation for C automatically simplifies to $C = C_n$, since there is no backdrop contribution to be factored out.

5.5 Knockout Groups

In a knockout group, each individual element is composited with the group's initial backdrop, rather than with the stack of preceding elements in the group. When objects have binary shapes (1 for "inside," 0 for "outside"), each object "knocks out" the effects of any overlapping elements within the same group. At any given point, only the topmost object enclosing the point contributes to the result color and opacity of the group as a whole.

This model is similar to the opaque painting model of the existing Adobe imaging model, except that the "topmost object wins" rule applies to both the color and the opacity. Knockout groups are useful in composing a piece of artwork from a collection of overlapping objects, where the topmost object in any overlap completely obscures the objects underneath. At the same time, the topmost object interacts with the group's initial backdrop in the usual way, with its opacity and blend mode applied as appropriate.

The concept of "knockout" is generalized to accommodate fractional shape values. In that case, the immediate backdrop is only partially knocked out and replaced by only a fraction of the result of compositing the object with the initial backdrop.

The restated group compositing function deals with knockout groups by introducing a new variable, b, which is a subscript that specifies which previous result to use as the backdrop in the compositing computations: 0 in a knockout group; $i - 1$ in a non-knockout group. When $b = i - 1$, the formulas simplify to the ones given in Section 5.3, "Group Compositing Computations."

In the general case, the computation proceeds in two stages:

1. Composite the object with the group's initial backdrop, but disregarding the object's shape and using a source shape value of 1 everywhere. This produces unnormalized temporary alpha and color results, αt and Ct. (For color, this computation is essentially the same as the unsimplified color compositing for-

mula given in Section 3.3, “Interpretation of Alpha,” but using a source shape of 1.)

$$\alpha t = \text{Union}(\alpha g_b, qs_i)$$
$$Ct = (1 - qs_i) \cdot \alpha_b \cdot C_b + qs_i \cdot ((1 - \alpha_b) \cdot Cs_i + \alpha_b \cdot B_i(C_b, Cs_i))$$

2. Compute a weighted average of this result with the object’s immediate backdrop, using the source shape as the weighting factor. Then normalize the result color by the result alpha.

$$\alpha g_i = (1 - fs_i) \cdot \alpha g_{i-1} + fs_i \cdot \alpha t$$
$$\alpha_i = \text{Union}(\alpha_0, \alpha g_i)$$
$$C_i = \frac{(1 - fs_i) \cdot \alpha_{i-1} \cdot C_{i-1} + fs_i \cdot Ct}{\alpha_i}$$

This averaging computation is performed for both color and alpha. The above formulas show this averaging directly. The formulas given below in Section 5.6, “Summary of Group Compositing Computations,” are slightly altered to use source shape and alpha, rather than source shape and opacity, avoiding the need to compute a source opacity value explicitly. (Note that *Ct* there is slightly different from *Ct* above; it is premultiplied by fs_i.)

The extreme values of the source shape produce the straightforward “knockout” effect. That is, a shape of 1 (“inside”) yields the color and opacity that result from compositing the object with the initial backdrop. A shape of 0 (“outside”) leaves the previous group results unchanged. The existence of the knockout feature is the main reason for maintaining a separate shape value, rather than only a single alpha value that combines shape and opacity. The separate shape value must be computed in any group that is subsequently used as an element of a knockout group.

A knockout group can be non-isolated or isolated; that is, *isolated* and *knockout* are independent attributes of a group. A non-isolated knockout group composites its topmost enclosing element with the group’s backdrop; an isolated knockout group composites the element with a transparent backdrop.

Note: *When a non-isolated group is nested within a knockout group, the initial backdrop of the inner group is the same as the initial backdrop of the outer* *group; it is not the immediate backdrop of the inner group. This non-obvious nesting behavior is a consequence of the formulas for Composite when b = 0.*

5.6 Summary of Group Compositing Computations

The following restatement of the group compositing function also takes isolated groups and knockout groups into account. See Table 7 on page 166 and Table 8 on page 168 for the definitions of the variables.

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

Initialization:

$$fg_0 = \alpha g_0 = 0$$

If the group is isolated: $\alpha_0 = 0$

For each group element E_i in G, $i \in [1, n]$:

$$b = \begin{cases} 0 \text{ if the group is knockout} \\ i - 1 \text{ otherwise} \end{cases}$$

$$\langle Cs_i, fj_i, \alpha j_i \rangle = \begin{cases} \text{Composite}(C_b, \alpha_b, E_i) \text{ if } E_i \text{ is a group} \\ \text{intrinsic color, shape, and (shape} \cdot \text{opacity) of } E_i \text{ otherwise} \end{cases}$$

$$fs_i = fj_i \cdot fk_i \cdot fm_i$$

$$\alpha s_i = \alpha j_i \cdot (fk_i \cdot qk_i) \cdot (fm_i \cdot qm_i)$$

$$fg_i = \text{Union}(fg_{i-1}, fs_i)$$

$$\alpha g_i = (1 - fs_i) \cdot \alpha g_{i-1} + (fs_i - \alpha s_i) \cdot \alpha g_b + \alpha s_i$$

$$\alpha_i = \text{Union}(\alpha_0, \alpha g_i)$$

$$Ct = (fs_i - \alpha s_i) \cdot \alpha_b \cdot C_b + \alpha s_i \cdot ((1 - \alpha_b) \cdot Cs_i + \alpha_b \cdot B_i(C_b, Cs_i))$$

$$C_i = \frac{(1 - fs_i) \cdot \alpha_{i-1} \cdot C_{i-1} + Ct}{\alpha_i}$$

Result:

$$C = C_n + (C_n - C_0) \cdot \left(\frac{\alpha_0}{\alpha g_n} - \alpha_0\right)$$

$$f = f g_n$$

$$\alpha = \alpha g_n$$

Note: *Once again, keep in mind that these formulas are in their most general form. They can be significantly simplified when some sources of shape and opacity are not present or when shape and opacity need not be maintained separately. Furthermore, in each specific type of group (isolated or not, knockout or not), some terms of these equations cancel or drop out. An efficient implementation should use the derived simplified equations.*

5.7 Page Group

All of the elements painted directly onto a page—both top-level groups and top-level objects that are not part of any group—are treated as if they were contained in a group *P*, which in turn is composited with a context-dependent backdrop. This group is called the *page group*.

The page group can be treated in two distinctly different ways:

- Ordinarily, the page group is imposed directly on media, such as paper or a display screen. The page group is treated as an isolated group, whose results are then composited with a backdrop color appropriate for the media. The backdrop is nominally white, though varying according to actual properties of the media. However, some applications may choose to provide a different backdrop, such as a checkerboard that is useful for visualizing the effects of transparency in the artwork.
- A “page” of a PDF file can be treated as a graphics object that is to be used as an element of a page of some other document. This case arises, for example, when placing a PDF file containing a piece of artwork produced by Illustrator into a page layout produced by InDesign™. In this situation, the PDF “page” is not composited with the media color; instead, it is treated as an ordinary transparency group, which can be either isolated or non-isolated and is composited with its backdrop in the normal way.

The remainder of this section pertains only to the first use of the page group, where it is to be imposed directly on the media.

The color *C* of the page at a given point is defined by a simplification of the general group compositing formula:

$$\langle Cg,\ fg, \alpha g\rangle \ = \ \text{Composite}(U, 0, P)$$
$$C \ = \ (1 - \alpha g) \cdot W + \alpha g \cdot Cg$$

where the variable definitions are given in Table 9. The first formula computes the color and alpha for the group given a transparent backdrop—in effect, treating *P* as an isolated group. The second formula composites the results with the context-dependent backdrop (using the equivalent of **Normal** blend mode).

TABLE 9 Variables for page group computation

VARIABLE	MEANING
αg	Computed alpha of the group
C	Color of the page
Cg	Computed color of the group
fg	Computed shape of the group (this result is discarded)
P	A group consisting of all elements $E_1 \ldots E_n$ in the page's top-level stack
U	An undefined color (which is not used, since the α_0 argument of Composite is 0)
W	The initial color of the page, which is nominally white but can be context-dependent depending on properties of the media or the needs of the application.

If not otherwise specified, the page group's color space is inherited from the native color space of the output device—that is, a device color space, such as **DeviceRGB** or **DeviceCMYK**. It is often preferable to specify an explicit color space, particularly a CIE-based color space, to ensure more predictable results of the compositing computations within the page group. In this case, all page-level compositing is done in that color space, with the entire result then converted to the native color space of the output device before being composited with the context-dependent backdrop. This case also arises when the page is not actually being rendered but is converted to a “flattened” representation in an opaque imaging model, such as PostScript.

6 Soft Masks

Earlier sections have mentioned the mask shape, *fm*, and mask opacity, *qm*, that are contributors to the source shape and opacity. These enable shape and opacity to originate from a source that is independent of the objects being composited. A soft mask (or just mask for short) defines values that can vary across different points on the page. The word “soft” emphasizes that the mask value at a given point is not just 0 or 1 but can take on fractional values.

A mask used as a source of shape values is also called a soft clip. The term *soft clip* arises from analogy with the “hard” clipping path of the existing Adobe imaging model. The soft clip is a generalization of the hard clip: a hard clip can be represented as a soft clip having shape value 1 inside and 0 outside the clipping path. Everywhere inside a hard clipping path, a source object’s color replaces the backdrop; everywhere outside, the backdrop shows through unchanged. With a soft clip, on the other hand, one can create a gradual transition between an object and the backdrop, such as in a vignette. (Although “soft clip” suggests a vignette around an enclosed area, in fact the value of the mask can be an arbitrary function of position.)

A mask is typically the only means of providing position-dependent opacity, since elementary objects do not have intrinsic opacity.

A mask can be defined by creating a group and then painting objects into it, thereby defining color, shape, and opacity in the usual way. The resulting group can then be treated as a mask by following one of the two procedures described below.

6.1 Mask from group alpha

The color, shape, and opacity of the group *G* are first computed by the usual formula:

$$\langle C, f, \alpha \rangle = \text{Composite}(C_0, \alpha_0, G)$$

where C_0 and α_0 are an arbitrary backdrop whose value does not contribute to the eventual result. The C, f, and α results are the group's color, shape, and alpha, with the backdrop factored out.

The mask value at each point is then derived from the alpha of the group. In this case, the group's color is not used, so there is no need to compute it. The alpha value is passed through a separately-specified transfer function, enabling the masking effect to be customized.

6.2 Mask from group luminosity

The group is composited with a fully-opaque backdrop of some selected color. The mask value at any given point is then defined to be the luminosity of the resulting color. This enables the mask to be derived from the shape and color of an arbitrary piece of artwork, drawn with ordinary painting operators.

The color C used to create the mask from a group G is defined by:

$$\langle Cg,\ fg,\ \alpha g\rangle \;=\; \text{Composite}(C_0, 1, G)$$
$$C \;=\; (1 - \alpha g)\cdot C_0 + \alpha g \cdot Cg$$

where C_0 is the selected backdrop color.

G can be any kind of group—isolated or not, knockout or not—with various effects in the C result produced in each case. The color C is then converted to luminosity in one of the following ways, depending on the group's color space:

- For CIE-based spaces, convert to CIE XYZ and use the Y component as luminosity. This procedure produces a colorimetrically correct luminosity. In the case of a PDF **CalRGB** space, the formula is:

  $$Y \;=\; Y_A \cdot A^{G_R} + Y_B \cdot B^{G_G} + Y_C \cdot C^{G_B}$$

  using components of the **Gamma** and **Matrix** entries of the color space dictionary (see *PDF Reference*, Table 4.14 [7.28]). An analogous computation applies to other CIE-based color spaces.
- For device color spaces, convert the color to **DeviceGray** by device-dependent means and use the resulting gray value as luminosity, with no compensation for gamma or other color calibration. This method makes no pretense of colori-

metric correctness; it merely provides a numerically simple means to produce contone mask values. Here are some recommended formulas for converting from **DeviceRGB** and **DeviceCMYK**, respectively:

$$Y = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B$$
$$Y = 0.3 \cdot (1 - C) \cdot (1 - K) + 0.59 \cdot (1 - M) \cdot (1 - K) + 0.11 \cdot (1 - Y) \cdot (1 - K)$$

Following this conversion, the result is then passed through a separately-specified transfer function, enabling the masking effect to be customized.

The backdrop color that is most likely to be useful is black. If the backdrop is black, any areas outside the group's shape will end up with a zero luminosity value in the resulting mask. If we view the contents of the group as a positive mask, this result matches our expectations with respect to the points that are outside the shape.

7 Color Space and Color Rendering Issues

This section describes the interactions between the transparency model and other aspects of color specification and rendering in the Adobe imaging model.

7.1 Color Spaces

A group can have either an explicitly declared color space or the color space inherited from the parent group. In either case:

1. A source object's color is converted to the group's color space if necessary.
2. The blending and compositing computations are done in that color space (see Section 3.1, "Blending Color Space").
3. The group's resulting color is interpreted as being in that color space when the group is subsequently composited with the backdrop.

Under this arrangement, we envision that all or most of a piece of artwork will be created in a single color space—most likely, the working color space of the application that generated it. Use of multiple color spaces typically will arise only when assembling independently-produced artwork onto a page. When the complete artwork is placed on a page, the conversion from the group's color space to the page's device color space will be done as the last step, without any further transparency compositing.

The transparency model does not impose any requirement that such a convention be followed. The reason for adopting it is to avoid loss of color information and introduction of errors due to performing unnecessary conversions.

Only isolated groups may have an explicitly declared color space; non-isolated groups must inherit their color space from the parent group. Use of an explicit color space in a non-isolated group would require converting colors from the backdrop's color space to the group's color space in order to perform the compositing computations. This may not be possible, since some color conversions are one-way; in any event, it would cause an excessive number of color conversions.

The choice of group color space will have significant effects on the results that are produced. In particular:

- As indicated in Section 3.1, "Blending Color Space," if the group uses a device color space, then the transparency compositing computations will produce device-dependent results.
- In order for the compositing computations to work in a device-independent way, the group's color space must be a CIE-based color space.
- A consequence of choosing a CIE-based group color space is that only CIE-based color spaces can be used to specify the colors of objects within the group. This is because conversion from device to CIE-based colors is not possible in general; the defined conversions work only in the opposite direction.
- The compositing computations and blend modes generally compute linear combinations of color component values, based on the assumption that the color component values themselves are linear. Therefore, it is usually best to choose a group color space that is linear (that is, it has a linear gamma curve). If a non-linear color space is chosen, the results are still well-defined, but the appearance may not match the user's expectations.

***Note:** In this connection, note that sRGB is a non-linear CIE-based color space, hence possibly unsuitable for use as a group color space.*

***Note:** An implementation of the transparency model is advised to use as much precision as possible to represent colors during the compositing computations* *and in the accumulated group results. More precision is needed for intermediate results than is typically used to represent either the original source data or the final rasterized results. This is to minimize accumulation of roundoff errors and to avoid additional errors that arise from the use of linear group color spaces.*

7.2 Spot Colors

The preceding discussion about color spaces has been concerned about *process colors*—that is, the colors that are produced by combinations of the device's process colorants. (Process colors are sometimes called "composite colors," but we will avoid that term here due to possible confusion with the transparency compositing operations.) Process colors may be specified directly in the device's color space (such as **DeviceCMYK**), or they may be produced by conversion from some other color space, such as a CIE-based color space (**CalRGB** or **ICCBased**). Whatever means is used to specify them, process colors are subject to conversion to and from the group's color space.

A *spot color* is an additional color component, independent of the color components that are used to produce process colors. A spot color can represent either an additional separation that is to be produced or an additional colorant that is to be applied to the composite page. The color component value for a spot color is called a *tint*, representing the concentration of the spot colorant. (Tints are conventionally represented as subtractive values, not additive.)

Spot colors are inherently device-dependent and are not always available. In the Adobe imaging model as represented in PostScript and PDF, each use of a spot color (in a **Separation** or **DeviceN** color space) is accompanied by an *alternate color space* and a function for mapping tint values into that color space. This enables the spot color to be approximated with process colors when the spot colorant is not available in the device.

Spot colors can be accommodated straightforwardly in the transparency model, except for issues relating to overprinting, discussed in Section 7.3, "Overprinting and Erasing." If an object that is an element of a group is painted with a spot color, one of two things can happen:

- The group maintains a separate color value for each spot color component, independent of the group's color space. Effectively, the spot color passes directly through the group hierarchy to the device, with no color conversions performed; however, it is still subject to blending and compositing with other objects that use the same spot color.

- The spot color is converted to its alternate color space. The resulting color is then subject to the usual compositing rules for process colors.

Only a single shape value and opacity value are maintained at each point in the computed group results; they apply to both process and spot color components. In effect, every object is considered to paint every color component that exists. Where no value has been specified for a given color component in a given object, an additive value of 1 (or subtractive tint value of 0) is assumed.

For instance, when painting an object with a color specified as **DeviceCMYK** or **ICCBased**, the process color components are painted as specified and the spot color components are painted with additive value 1. Likewise, when painting an object with a color specified as **Separation**, the named spot color is painted as specified and the other components (both process and other spot colors) are painted with additive value 1. The consequences of this are discussed in Section 7.3, "Overprinting and Erasing."

The existing Adobe imaging model also allows the process color components to be addressed individually, as if they were spot colors. For instance, one can specify a **Separation** color space named Cyan, which paints just the cyan component in a CMYK output device.

This is very difficult to extend to work in transparency groups. In general, the color components in a group are not the process colorants themselves, but are converted to process colorants only after completion of all color compositing computations for the group (and perhaps some of the group's parents as well). For instance, if the group's color space is **ICCBased**, the group has no Cyan component to be painted.

Therefore, treating a process color component as if it were a spot color is permitted only within a group that inherits the native color space of the output device. Attempting to do so in a group that specifies its own color space will result in conversion of the requested spot color to its alternate color space.

7.3 Overprinting and Erasing

This section addresses the relationship between overprinting and transparency and discusses how the Adobe imaging model is altered to deal with it. This discussion is limited to what can be represented in PDF. There is an entirely separate question of how applications can use transparency to simulate the effects of overprinting colorants; that topic is beyond the scope of this specification.

Background

PDF has an *overprint* parameter and a *nonzero overprint mode* parameter in the graphics state. They are documented in *PDF Reference*, Section 4.5.6. (The older PDF 1.3 specification lacks detailed documentation of these parameters. The overprint parameter, but not the nonzero overprint mode, also exists in PostScript; it is documented in *PostScript Language Reference, third edition*, Section 4.8.5.)

Briefly, in the existing imaging model, painting an object causes some specific set of device colorants to be marked according to the current color space and color value. The remaining colorants are either erased or left unchanged, according to whether the overprint parameter is *false* or *true*. The nonzero overprint mode parameter additionally enables this selective marking of colorants to be applied to individual components of **DeviceCMYK** according to whether the component value is zero or nonzero.

This model of overprinting is very device-dependent. It deals directly with the painting of device colorants, independent of the color space in which source colors have been specified. It primarily addresses production needs, not design intent. Overprinting is usually reserved for an opaque colorant or for a very dark color such as black. It is also invoked during late-stage production operations, such as trapping, when the actual set of device colorants has already been determined.

It is best to think of transparency as taking place in appearance space, but overprinting of device colorants in device space. This means that colorant overprint decisions should be made at output time based on the actual resultant colorants of any transparency compositing operation. On the other hand, effects similar to overprinting can be achieved in a device-independent manner by taking advantage of blend modes; this is described below.

Use of Blend Modes to Erase or Overprint

As indicated in Section 7.2, “Spot Colors,” each object paints every color component that exists—both the process color components in the group’s color space and any available spot color components. For color components whose value has not been specified, a source color of 1 is assumed. When objects are fully opaque and **Normal** blend mode is used, this has the effect of erasing those components. This is consistent with the existing opaque imaging model when the overprint flag is turned off.

The transparency model defines some blend modes, such as **Darken**, that can be used to achieve effects similar to overprinting. The definition of **Darken** is:

$$\text{Darken}(cb, cs) = \min(cb, cs)$$

If the blend mode is **Darken**, its result will always be the same as the backdrop color when the source color is 1, as it is for all unspecified color components. When the backdrop is fully opaque, painting with a source color of 1 and the **Darken** blend mode leaves the result color unchanged from the backdrop. This is consistent with the existing opaque imaging model when the overprint flag is turned on.

If the object or backdrop are not fully opaque, the above actions are altered correspondingly. That is, the “erasing” effect is reduced, and “overprinting” an object with color value 1 may affect the result color. While these results may or may not be useful, they lie outside the realm of the erasing and overprinting that are defined in the existing opaque imaging model.

When process colors are erased or overprinted (because a spot color is being painted), the blending computations described above are done componentwise in the group’s color space. If the group’s color space is different from the native color space of the output device, its components are not the actual process colorants of the output device; the blending computations affect the process colorants only after the group’s results are converted to the device color space. Thus, the effect is different from erasing or overprinting the device’s process colorants directly. On the other hand, this is a fully general operation that works uniformly, regardless of the type of object and regardless of what computations produced the source color.

The above discussion has concentrated on the color components whose values have *not* been specified and that are to be either erased or left unchanged. The **Normal** or **Darken** blend modes used for those purposes may not be suitable for use on the components whose color values *have* been specified. In particular, the **Darken** blend mode for those components would preclude overpainting a dark color with a lighter color. Moreover, some other blend mode may be specifically desired for those components.

PDF provides means to specify only one blend mode, which always applies to process colorants and sometimes applies to spot colorants as well. Specifically, only *white-preserving* blend modes can be used for spot colors—that is, functions having the property that $B(1, 1) = 1$. If a non-white-preserving blend mode is specified, it applies only to the process color components; **Normal** blend mode is substituted for the spot colors. This ensures that when objects accumulate in an isolated group, the accumulated values for unspecified components remain 1. The group's results can then be overprinted using **Darken** (or other useful modes) while avoiding unwanted interactions with the components whose values were never specified within the group.

Interpretation of Overprint Parameters

The previous section describes how effects similar to overprinting can be achieved using blend modes. Those methods do not make direct use of the overprint flag and nonzero overprint mode; they are usable only by transparency-aware applications.

PDF provides for compatibility with PDF 1.3 overprint control by defining a special CompatibleOverprint blend mode that consults the overprint control parameters to compute its result. The value of CompatibleOverprint(*Cb*, *Cs*) is either *Cb* or *Cs*, depending on the settings of the overprint parameters and the value of *Cs*. When CompatibleOverprint needs to be applied in conjunction with some other (non-**Normal**) blend mode, a nested transparency group is implicitly created. See "CompatibleOverprint Blend Mode" on page 194 for a complete explanation.

7.4 Rendering Parameters

PDF has several graphics state parameters dealing with the rendering of color: halftone, transfer functions, color rendering intent, undercolor removal, and black generation. How should these parameters work in the presence of transparency?

The problem is this: The rendering parameters can be specified on a per-object basis; they control how that object will be rendered. When all objects are opaque, it is easy to define what this means. When they are transparent, more than one object can contribute to the color at a given point. It is unclear which rendering parameters to apply in an area where transparent objects overlap. (Devising a way to “blend” the effects of these parameters seems hopelessly difficult.)

Furthermore, the operations that the rendering parameters control—halftoning in particular—can be performed only when the final color at a given point is known. When objects are transparent, rendering of an object does not occur at the time the object is specified, but at some later time. The implementation must keep track of the rendering parameters at each point from the time they are specified until the time the rendering actually occurs. In other words, rendering parameters must be associated with regions of the page rather than with individual objects.

At the same time, we would like the transparency imaging model to be compatible with the existing opaque imaging model in the case that only opaque objects are painted.

Certain of the rendering parameters—halftone and transfer function—apply only when the final color at each point on the page is known. The other parameters—color rendering intent, undercolor removal, and black generation—apply anytime colors must be converted from one color space to another. In the presence of transparency, these two classes of parameters are treated somewhat differently.

Halftone and Transfer Function

For the halftone and transfer function parameters, the problem is solved in the following way. Conceptually, there is a map over the entire page defining the rendering parameters to be used at each point. This map is defined as follows:

- Initially, the rendering parameters for the entire page have default values, which are the values in effect at the beginning of the current page.
- If the topmost object at a given point is fully opaque, then the rendering parameters associated with the object are used at that point. This provides exact com-

patibility with the existing opaque imaging model. The definitions of “topmost object” and “fully opaque” are given below.

Only elementary objects define the rendering parameters map; the rendering parameters associated with a group object are ignored. At a given point, the *topmost object* is the topmost elementary object in the entire page stack that has a nonzero object shape value (*fj*) at that point (in other words, the point is “inside” the object). An object is considered to be *fully opaque* if its source alpha is 1, its blend mode is **Normal** or **Compatible**, and each direct ancestor group likewise specifies a source alpha of 1 and a blend mode of **Normal** or **Compatible**. These conditions ensure that only the object itself contributes to the color at that point; it completely obscures the backdrop.

Rendering Intent and Color Conversions

The color rendering intent parameter needs to be handled somewhat differently. This parameter influences the conversion from a CIE-based color space to a target color space, taking the target space’s gamut into account. In an opaque imaging model, the target space is always the native color space of the output device. However, in the transparency model, the target space may be the group color space of a transparency group into which an object is being painted.

The rendering intent is needed at the moment such a conversion must be performed—that is, when painting an object (elementary or group object) having a CIE-based color into a parent group having a different color space. This is unlike the halftone and transfer function parameters, whose values are used only after all color compositing has been completed and rasterization is performed.

In all cases, the rendering intent to use for converting an object’s color is determined by the rendering intent parameter associated with that object. That is, when painting an elementary object into a group, the object’s rendering intent is used. When painting a group’s results into a parent group, the group’s rendering intent is used.

Note: *Since there may be one or more nested groups having different CIE-based color spaces, the color of an elementary source object may be converted to the device color space in multiple stages, controlled by the rendering intent in effect at each stage. The proper choice of rendering intent at each stage depends on the relative gamuts of the source and target color spaces. It is specified explicitly by* *the document creator, not prescribed by the PDF specification, since no single policy for managing rendering intents is appropriate for all situations.*

A similar approach works for the undercolor removal and black generation functions, which are applied only during conversion from **DeviceRGB** to **DeviceCMYK** color spaces.

8 Overview of PDF Extensions

The preceding sections have described the transparency model at a fairly abstract level, with relatively little mention of how it is represented in PDF. This section introduces the PDF representation; it is organized according to the presentation of the transparency model above. The next section specifies the PDF extensions for transparency in detail; it is organized according to the presentation in *PDF Reference*.

8.1 Color Compositing Computations

The object color *Cs* comes from the usual sources of color, that is, the current color in the graphics state or the source samples in an image. The backdrop color *Cb* is the result of previous painting operations.

The blending color space is an attribute of the transparency group within which an object is painted. The page as a whole is also treated as a group (the page group) that has a color space attribute. If not otherwise specified, the page group's color space is inherited from the native color space of the output device.

The blend mode *B*(*Cb*, *Cs*) is a parameter in the graphics state. It is a name selecting among a fixed enumeration of blend modes.

8.2 Shape and Opacity Computations

Every object painted by PDF painting operators has a shape value *fj* at each point, defined as follows:

- The shape of an object defined by a path (fill, stroke, text) and painted with a simple paint is always 1 inside and 0 outside the path.
- The shape of an image is nominally 1 inside and 0 outside the image rectangle. This can optionally be reduced by a binary mask accompanying the image, which is specified by either an explicit mask or a color key (using the PDF 1.3 masked image feature).
- The shape of an image mask is 1 for painted regions and 0 for masked regions.
- The shape of a shading object (painted by the **sh** operator) is 1 inside and 0 outside the bounds of the shading's painting geometry, disregarding the **Background** entry.
- The shape of an object painted with a tiling pattern is recursively determined by the objects that define the pattern.
- The shape of an object painted with a shading pattern is the intersection of the object itself and the geometry of the shading.

All elementary objects have an intrinsic opacity *qj* of 1 everywhere. Any desired opacity less than 1 must be applied by means of the *qk* and *qm* parameters described below.

The graphics state contains a constant alpha value and a position-dependent alpha mask. There is a separate parameter that specifies whether these alpha sources are to be treated as shape (*fk* and *fm*) or as opacity (*qk* and *qm*).

The constant alpha is specified by two simple scalar parameters in the graphics state: one for strokes; one for all nonstroking operations. It is reasonable to think of these parameters as the "current alpha," analogous to the current color used when painting elementary objects. (Note, however, that the nonstroking alpha is also applied when painting a group's results onto the backdrop.)

There can be at most one mask input in any compositing operation. It can be specified in either of the following ways:

- The graphics state contains a soft mask parameter. If present, its value is a *soft-mask dictionary,* which includes a transparency group that is to be used as a position-dependent source of alpha values. By this means, an arbitrary collection of objects can be used to define the shape or opacity that is then imposed on the objects painted onto the page.
- An image XObject can contain a *soft-mask image,* specified as a subsidiary image XObject. This mask, if present, overrides the PDF 1.3 masked image feature. Either form of mask in the image XObject overrides the soft mask parameter in the graphics state.

Note: *The limitation of one mask per compositing operation causes no loss of generality in the transparency model. A soft mask can also be applied to a group, and groups can be nested to multiple levels.*

8.3 Groups

A group is represented in PDF as a *transparency-group XObject*—a form XObject having some additional transparency-related attributes, which are carried in a separate *group attributes* subdictionary. The elements of the group consist of the graphics objects that are painted by execution of the XObject's content stream. The results of the group compositing computations—color, shape, and opacity—are then painted into the group's parent group or page. The **Do** operator invokes both of the above operations.

The entries in the group attributes dictionary include:

- Color space for blending and compositing, if different from the one used in the parent group or page
- Isolated and knockout boolean attributes

For a non-isolated group, the group's backdrop is defined as the computed color, shape, and opacity of everything that has been painted into the parent group or page prior to execution of the **Do** operator.

8.4 Soft Masks

A soft mask is represented by a *soft-mask dictionary.* This dictionary contains the following:

- A transparency-group XObject describing the group that is to be used as the source of position-dependent mask values
- Backdrop color and blending color space to use for the group compositing operation
- Other entries controlling conversion from the group results to mask values

A soft mask created in this way can be established as the soft mask parameter in the graphics state. This causes it to be treated as a position-dependent source of shape or opacity (*fm* or *qm*) in subsequent compositing operations.

8.5 Color Space and Color Rendering Issues

The issues and proposed model for color spaces, spot colors, and overprinting are quite thoroughly covered in Section 7, "Color Space and Color Rendering Issues." The PDF representation follows straightforwardly from that description.

8.6 Limitations in the PDF Transparency Model

The PDF realization of the general transparency model has several limitations in order to simplify the representation and the implementation. The following is a summary of these limitations, which are further described elsewhere in this specification.

- In any given transparency compositing operation, there is at most one mask input, which can be treated as either shape or opacity. (This is in addition to the shape and opacity that are intrinsic to the source object itself.)
- There are not independent parameters for constant shape and constant opacity. Instead, there is a constant alpha value, which can be treated as either shape or opacity (the other parameter is implicitly 1).
- There is only one blend mode specified per object, regardless of the number of color components that are affected when the object is painted. It always applies to the process color components. If it is a separable, white-preserving blend

mode, then it also applies to any spot color components that exist; otherwise, **Normal** blend mode is substituted for those components. There is a fixed enumeration of blend modes, with no provision for specifying an arbitrary function as a blend mode.

9 PDF Specification

This section describes the additions and changes to the PDF specification. It is organized according to the presentation in *PDF Reference, second edition*. The numbers in brackets refer to sections in the older *Portable Document Format Reference Manual, version 1.3*, where the closest equivalent material can be found.

9.1 Adobe Imaging Model—2.1.2

[This section needs to be revised to incorporate the transparency extensions to the Adobe imaging model, described here in Section 2, "Overview."]

9.2 Page Tree (Page Objects)—3.6.2 [6.4]

[Add the following entry to Table 3.17:]

Table 3.17 [6.5] Entries in a page dictionary

KEY	TYPE	VALUE
Group	dictionary	*(Optional)* A *group attributes dictionary*, which specifies the group attributes of the page group. See Table 11 on page 208 for the contents of this dictionary. See also Section 5.7, "Page Group."

9.3 Graphics Objects—4.1 [8.1]

[Replace the sentence "Each graphics object is painted...opaque painting model..." with the following:]

A PDF content stream represents a sequence of *graphics objects*. Each of these objects has a *shape*. Within the shape, every point has a *color* and an *opacity*. For some objects, the color and opacity are constant within the entire shape; for others, they can vary at different points within the shape.

In the case that all the objects are fully opaque, one can consider the objects simply to be painted onto the current page, obscuring any existing marks they may overlay. However, in the general case where transparency can occur, it is necessary to think of the objects as forming a *stack*, where the stacking order is defined to be the order in which the objects are specified, bottommost object first. All of the objects in a stack can potentially contribute to the result, according to the color, shape, and opacity compositing rules specified in Section 3, "Color Compositing Computations," and Section 4, "Shape and Opacity Computations."

[Append the following:]

These graphics objects are also treated as the elementary objects for transparency compositing purposes (subject to special treatment for text objects, described in Section 9.18). That is, all of a given object is considered to be one element of a stack. Portions of an object are not composited with one another, even if they are described in a way that would seem to cause overlaps, such as a self-intersecting path, combined fill and stroke of a path, or a shading pattern containing an overlap or fold-over.

The result of compositing a transparency-group XObject is itself treated as if it were an elementary graphics object, having shape, opacity, and color at each point. It is then composited with its parent group or page. Note that this treatment applies only to form XObjects having a **Group** entry specifying a group attributes dictionary whose **S** (Subtype) is **Transparency**. Painting an ordinary form XObject (lacking a **Group** entry) is equivalent to painting the form's constituent graphics objects individually; there is no transparency grouping behavior.

9.4 Details of Graphics State Parameters—4.3.2 [8.4]

[Add the following subsection:]

Blend Modes

The *blend mode* graphics state parameter is specified as the value of the **BM** entry in a graphics state parameter dictionary. Its value can be one of the following:

- The name of a standard blend mode, which is one of: **Compatible**, **Normal**, **Multiply**, **Screen**, **Difference**, **Darken**, **Lighten**, **ColorDodge**, **ColorBurn**, **Exclusion**, **HardLight**, **Overlay**, **SoftLight**, **Luminosity**, **Hue**, **Saturation**, and **Color**. These blend modes are explained in Section 3.2, “Blend Mode,” except for the blend mode named **Compatible**, whose meaning is the same as **Normal**.

  Note: *The **Compatible** blend mode is a relic of an earlier design. It explicitly invoked the CompatibleOverprint blend mode, described below. Compatible-Overprint is now invoked implicitly when appropriate, so it is never necessary to specify **Compatible** blend mode.*

- An array of one or more blend modes, each specified by a name as described above. This specifies alternate blend modes that can be used; the viewer will use the first blend mode that it recognizes in the array. If the viewer does not recognize any of the blend modes, it will use **Normal** mode. This allows a PDF file to use a new blend mode that has been introduced, while specifying reasonable fall-back behavior in a viewer that doesn’t recognize the new mode.

There is only one blend mode parameter in the graphics state. This blend mode always applies to process color components and sometimes applies to spot color components as well. Specifically:

- If the blend mode is separable and white-preserving, then it applies to all spot color components that exist, as well as to process color components. A *separable* blend mode is one that is performed componentwise, with no interaction between components. A *white-preserving* blend mode is a function $B(cb, cs)$ having the property that $B(1, 1) = 1$. Of the named blend modes listed above, the following are separable and white-preserving: **Compatible**, **Normal**, **Multiply**, **Screen**, **Darken**, **Lighten**, **ColorDodge**, **ColorBurn**, **HardLight**, **Overlay**, **SoftLight**.

- Otherwise, the blend mode applies only to process color components. Any spot color components are blended using **Normal** mode.

As explained in Section 7.2, “Spot Colors,” components whose values are not explicitly specified in the current color space are implicitly painted with additive value 1 (that is, subtractive tint 0). When objects accumulate in an isolated group, the accumulated values for unspecified components remain 1 so long as only white-preserving blend modes are used. This enables the group's results to be overprinted using **Darken** (or other useful modes) while avoiding unwanted interactions with the components whose values were never specified within the group.

CompatibleOverprint Blend Mode

The overprinting semantics of PDF 1.3 and earlier can be modelled as transparency with a special blend mode, as discussed in Section 7.3, "Overprinting and Erasing." There is a special blend mode called CompatibleOverprint, which implements overprinting and erasing behavior that is compatible with the existing Adobe imaging model and PDF 1.3 when painting opaque objects. (The CompatibleOverprint blend mode is never invoked explicitly; there is not any PDF name object that identifies it.)

The CompatibleOverprint blend mode is implicitly invoked whenever an elementary graphics object is painted while overprinting is enabled (that is, when the overprint flag, set by **op** or **OP** in a graphics state parameter dictionary, is *true*). If the current blend mode parameter is anything besides **Normal**, the object is treated as if it were defined in a non-isolated, non-knockout transparency group and painted using the CompatibleOverprint blend mode; the group's results are then painted using the current blend mode in the graphics state.

Note: *It is not necessary to create the implicit transparency group if the current blend mode is* ***Normal****. Simply substituting the CompatibleOverprint blend mode while painting the object produces equivalent results. There are some additional cases in which the implicit transparency group can be optimized out.*

The function CompatibleOverprint(*Cb*, *Cs*) returns either *Cb* or *Cs*, depending on the current value of the nonzero overprint mode (set by **OPM**), the value of *Cs*, and the operation that is being performed. Its definition is as follows:

- Overprinting applies only when painting elementary objects—fills, strokes, text, images, and shading objects—and only if the overprint flag in the graphics state (set by **op** or **OP**) is *true*. Otherwise, CompatibleOverprint is not invoked (but if it were, its result would be *Cs*, making it equivalent to **Normal** blend mode).

- If the overprint mode is 1 (nonzero overprint mode enabled) and the current color space and group color space are both **DeviceCMYK**, then Compatible-Overprint returns *Cb* for any **DeviceCMYK** component whose (subtractive) color value is zero; it returns *Cs* otherwise. For spot color components, CompatibleOverprint returns *Cb.*
- Otherwise, CompatibleOverprint returns *Cs* for a color component that is specified in the current color space; it returns *Cb* otherwise. For instance, if the current color space is **DeviceCMYK** or **CalRGB**, CompatibleOverprint returns *Cs* for process color components and *Cb* for spot color components. On the other hand, if the current color space is a **Separation** color space, CompatibleOverprint returns *Cs* for that spot color component and *Cb* for process colors and all other spot color components.

In the above description, "current color space" refers to the color space used for a painting operation. This may be the color space parameter in the graphics state, a color space used implicitly by operators such as **rg**, or an attribute of an image XObject. In the case of an **Indexed** space, this refers to the underlying color space; likewise for **Separation** and **DeviceN** spaces that revert to their alternate color space.

See Section 12, "Overprinting, Erasing, and Transparency," for tables giving details of the overprinting and erasing behavior described above.

9.5 Graphics State Parameter Dictionaries—4.3.4 [7.15]

[Add the following entries to table 4.8:]

Table 4.8 [7.49] Entries in a graphics state parameter dictionary

KEY	TYPE	VALUE
ca	number	*(Optional)* Constant alpha, used in the computation of the source shape or opacity for each object painted by operations other than stroke. It must be in the range 0 to 1. Default value: 1. This parameter is implicitly reset to its default value at the beginning of execution of a transparency-group XObject.
CA	number	*(Optional)* Similar to **ca**, but applies only when stroking paths and glyph outlines.

SMask	dictionary or name	*(Optional)* Soft mask, used to provide the mask shape or opacity value at each point when computing the source shape or opacity for an object being painted. This is described by a *soft-mask dictionary*; see "Soft-Mask Dictionaries" on page 211. If this parameter is the name **None**, the mask value is implicitly 1 everywhere. Default value: **None**. This parameter is implicitly reset to its default value at the beginning of execution of a transparency-group XObject. When painting an image XObject, the soft mask parameter in the graphics state is ignored (treated the same as **None**) if the image XObject contains a **Mask** or **SMask** entry.
AIS	boolean	*(Optional)* Alpha is shape, indicating whether the sources of alpha are to be treated as shape (*true*) or opacity (*false*). This determines the interpretation of the constant alpha (**ca** and **CA**) and soft mask (**SMask**) parameters of the graphics state, as well as a soft-mask image (**SMask** entry) of an image XObject. Default value: *false*.
BM	name or array	*(Optional)* The blend mode for color compositing for each object painted. It must be the name of a blend mode or an array of alternate blend modes. See "Blend Modes" on page 193. Default value: **Normal**. This parameter is implicitly reset to its default value at the beginning of execution of a transparency-group XObject.
TK	boolean	*(Optional)* Text knockout parameter, which determines the behavior of overlapping glyphs within a text object. If the value of **TK** is *false*, each glyph in a text object is treated as a separate elementary object; when glyphs overlap, they will composite with one another. If the value of **TK** is *true*, all the glyphs in a text object are treated together as a single elementary object; when glyphs overlap, later glyphs will knock out earlier ones in the area of overlap. See Section 9.18. Default value: *true*.

Note: *When the* **gs** *operator alters any of the above parameters, the new values completely replace the old ones. In particular, the soft mask is not intersected with its former value, as might be inferred from its role as a "soft clip" that is a generalization of the clipping path.*

Note: *Although the soft mask is defined as a parameter in the graphics state, its intended use is to clip only a single object at a time (either an elementary object or a group). If a mask is applied when painting two or more objects that overlap, the effect of the mask will multiply with itself in the area of overlap (except in a knockout group), producing a result shape or opacity that is probably not what is intended. To apply a soft mask to multiple objects, it is usually best to treat those* *objects as a group and apply the mask when painting the group. The foregoing considerations also apply to the constant alpha parameter in the graphics state.*

9.6 Path-Painting Operators—4.4.2 [8.6.2]

[Change the following entries in Table 4.10:]

Table 4.10 Path-painting operators

OPERANDS	OPERATOR	DESCRIPTION
—	**B**	Fill and then stroke the path, using the nonzero winding number rule to determine the region to fill. This produces the same result as specifying two path objects with the same path, painting the first with **f** and the second with **S**. Note that the filling and stroking portions of the operation consult different values of several graphics state parameters, such as color. For transparency compositing purposes, the combined fill and stroke are treated as a single graphics object, as if they were enclosed in a transparency group. This implicit group is established and used as follows: • If overprinting of the stroke is enabled (**OP** parameter is *true*) and the values of the fill and stroke alphas (**ca** and **CA**) are equal, then a non-isolated non-knockout group is established. Within the group, the fill and stroke are performed with alpha value 1 but with CompatibleOverprint blend mode. Then the group results are composited with the backdrop using the originally-specified alpha and blend mode. • Otherwise, a non-isolated knockout group is established. Within the group, the fill and stroke are performed with their prevailing alpha values (**ca** and **CA**) and blend mode. Then the group results are composited with the backdrop using alpha value 1 and **Normal** blend mode. ***Note:*** *Overprinting of the stroke over the fill does not work in the second case (although either the fill or the stroke can still overprint the backdrop). Furthermore, if* ***OP*** *is true, the results produced by the* ***B*** *operator are discontinuous at the transition between equal and unequal values of the* ***ca*** *and* ***CA*** *parameters. For this reason, it is best not to use overprinting with the* ***B*** *operator if* ***CA*** *and* ***ca*** *are being varied independently.* The purpose of the above rules is to avoid having a non-opaque stroke composite with the result of the fill in the region of overlap, which would produce a “double border” effect that is usually undesirable. The special case that applies when **OP** is *true* is for upward compatibility with the overprinting behavior of PDF 1.3 and earlier. If a desired effect cannot be achieved

with the **B** operator, it can be achieved by specifying the fill and stroke with separate path objects and an explicit transparency group.

Note: *These transparency semantics also apply to the* ***B****, ***b***, *and* ***b**** *operators, as well as to glyphs painted with text rendering mode 2 or 6. However, this does not change their descriptions, which are based on the behavior of the* ***B*** *operator.*

9.7 Clipping Path Operators—4.4.3 [8.3.1.1]

[Append the following to the first paragraph:]

In the context of the transparency model, the clipping path constrains an object's shape. The effective shape is the intersection of the object's intrinsic shape and the current clipping path: the source shape value is 0 outside this intersection. In the same vein, the shape of a transparency group is defined as the union of the shapes of all constituent objects. This shape is likewise influenced by the clipping path in effect when each of those objects is painted; it is additionally constrained by the clipping path in effect at the time of the **Do** operator, when the group's results are painted onto the backdrop.

Note: *The clipping path and the soft mask are independent parameters of the graphics state. Even though the "soft" shape mask could be considered a generalization of the "hard" clip, attempting to unify these parameters would disrupt the existing PDF graphics model unacceptably.*

9.8 Device Color Spaces—4.5.3 [7.12]

[Append the following:]

Use of device color spaces is subject to special treatment within a transparency group whose group color space is CIE-based; see "Transparency-Group XObjects" on page 207. In particular, the device color space operators should be used only if the device color spaces have been remapped to CIE-based color space by means of the **DefaultGray**, **DefaultRGB**, and **DefaultCMYK** color space resources. Otherwise, the results will be implementation-dependent and unpredictable.

9.9 CIE-Based Color Spaces—4.5.4 [8.5.1.5]

[Append the following to the introductory text, immediately before the “CalGray Color Spaces” subsection:]

In some workflows, a PDF document is intended for reproduction on a specific output device, such as a printing press with particular inks and media. The source colors for some or all objects in the document are specified with CIE-based color spaces that match the calibration of the intended output device. The document is therefore device-independent and will produce reasonable results if retargeted to a different output device.

In this situation, the expectation is that if the document is printed on the intended output device, source colors that have been specified in a color space that matches the output device will pass through unchanged, without conversion to and from the intermediate CIE 1931 XYZ space. The particular case of interest is an ICCBased color space specifying a CMYK printing profile that matches the output device. Converting 4-component colors to 3-component colors and back to 4-component colors will lose information; this is best avoided if not necessary.

PDF does not prescribe the conditions under which this should be done, since there is nothing in PDF that describes the calibration of the output device. However, PDF viewer applications may provide the ability to specify a particular calibration to use for printing, proofing, or previewing. This is considered to be the calibration of the native color space of the intended output device, which is usually **DeviceCMYK**. (In this situation, conversion to the actual calibration of the display or proofing device is completely hidden by the viewing application and plays no part in the interpretation of PDF color spaces.)

When **DeviceCMYK** is calibrated in this way, any CIE-based source color space matching this calibration can be treated as if it were specified as **DeviceCMYK**. When this is done, all of the semantics of **DeviceCMYK** spaces should also apply, even though they do not apply to CIE-based spaces in general. In particular:

- The nonzero overprint mode (set by **OPM**) determines the interpretation of color component values in this space.
- If the space is used as the blending color space for a transparency group, components of the space, such as /Cyan, can be selected in a **Separation** or **DeviceN** color space used within the group.

- Likewise, any uses of device color spaces for objects within the group have well-defined conversions to the group color space.

Note: *A source color space can be specified directly—say, with an* ***ICCBased*** *color space—or indirectly using the default color space feature—say,* ***DefaultCMYK****. The treatment of a CIE-based color space as if it were a device color space should not depend on whether the CIE-based color space is specified directly or indirectly.*

9.10 CIE-Based Color Spaces (ICCBased Color Spaces)—4.5.4 [8.5.1.5]

[After the paragraph following Table 4.18, insert the following:]

The above requirements apply to an **ICCBased** color space that is used to specify the source colors of graphics objects. An **ICCBased** color space that is used as the blending color space for a transparency group must have both “to CIE” (*AToB*) and “from CIE” (*BToA*) information. This is because the group color space is used both as the destination for objects being painted within the group and as the source for the group’s results.

9.11 CIE-Based Color Spaces (Rendering intents)—4.5.4 [8.5.1.5]

[Append the following:]

For an introduction to the problem of determining rendering parameters in the face of transparency, see Section 7.4, “Rendering Parameters.” The rendering intent parameter is needed whenever a color must be converted from a CIE-based color space to a different (CIE-based or device) color space. This can occur:

- when painting an elementary object with a CIE-based color into a transparency group having a different color space. In this case, the rendering intent used is the current rendering intent in the graphics state at the time of the painting operation.
- when painting a group whose color space is CIE-based into a parent group having a different color space. In this case, the rendering intent used is the current rendering intent in effect at the time of the **Do** operator for the group.

- when the page group has a CIE-based color space. The rendering intent used to convert colors to the native color space of the output device is the default rendering intent for the page.

9.12 Special Color Spaces (Separation Color Spaces)—4.5.5 [7.12.8]

[Append the following:]

A **Separation** color space is ordinarily used to produce a spot color—an additional color component, independent of the ones that are used to produce process colors. When an object is painted transparently with a spot color, that color is composited with the corresponding spot color component of the backdrop, independent of the compositing that is performed for process colors. A spot color retains its own identity; it is not subject to conversion to or from the color space of the enclosing transparency group or page.

Any object, including a transparency group, is considered to have been painted with all color components that exist, both process and spot. Components that haven't been explicitly painted have an additive color value of 1 (that is, a subtractive tint value of 0). This has consequences that are discussed in Section 7.3, "Overprinting and Erasing."

Instead of producing a spot color, a **Separation** color space can be used to paint a single process colorant, such as the Cyan component in a device whose native color space is **DeviceCMYK**. However, this is permitted only in a group that inherits the native color space of the output device. If such a **Separation** color space is used in a transparency group that specifies its own color space, the alternate color space will be substituted.

Spot colors are never available in a transparency-group XObject that is used to define a soft mask. The alternate color space will always be substituted in that situation.

(DeviceN Color Spaces)

[Append the following:]

The transparency considerations described above for **Separation** color spaces also apply to **DeviceN** color spaces.

9.13 Overprint Control—4.5.6 [8.4.9]

[Append the following:]

The semantics of the overprint control parameters (set by the **op**, **OP**, and **OPM** entries in a graphics state parameter dictionary) are respecified in terms of transparency, using a special blend mode named CompatibleOverprint. See Section 7.3, “Overprinting and Erasing” and “CompatibleOverprint Blend Mode” on page 194. Note that overprinting behavior applies only to elementary graphics objects, not to the results of transparency groups.

9.14 Tiling Patterns—4.6.2 [7.17.2]

[Include the following subsection somewhere:]

Tiling Patterns and Transparency

A tiling pattern is defined by an arbitrary sequence of graphics objects that are used to paint the pattern cell, which is then replicated to tile the region being painted. Those objects can include transparent objects and transparency groups. Transparent compositing can occur both within the pattern cell and between it and the backdrop wherever the pattern is painted. This section specifies the semantics of patterns in the presence of transparency.

A pattern definition is treated as if it were enclosed in a non-isolated, non-knockout transparency group. This has the following consequences:

- As is always the case for transparency groups, the transparency parameters in the graphics state (blend mode, alpha, and soft mask) are initialized to default values prior to evaluating the pattern definition.

  Note: *Except for the transparency parameters, all graphics state parameters are set to their state at the beginning of the content stream in which the pattern is defined as a resource. This is the normal behavior of patterns, independent of their interaction with transparency.*

- The result of evaluating the pattern is to produce a color, shape, and opacity at each point. This result is then applied to the interior of the graphics object being painted with the pattern. Effectively, the pattern defines the object’s source color (*Cs*), shape (*fj*), and opacity (*qj*), as used in the transparency compositing formulas.

- Painting the object with the pattern is subject to the transparency parameters in effect at the time the object is specified, just the same as painting an object with a constant color.

In other words, the current transparency parameters are not inherited by the pattern definition, but take effect only when the result of evaluating the pattern is used to paint an object using the pattern.

In the general case, the effect of tiling with a pattern must be as if the definition of the pattern were re-executed for each tile, taking into account the color of the backdrop at each point. This is unlike painting with a completely opaque pattern, where the pattern cell can be evaluated once and then replicated.

Note: *This can be significantly optimized in the common case in which the pattern consists entirely of objects painted with* **Normal** *blend mode. In that case, the same effect can be produced by treating the pattern cell as if it were an isolated group. The pattern cell can have color, shape, and opacity at each point, but those results do not depend on the backdrop. This means that the pattern cell can be evaluated once and then replicated, just as it can for a pattern defined with opaque painting.*

Note: *In a raster-based implementation of tiling, it is important that all tiles together be treated as a single transparency group. This avoids artifacts due to multiple marking of pixels along the boundaries between adjacent tiles.*

The above discussion applies to both colored (**PaintType** 1) and uncolored (**PaintType** 2) tiling patterns. These two types of patterns differ in how colors are specified. A colored tiling pattern specifies colors as part of the definition of the pattern cell; an uncolored tiling pattern uses a single color that is specified separately when the pattern is used. An uncolored pattern's definition may not specify colors; this restriction extends to any transparency group that the definition includes. However, there are no corresponding restrictions on specifying transparency parameters in the graphics state.

9.15 Shading Patterns—4.6.3 [7.17.3]

[Include the following subsection somewhere:]

Shading Patterns and Transparency

For transparency compositing purposes, the effect of a shading pattern is as if the shading dictionary were applied with the **sh** operator in a non-isolated, non-knockout transparency group. If the shading dictionary has a **Background** entry, the group is first filled with the background color before **sh** is invoked.

The transparency group's resulting color, shape, and opacity are then applied to the interior of any graphics object being painted with the pattern. The consequences of this are discussed in "Tiling Patterns and Transparency" on page 202; in particular, graphics state parameters are handled as described there.

Note: *The initial graphics state parameters obtained from the parent content stream, possibly augmented by the* ***ExtGState*** *entry in the pattern dictionary, apply only to the shading itself, not to the object being painted with the shading. Only the parameters that affect the* ***sh*** *operator, such as the CTM and rendering intent, are used. Parameters that affect path-painting operators are not used, since the execution of* ***sh*** *does not entail painting a path.*

[Change Table 4.25 as follows:]

Table 4.25 [7.58] Entries common to all shading dictionaries

KEY	TYPE	VALUE
Background	array	*(Optional)* An array of color components appropriate to the color space, specifying a single background color value. If present, this value specifies the color to use at all points lying outside the bounds of the shading object itself. The background is applied only in a shading pattern, not in a shading that is painted directly with the **sh** operator..

9.16 Image Dictionaries—4.8.4 [7.13.1]

[Add the following entry to Table 4.35:]

Table 4.35 [7.33] Entries in an image dictionary

KEY	TYPE	VALUE
SMask	stream	*(Optional)* A subsidiary image XObject defining a *soft-mask image* to be used as a source of shape or opacity values for transparency purposes. Unlike

a general soft mask that can be derived from arbitrary objects (see Section 6, “Soft Masks”), this mask is defined solely by an image, whose samples are interpreted as mask values directly. The contents of the soft-mask image are described in Table 10. If this entry is absent, no soft mask is associated with the image (though the soft mask in the graphics state may still apply).

If **SMask** is present, the **Mask** entry, if present, is disregarded. Furthermore, if either **SMask** or **Mask** is present, it overrides the soft mask parameter in the graphics state. (However, the other transparency-related graphics state parameters—constant alpha and blend mode—remain in effect.) The alpha is shape (**AIS**) parameter in the graphics state determines whether this soft-mask image is to be treated as shape or opacity.

Soft-Mask Images

Table 10 documents the entries in a soft-mask image dictionary. Except as noted, the syntax and semantics of the entries are the same as in a regular image XObject, documented in Table 4.35. The image XObject entries that are not mentioned here are not relevant to this type of image and are ignored.

TABLE 10 Entries in a soft-mask image dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* If present, must be **XObject**.
Subtype	name	*(Required)* Must be **Image**.
Width	integer	*(Required)* If a **Matte** entry is specified, the value of **Width** must be the same as the **Width** of the parent image. Otherwise, the **Width** of the mask image is independent of the **Width** of the parent image. Both images are mapped to the unit square in user space (as all images are), whether or not the samples coincide individually.
Height	integer	*(Required)* Same considerations as for **Width**.
BitsPerComponent	integer	*(Required)*
ColorSpace	name	*(Required)* Must be **DeviceGray**.
Decode	array	*(Optional)* Default value: [0 1].
Interpolate	boolean	*(Optional)*
ImageMask	boolean	*(Optional)* Must be *false* or absent.
Mask	stream	*(Optional)* Must be absent.

SMask	stream	*(Optional)* Must be absent.
Matte	array	*(Optional)* Specifies a color, the *matte color*, with which the image data in the parent image has been pre-blended (see below). The array must contain *n* numbers, where *n* is the number of components in the **ColorSpace** entry for the parent image; the numbers must be valid color components in that color space. If the **Matte** entry is absent, the image data is not pre-blended.

Pre-Blended Image Data

When an image is accompanied by a soft-mask image, it is sometimes advantageous for the image data to be pre-blended with some background color, called the *matte color*. Each image sample represents a weighted average of the original source color and the matte color, using the corresponding mask sample as the weighting factor. (This is a generalization of a technique that is commonly called "pre-multiplied alpha.")

If the image data is pre-blended, the matte color must be specified by a **Matte** entry in the soft-mask image XObject. The pre-blending computation, performed componentwise, is as follows:

$$c' = m + \alpha \cdot (c - m)$$

where c is the original image component value, m is the matte color component value, and α is the corresponding mask sample. The result, c', is the value that should be provided in the image source data.

Note: *This computation uses actual color component values, with the effects of the* **Filter** *and* **Decode** *transformations already performed. The computation is the same whether the color space is additive or subtractive.*

When pre-blended image data is used in transparency blending and compositing computations, the results are the same as if the original, unblended image data was used and no matte color was specified. In particular, the inputs to the blend mode function are the original color values. This may sometimes require the implementation to invert the above formula to derive c from c'. If this produces a c value that lies outside the bounds of color component values for the image color space, the results are unpredictable.

The pre-blending computation is done in the color space specified by the parent image's **ColorSpace** entry. This is independent of the group color space into which the image may be painted; if a color conversion is required, inversion of the pre-blending must precede the color conversion. If the image color space is an **Indexed** color space, the color values in the color table (not the index values themselves) must be pre-blended.

9.17 Form XObjects—4.9 [7.13.7]

[Add the following entry to Table 4.41:]

Table 4.41 [7.37] Entries in a type 1 form dictionary

KEY	TYPE	VALUE
Group	dictionary	*(Optional)* A *group attributes dictionary*, which indicates that the entire contents of the form XObject are to be treated as a group and specifies the attributes of that group. See Table 11 for the contents of this dictionary. ***Note:*** *The following paragraph refers to a PDF 1.4 feature, referenced PDF, whose specification has not yet been published.* If the form XObject contains a **Ref** entry specifying a referenced PDF file, these group attributes also apply to the referenced PDF page when it replaces the proxy. This allows an arbitrary external PDF page to be treated as a transparency group, without requiring that the referenced PDF file be modified.

Transparency-Group XObjects

A form XObject containing a **Group** entry is to be treated as a group. The value of this entry is a *group attributes dictionary* containing the entries described in Table 11. A *transparency-group XObject* is a form XObject whose group attributes dictionary has subtype **Transparency**, which is the only group subtype defined at present.

A page object may also have a **Group** entry, which specifies the group attributes dictionary for the page as a whole. Some entries in the dictionary are interpreted slightly differently for a page group than for a transparency-group XObject, as indicated in the descriptions of those entries below. See also Section 5.7, "Page Group."

TABLE 11 Entries in a group attributes dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* If present, must be **Group**.
S	name	*(Required)* Group subtype. At present, the only defined value is **Transparency**; the remaining entries described below apply to this subtype. Future extensions may introduce other group subtypes, which will most likely have a different set of additional entries.
CS	name or array	*(Sometimes required, as discussed below)* Group color space. This specifies the color space into which colors are converted when painted into this group, and it defines the blending color space; see Section 3.1, “Blending Color Space.” It also defines the color space of the group as a whole when it in turn is painted as an object onto the group’s backdrop. **CS** may be any device or CIE-based color space that treats its components as independent additive or subtractive values in the range 0 to 1, subject to the restrictions described in Section 3.1, “Blending Color Space.” This excludes **Lab** and lightness-chromaticity **ICCBased** color spaces, as well as the special color spaces **Indexed**, **Separation**, **DeviceN**, and **Pattern**. Device color spaces are subject to remapping according to the **DefaultGray**, **DefaultRGB**, and **DefaultCMYK** entries in the **ColorSpace** dictionary in the current **Resources** dictionary; see *PDF Reference*, Section 4.5.4 [7.12.12]. Ordinarily, the **CS** entry is allowed only if **I** (Isolated) is *true*, and even then it is optional. However, **CS** is required in the group attributes dictionary for a transparency-group XObject that has no parent group or page from which to inherit. This situation arises for a transparency-group XObject that is the value of the **G** entry in a soft-mask dictionary of subtype **Luminosity**; see “Soft-Mask Dictionaries” on page 211. Additionally, it is always permissible to specify **CS** in the group attributes dictionary associated with a page object, even if **I** is false or absent. In the normal case in which the page is imposed directly on the output media, the page group is effectively isolated, regardless of the **I** value; thus, the **CS** value can take effect. But if the page is in turn used as an element of some other page and if the group is non-isolated, **CS** is ignored; the color space is inherited from the actual backdrop with which the page is composited. See Section 5.7, “Page Group.” Default value: color space of the parent group or page into which this transparency-group XObject is painted by **Do**. (The parent’s color space in turn can be either explicitly specified or inherited.) If a transparency-group XOb-

		ject is used as an annotation appearance, its default **CS** value inherits from the page.
I	boolean	*(Optional)* Specifies the isolated attribute of the group. This determines whether the initial backdrop for compositing the objects within the group is the group's backdrop (*false*) or a transparent backdrop (*true*). See Section 5.4, "Isolated Groups." Default value: *false*. In the group attributes dictionary for a page, the interpretation of the **I** value is slightly altered. In the normal case in which the page is imposed directly on the output media, the page group is effectively isolated, regardless of the **I** value. But if the page is in turn used as an element of some other page, the page is treated as if it were a transparency-group XObject; the **I** value is interpreted in the normal way to determine that group's isolated attribute.
K	boolean	*(Optional)* Specifies the knockout attribute of the group. This determines whether each object is composited with earlier objects in the group (*false*) or with the group's initial backdrop (*true*). See Section 5.5, "Knockout Groups." Default value: *false*.

When a transparency-group XObject is invoked by **Do**, the following actions occur in addition to the normal actions for invoking a form XObject:

1. If the value of **I** (Isolated) is *false*, the initial backdrop (within the bounding box specified by the XObject's **BBox** entry) is defined to be the accumulated color and alpha of the parent group or page—that is, the result of everything that has been painted in the parent up to that point. (However, if the parent is a knockout group, the initial backdrop is the same as the parent's initial backdrop.) If the value of **I** is *true*, the initial backdrop is defined to be transparent.
2. In the graphics state, the current nonstroking and stroking alpha are initialized to 1, the current soft mask is initialized to **None**, and the blend mode is initialized to **Normal**.

   Note: *The purpose of initializing the soft mask, alpha, and blend mode parameters in the graphics state at the beginning of execution is to ensure that these parameters aren't applied twice: once when objects are painted into the group and again when the group is painted into the parent group or page.*
3. Objects painted by operators in the transparency-group XObject's content stream are composited into the group according to the rules described in Section 3, "Color Compositing Computations," and Section 4, "Shape and Opacity Computations." The **K** (Knockout) entry in the group attributes dictionary

and the transparency parameters of the graphics state contribute to this computation.

4. If **CS** (Color Space) is specified in the group attributes dictionary, all painting operators convert source colors to the specified color space prior to compositing objects into the group. The resulting color at each point in the group is interpreted in that color space. If **CS** is not specified, the prevailing color space is dynamically inherited from the parent group or page. (If not otherwise specified, the page group's color space is the device color space of the output device.)

5. After execution of the transparency-group XObject's content stream, the graphics state reverts to its former state prior to the execution of **Do** (as occurs for any form XObject, not just a transparency-group XObject). The group's shape—the union of all objects painted into the group, clipped by the XObject's bounding box—is then painted into the parent group or page, using the group's accumulated color and opacity at each point within the shape.

Note: *If a given transparency-group XObject is invoked by* **Do** *more than once, each invocation is treated as a separate transparency group. That is, the result must be as if the group were independently composited with the backdrop on each invocation. If the implementation performs any caching of rendered form XObjects, it must take this requirement into account.*

The above actions occur only for a transparency-group XObject—one having a group attributes subdictionary whose **S** (Subtype) is **Transparency**. A regular form XObject—one having no **Group** entry—is not subject to any grouping behavior for transparency purposes. That is, the graphics objects that it contains are composited individually, just as if they were painted directly into the parent group or page.

If the group's color space (either specified by **CS** or inherited) is a CIE-based color space (**CalGray**, **CalRGB**, or **ICCBased**), any use of device color spaces for painting objects is subject to special treatment. Device colors cannot be painted directly into such a group, since there is no generally-defined method for converting them to the CIE-based color space. This problem arises in the following cases:

- **DeviceGray**, **DeviceRGB**, and **DeviceCMYK** color spaces, unless remapped to CIE-based color spaces by means of the corresponding **DefaultGray**, **DefaultRGB**, or **DefaultCMYK** color space resources.
- Operators that specify a device color space implicitly, such as **rg**, unless that space is remapped.
- Special color spaces whose underlying space is a device color space, unless that space is remapped.

We recommend that the color space remapping mechanism always be employed when defining a transparency group whose color space is CIE-based. However, if a device color is specified and is not remapped, it will be converted to the CIE-based color space in an implementation-defined fashion, producing unpredictable results.

Soft-Mask Dictionaries

Instead of being painted on the current page, a transparency-group XObject can be used as a soft mask—a source of position-dependent shape or opacity values to apply as a mask when painting other objects; see Section 6, “Soft Masks.” The soft mask is a parameter of the graphics state; see Section 9.5. Its value is a *soft-mask dictionary*, one of whose entries is the transparency-group XObject that is to be treated as the source of the mask values. The soft-mask dictionary describes how the mask is to be derived from the results of compositing the transparency group; see Table 12.

TABLE 12 Entries in a soft-mask dictionary

KEY	TYPE	VALUE
Type	name	*(Optional)* If present, must be **Mask**.
S	name	*(Required)* Subtype that specifies how the mask is to be computed. **Alpha** means to just use the group’s computed alpha (product of shape and opacity), disregarding its color. **Luminosity** means to convert the group’s computed color to a single-component luminosity value. See Section 6, “Soft Masks.”
G	stream	*(Required)* Transparency-group XObject that is to be used as the source of alpha or color values for generating the mask. If the value of **S** (Subtype) is

		Luminosity, the group attributes dictionary (value of **Group** in the transparency-group XObject) must contain a **CS** entry that defines the color space in which the compositing computation is to be done.
BC	array	*(Optional; consulted only if* **S** *(Subtype) is Luminosity)* The color that is to be used as the backdrop with which the transparency-group XObject **G** is to be composited. The value of **BC** is an array of *n* numbers, where *n* is the number of components in the color space specified by the **CS** entry in the group attributes dictionary. Default value: the color space's initial value, which represents black.
TR	function or name	*(Optional)* Transfer function, used during the conversion to mask values. This is either a function object (dictionary or stream) or the name **Identity**. The input to this function is the computed alpha or luminosity (depending on the value of the **S** (Subtype) entry), in the range 0 to 1. The output is the mask value in the range 0 to 1; it is clipped to that range if necessary. Default value: **Identity**.

If **S** (Subtype) is **Alpha**, the transparency-group XObject **G** is evaluated to compute a group alpha only; the colors of the constituent objects are ignored and the color compositing computations are not performed. The computed group alpha is then passed through the **TR** function to produce the mask. Outside the bounding box of the transparency-group XObject, the mask value is the result of presenting 0 as input to the **TR** function.

If **S** is **Luminosity**, the transparency-group XObject **G** is composited with a fully-opaque backdrop whose color is **BC** everywhere. The computed result color is then converted to a single-component luminosity value, which is passed through the **TR** function to produce the mask. Outside the bounding box of the transparency-group XObject, the mask value is the **BC** color transformed to luminosity and passed through the **TR** function.

The coordinate system of the soft mask is defined as the **Matrix** of the transparency-group XObject concatenated with user space at the moment the soft-mask dictionary is established in the graphics state by the **gs** operator.

In a transparency-group XObject that defines a soft mask, spot color components are never available, regardless of whether they are available in the group or page on which the soft mask is used. If the content stream specifies a **Separation** or **DeviceN** color space that uses spot color components, the alternate color space will be substituted.

9.18 Text State Parameters and Operators—5.2 [8.7]

[Add the following somewhere:]

Text Knockout

The text knockout parameter determines what are considered to be the elementary objects for transparency compositing purposes. If text knockout is *false*, each glyph in a text object is treated as a separate elementary object; when glyphs overlap, they will composite with one another. If text knockout is *true*, all the glyphs in a text object are treated together as a single elementary object; when glyphs overlap, later glyphs will knock out earlier ones in the area of overlap. (The latter behavior is equivalent to treating the entire text object as if it were a knockout group.)

The text knockout parameter is specified as the value of the **TK** entry in a graphics state parameter dictionary. (Unlike other text state parameters, there is no special operator for specifying this parameter.) The text knockout parameter applies to entire text objects; it may not be set between the **BT** and **ET** operators that delimit a text object.

9.19 Text Rendering Mode—5.2.5 [8.7.1.7]

[Add the following somewhere:]

Rendering modes 2 and 6 specify both fill and stroke. If glyphs in the text object overlap, the result must be as if the fill and stroke are done on a glyph-by-glyph basis. That is, perform the fill and stroke for the first glyph, then the fill and stroke for the second glyph, and so on; not all of the fills followed by all of the strokes for multiple glyphs together. This produces the correct visual appearance of stacking of opaque glyphs.

For transparency compositing purposes, text rendering modes 2 and 6 have the same effect as applying the **B** operator to the glyph outline path. That is, the effect is equivalent to performing the fill and the stroke in a transparency group that is established according to the **B** operator description in Table 10 on page 205. Note that this behavior is independent of the value of the text knockout (**TK**) graphics state parameter.

9.20 Text-Showing Operators—5.3.2 [8.7.5]

[Add the following somewhere:]

Note: *Within a text object, there is no significance in the grouping of glyphs into strings presented to text-showing operators such as* **Tj**. *Showing multiple glyphs with one invocation of* **Tj** *produces the same results as showing them with a separate invocation for each glyph.*

9.21 Conversion from DeviceRGB to DeviceCMYK—6.2.3 [8.4.10]

[Append the following:]

For an introduction to the problem of determining rendering parameters in the face of transparency, see Section 7.4, "Rendering Parameters." The undercolor removal and black generation functions are needed whenever a **DeviceRGB** color must be converted to a **DeviceCMYK** color. This can occur:

- when painting an elementary object with a **DeviceRGB** color directly into a transparency group whose color space is **DeviceCMYK**. In this case, the functions used are the current undercolor removal and black generation parameters in the graphics state at the time of the painting operation.
- when painting a group whose color space is **DeviceRGB** into a parent group whose color space is **DeviceCMYK**. In this case, the functions used are the ones in effect at the time of the **Do** operator for the group.
- when the color space of the page group is **DeviceRGB** and the native color space of the output device is **DeviceCMYK**. The functions used are the default functions for the page.

9.22 Transfer Functions—6.3 [8.4.12]

[Append the following:]

For an introduction to the problem of determining rendering parameters in the face of transparency, see Section 7.4, "Rendering Parameters." The transfer function to be used at any given point on the page is the transfer function parameter that is in effect at the time of painting the last (topmost) graphics object enclosing that point, but only if that object is fully opaque.

An object is considered to be fully opaque if all of the following conditions are true at the time the object is painted:

- The constant alpha parameter (**ca** or **CA**, whichever applies to the painting operation) is 1.
- The soft mask parameter (**SMask**) in the graphics state is **None**. If the object is an image, there is no **SMask** entry in the image dictionary.
- The blend mode parameter (**BM**) is either **Normal** or **Compatible**.
- The foregoing three conditions were also true at the time the group containing the object (or any direct ancestor group) was invoked by **Do**.
- If the current color is a tiling pattern, all of the objects comprising its definition also satisfy the above conditions.

The transfer function to use at a given point must be determined on a per-component basis if any graphics object is painted with overprinting enabled (the **op** or **OP** parameter is true). Overprinting implicitly invokes the CompatibleOverprint blend mode. An object is considered opaque for a given component only if CompatibleOverprint returns the source color (not the backdrop color) for that component. See “CompatibleOverprint Blend Mode” on page 194.

For portions of the page whose topmost object is not fully opaque or that are never painted at all, the default transfer function for the page is used.

9.23 Halftones—6.4 [8.4.13]

[Append the following:]

For an introduction to the problem of determining rendering parameters in the face of transparency, see Section 7.4, “Rendering Parameters.” The halftone to be used at any given point on the page is the halftone parameter that is in effect at the time of painting the last (topmost) graphics object enclosing that point, but only if that object is fully opaque. The conditions for applying it are the same as for the transfer function parameter, described in Section 9.22 above.

9.24 Annotation Appearances—7.4.4 [6.6.3]

[Add the following somewhere:]

An annotation appearance can include transparency if the form XObject that defines the appearance has a **Group** entry whose value is a group attributes dictionary of subtype **Transparency**, indicating that the XObject is a transparency-group XObject. This group is composited with a backdrop consisting of the page content and any other annotation appearances that are below this one. The final compositing operation is performed using a constant alpha of 1, **None** soft mask, and **Normal** blend mode.

If a transparent annotation appearance is placed on top of an annotation that is visible but is drawn without using an annotation appearance dictionary, the effect is implementation-dependent. This is because such annotations are sometimes drawn by means that do not conform to the Adobe imaging model. Also, the effect of highlighting a transparent annotation appearance is implementation-dependent.

10 Terminology Summary

Table 13 lists terms used in Section 2 through Section 7 to explain the transparency model. Where a term refers to some variable, the table gives the variable name that is conventionally used. This table does not include specific terms for PDF objects used to represent the transparency model.

TABLE 13 Terminology summary

TERM	VARIABLE(S)	MEANING
Alpha	α	The product of shape and opacity.
Backdrop		The stack of preceding objects onto which a new object is being painted.
Backdrop alpha	αb, α_{i-1}	The computed alpha of the backdrop at a given point.
Backdrop color	Cb, C_{i-1}	The computed color of the backdrop at a given point.
Backdrop fraction	bf	The fraction of the group's accumulated color that is attributable to the initial backdrop color.
Backdrop opacity	qb, q_{i-1}	The computed opacity of the backdrop at a given point.
Backdrop shape	fb, f_{i-1}	The computed shape of the backdrop at a given point.
Blend mode	$B(Cb, Cs)$	A function in the color compositing computation that customizes how source and backdrop colors combine.

Blending color space		The color space in which all colors in the compositing function are represented. Input colors are converted to this color space when necessary. Same as group color space.
Color compositing function		The function that computes the color result of painting an object over a backdrop.
Constant opacity	qk, qk_i	A simple scalar contribution to the source opacity.
Constant shape	fk, fk_i	A simple scalar contribution to the source shape.
Current page		The result produced by compositing the entire stack of objects onto a backdrop that is initially white and fully opaque.
Element	E_i	A compound variable representing all the parameters of an object being treated as an element of a group. The parameters include color, shape, and opacity (either intrinsic or computed), as well as the separately-specified shape, opacity, and blend mode parameters that are to be used when compositing the object. The element can correspond to either an elementary object or a group.
Elementary object		A unitary object in the imaging model, such as a fill, stroke, glyph, or sampled image. (Contrast with group.)
Group		A sequence of consecutive objects in a stack that are composited together and then treated as a single object to be composited with the group's backdrop.
Group alpha	αg_i	Accumulated alpha of objects within a group, excluding the group's backdrop.
Group backdrop		Ordinarily, the result of compositing all elements up to but not including the first element of the current group. However, if the parent of the current group is a knockout group, then the group backdrop is the same as the parent's group backdrop.
Group color space		A color space associated with the group as a whole. Source colors are converted into this color space if necessary. All blending and compositing computations among objects in the group occur using colors in this color space. The group's result color is interpreted as being in this color space.
Group opacity	qg_i	Accumulated opacity of objects within a group, excluding the group's backdrop. (This is rarely represented alone; it is usually combined with shape in the corresponding alpha value.)
Group shape	fg_i	Accumulated shape of objects within a group, excluding the group's backdrop.
Immediate backdrop		The result of compositing all elements of a group up to but not including the current element of interest.

Initial backdrop		A backdrop that is selected for compositing the current group’s first element. This is either the same as the group backdrop (non-isolated group) or a fully transparent backdrop (isolated group).
Isolated group		A group whose elements are composited with a transparent initial backdrop, rather than with the group’s backdrop.
Knockout group		A group whose elements are composited with the group’s initial backdrop, rather than with the results of compositing elements lower on the group’s stack.
Mask		A source of position-dependent shape or opacity values that is independent of the objects being composited.
Mask opacity	qm, qm_i	The value of an opacity mask at a given point.
Mask shape	fm, fm_i	The value of a shape mask at a given point.
Non-separable blend mode		A blend mode in which a component of the result color is a function of components other than the corresponding component of the backdrop and source colors.
Object		Either an elementary object or a group.
Object color		The intrinsic color of the source object at a given point, in its original color space.
Object opacity	qj, qj_i	The intrinsic opacity of the source object at a given point.
Object shape	qj, qj_i	The intrinsic shape of the source object at a given point.
Opacity	q	A weighting factor that determines the relative contribution of the associated color to the overall result of a color compositing computation. Its value is in the range 0 (fully transparent) to 1 (fully opaque).
Opacity compositing function		The function that computes the opacity result of painting an object over a backdrop.
Opacity mask		See mask.
Page group		A group consisting of all the objects (both elementary objects and groups) that are painted directly onto the current page.
Process color		Any color that is produced by combinations of the device’s process colorants. (Contrast with spot color.)
Result alpha	αr, α_i	The result of the alpha compositing function at a given point, which becomes the backdrop alpha for the next object painted.
Result color	Cr, C_i	The result of the color compositing function at a given point, which becomes the backdrop color for the next object painted.

Result opacity	qr, q_i	The result of the opacity compositing function at a given point, which becomes the backdrop opacity for the next object painted. (This is rarely represented alone; it is usually combined with shape in the corresponding alpha value.)
Result shape	fr, f_i	The result of the shape compositing function at a given point, which becomes the backdrop shape for the next object painted.
Separable blend mode		A blend mode in which each component of the result color is a function of the corresponding component of the backdrop and source colors.
Shape	f	A weighting factor that determines the degree to which the results of a compositing operation (color and opacity) replace the backdrop. The extreme values 0 and 1 represent the familiar notions of “outside” and “inside” a hard-edge shape.
Shape compositing function		The function that computes the shape result of painting an object over a backdrop.
Shape mask		See mask.
Soft clip		See shape mask.
Soft mask		Same as mask; the “soft” adjective emphasizes that this is a continuous-tone value, not just 0 or 1.
Source alpha	αs, αs_i	The product of source shape and source opacity.
Source color	Cs, Cs_i	The intrinsic color of the source object at a given point, converted to the blending color space if necessary.
Source opacity	qs, qs_i	The product of all input opacity values: object opacity, constant opacity, and mask opacity.
Source shape	fs, fs_i	The product of all input shape values: object opacity, constant opacity, and mask opacity.
Spot color		A color that is produced by application of a separate colorant, such as special ink, that is independent of the normal process colorants.
Stack		An ordered sequence of objects painted onto the page or collected into a group.
Transparency group		See group.
Union	Union(b, s)	The function used to composite shape and opacity values. It is a generalization of the conventional concept of “union” for solid shapes.
White-preserving blend mode		A separable function $B(cb, cs)$ having the property that $B(1, 1) = 1$ for all color components (expressed in additive form).

11 Compatibility

11.1 Backward Compatibility

A PDF 1.3 or earlier viewer will ignore all transparency-related parameters, such as alpha, soft mask, and blend mode. All graphics objects, including ones defined in transparency-group XObjects, will be painted opaquely.

11.2 Forward Compatibility

The PDF 1.3 overprint control parameters (set by the **op**, **OP**, and **OPM** entries in a graphics state parameter dictionary) are respecified to work as special transparency blend modes. Their behavior should be compatible when only opaque painting operations are performed. When they are combined with other transparency operations, their behavior, though well-defined, is not a logical extension of the selective colorant marking semantics of PDF 1.3.

A process colorant (e.g., Cyan in a **DeviceCMYK** device) may not be treated as a spot color (via a **Separation** or **DeviceN** color space) within a transparency group whose color space is not inherited from the device. The alternate color space will be used instead. This isn't really a compatibility issue, since PDF 1.3 doesn't have transparency groups to begin with. The concern is about legacy artwork that might be imported and then have transparency applied to it globally.

If a transparency group's color space is CIE-based, any objects that are painted using device color spaces will be converted into the group's color space by implementation-defined means, possibly producing unexpected results. This problem can be circumvented by properly defining the **DefaultGray**, **DefaultRGB**, and **DefaultCMYK** color space resources. Once again, this is not really a compatibility issue, but it may arise when attempting to apply transparency to legacy artwork.

11.3 PostScript Printing

The PostScript language does not support the transparency model. Therefore, a PDF 1.4 viewer must have means to produce a completely opaque description of the appearance of a document that uses transparency. A similar operation can produce a PDF file that can be correctly viewed by a PDF 1.3 viewer.

This involves some combination of shape decomposition and pre-rendering to "flatten" the stack of transparent objects on a page, perform all the transparency computations, and describe the final appearance using opaque objects only. This is an irreversible operation, since all of the information about how the transparency effects were produced has been lost.

Note: *Determining if a page contains transparency needing to be "flattened" can be accomplished by straightforward analysis of the page's resources; it isn't necessary to analyze the content stream.*

In order to perform the transparency computations properly, the PDF viewer needs to know the native color space of the output device. When the viewer controls the output device directly, this is no problem. However, when the viewer is generating PostScript output, it has no way to know the native color space of the PostScript output device. Making an incorrect assumption will ruin the calibration of any CIE-based colors appearing on the page. This problem can be addressed in a couple of ways:

- If the entire page consists of CIE-based colors, then flatten the colors to a single CIE-based color space, rather than to a device color space. The preferred color space for this purpose can easily be determined in the case that the page has a **Group** dictionary that specifies a CIE-based color space.
- Otherwise, flatten the colors to some assumed device color space with predetermined calibration. In the generated PostScript output, paint the flattened colors in a CIE-based color space having that calibration.

During conversion to PostScript, a decision must also be made about the set of available spot colors to assume. This is because the choice of spot colorant versus alternate color space affects the flattened results of the process colors. (This is unlike the situation with strictly opaque painting, where the decision can be deferred until the generated PostScript is executed.)

12 Overprinting, Erasing, and Transparency

This section provides details of the overprinting and erasing rules in PDF. The information here duplicates material that can be found elsewhere. However, the rules are complex and interact with transparency in non-obvious ways, so it is useful to have the information collected in one place and the results spelled out for every case. This material should be read in conjunction with Section 7.2, “Spot Colors,” and Section 7.3, “Overprinting and Erasing.”

Table 14 shows the existing rules as defined in the opaque imaging model of PDF 1.3 (which is an extension of the PostScript imaging model). Table 15 shows the equivalent rules, respecified as a special CompatibleOverprint blend mode in the transparency imaging model of PDF 1.4.

OP and **OPM** refer to the overprint flag and nonzero overprint mode control parameters in PDF. **OP** is equivalent to the **setoverprint** value in PostScript. There is no PostScript equivalent to **OPM**; it enables or disables the interpretation of **DeviceCMYK** color values to specify componentwise overprinting, as described in Section 7.3, “Overprinting and Erasing.”

TABLE 14 Overprinting and erasing rules in PDF 1.3

Source color space	Affected color component	Effect on that color component		
		OP false	OP true, OPM 0	OP true, OPM 1
DeviceCMYK & specified directly & not image	C, M, Y, or K	Paint source	Paint source	Paint source if source ≠ 0 Don’t paint if source = 0
	Process colorant ≠ CMYK	Paint source	Paint source	Paint source
	Spot colorant	Paint 0	Don’t paint	Don’t paint
Any process color space (including other cases of DeviceCMYK)	Process colorant	Paint source	Paint source	Paint source
	Spot colorant	Paint 0	Don’t paint	Don’t paint

TABLE 14 Overprinting and erasing rules in PDF 1.3

Source color space	Affected color component	Effect on that color component		
		OP false	OP true, OPM 0	OP true, OPM 1
Separation or DeviceN	Process colorant	Paint 0	Don’t paint	Don’t paint
	Spot colorant named in source space	Paint source	Paint source	Paint source
	Spot colorant not named in source space	Paint 0	Don’t paint	Don’t paint

TABLE 15 Overprinting and erasing recast as blend modes in PDF 1.4

Source color space	Affected color component of group color space	Value of blend mode CompatibleOverprint(*Cb*, *Cs*), expressed as tint		
		OP false	OP true, OPM 0	OP true, OPM 1
DeviceCMYK & specified directly & not image	C, M, Y, or K	*Cs*	*Cs*	*Cs* if *Cs* ≠ 0 *Cb* if *Cs* = 0
	Process color component ≠ CMYK	*Cs*	*Cs*	*Cs*
	Spot colorant	*Cs* (= 0)	*Cb*	*Cb*
Any process color space (including other cases of DeviceCMYK)	Process color component	*Cs*	*Cs*	*Cs*
	Spot colorant	*Cs* (= 0)	*Cb*	*Cb*

TABLE 15 Overprinting and erasing recast as blend modes in PDF 1.4

Source color space	Affected color component of group color space	Value of blend mode CompatibleOverprint(*Cb*, *Cs*), expressed as tint		
		OP false	OP true, OPM 0	OP true, OPM 1
Separation or DeviceN	Process color component	*Cs* (= 0)	*Cb*	*Cb*
	Spot colorant named in source space	*Cs*	*Cs*	*Cs*
	Spot colorant not named in source space	*Cs* (= 0)	*Cb*	*Cb*
Source is a group (not an elementary object)	All color components	*Cs*	*Cs*	*Cs*

In these tables, color component values are represented as subtractive tint values, because one ordinarily thinks of overprinting inks, not additive light values. The CompatibleOverprint blend mode is described as if it took subtractive arguments and returned subtractive results. However, in reality, the CompatibleOverprint blend mode (like all blend modes) treats color components as additive values; subtractive components must be complemented before and after application of the blend mode.

Please note an important difference between the two tables. In Table 14, the process color components being discussed are the actual device colorants—the color components of the native color space of the output device (**DeviceRGB**, **DeviceCMYK**, or **DeviceGray**). In Table 15, the process color components are those of the group's color space, which is not necessarily the same as that of the output device (and can even be something like **CalRGB** or **ICCBased**). For this reason, the process color components of the group color space cannot be treated as if they were spot colors (in a **Separation** or **DeviceN** color space); see Section 7.2, "Spot Colors."

This difference between PDF 1.3 and PDF 1.4 overprinting and erasing rules arises only within a transparency group (including the page group, if its color space is different from the native color space of the output device). Since PDF 1.3 doesn't have transparency groups to begin with, this difference is not considered to be an incompatibility. (There is no difference in the treatment of spot color components.)

Table 15 has one additional row at the bottom. It applies when painting an object that itself is a transparency group instead of an elementary object (fill, stroke, text, image). As explained in Section 7.2, "Spot Colors," a group is considered to paint every color component, both process and spot. Color components that were not explicitly painted by any object in the group have an additive color value of 1 (subtractive tint 0).

Since no information is retained about which components were actually painted within the group, compatible overprinting is not possible in this case. Thus, the CompatibleOverprint blend mode is never applied to a group. (Note that a transparency-aware application can choose a suitable blend mode, such as **Darken**, if it desires to produce an effect similar to overprinting of group results.)

13 Revision History

Note: *Change bars mark all changes since the initial release of May 30, 2000.*

November 30, 2000

Changed the semantics of the page group. The compositing formula for the page group is now applied only if the page is imposed on output media, not if it is placed on other content by an aggregating application. The formula itself is changed to isolate the page group from the backdrop. If a page object has a group-attributes dictionary, it specifies attributes of the page group, rather than being a shorthand for specifying a subsidiary transparency-group XObject. The interpretation of the transparency group attributes for a page group is slightly changed.

Clarified the kinds of **ICCBased** color spaces that are permitted as group color spaces.

January 23, 2001

Refined the treatment of the color rendering intent parameter in the presence of transparency groups. Rendering intent is now tied to objects (including group objects) instead of regions of the page. For consistency, the undercolor removal and black generation parameters work the same way.

Specified additional requirements on **ICCBased** color spaces that are permitted as group color spaces. In particular, they must be bidirectional.

Introduced a more complete specification of the semantics of tiling patterns in the presence of transparency. A pattern is treated as if it were defined within a transparency group, thereby defining the color and alpha at each point within an object being painted with the pattern.

Revised the treatment of overprinting as a blend mode. Formerly, overprinting behavior occurred only if the blend mode was specified as **Compatible**. Now, overprinting can be combined with any blend mode; a nested transparency group is implicitly created if necessary to achieve this. The semantics of the overprinting blend mode itself are unchanged. The default value of the blend mode parameter is now **Normal**, not **Compatible**.

Added guidelines on treating a CIE-based color space as if it were a device color space. (Although this doesn't have anything directly to do with transparency, it does have transparency-related ramifications, such as in the treatment of overprinting.)

February 5, 2001

Added a special case for the handling of the **B** (combined fill/stroke) operator if overprinting is enabled. This is for upward compatibility from the overprinting model of PDF 1.3 and earlier.

Extended the new specification of the semantics of tiling patterns to apply to shading patterns as well, particularly in how graphics state parameters are interpreted. Clarified the semantics of the **Background** entry in a shading dictionary.

Further clarified the semantics of the page group.

APPENDIX A

Adding Unicode Codepoints to Font Descriptors

This appendix discusses how Unicode characters that are not in a standard font should be represented in a Tagged PDF document. This is necessary to allow pass-through of characters such as *Figure Space*, *Em Quad*, *Tab*, *Non-Breaking Space*, etc., so that they will be preserved if the document is saved as, for example, HTML or RTF. This approach should be used when the goal is to avoid embedding a font or a font subset in the PDF.

A.1 Case 1: If the authoring tool is generating PDF directly, such as through the PDF library:

For Type 1 and similar fonts, use the differences table in a Font descriptor:

- Find an unused code-point
- Add a **ToUnicode** entry at that code-point for the desired Unicode
- Add a glyphname entry at that code-point which is the name of some real character in the font (often *space*, sometimes *minus*).
- Add width entry at that code-point which is the width of the real character (*not* the width of the unicode character), but remember the desired width of the new character somewhere else.
- When putting out the new code-point, explicitly kern (in the **TJ**) by the difference between the desired width and the width of the “real” substituted character. This will preserve the spacing both on the display and on printers.

For Type 0 fonts with **CMap**s:

- Make up a new **CMap** with all the existing characters and with the desired characters added at open code-points, and embed the new **CMap** in the PDF.

- Add width entry for the added characters which is the width of the real character (NOT the width of the unicode character) using the **W** or **DW** keys in the manner described in the "Character Widths in CIDFonts" section of the PDF Reference, but remember the desired width of the new character somewhere else.
- When putting out the new code-point, explicitly kern (in the **TJ**) by the difference between the desired width and the width of the "real" substituted character. This will preserve the spacing both on the display and on printers.

A.2 Case 2: If the authoring tool is generating PDF files through Distiller:

There is currently no way to build a **/Difference** entry via Distiller. Until this is fixed any application using the Distiller path should use the following work-around:

- Put out a single-character **TJ** string using the "real" font character and any necessary kerning.
- Make that single character a piece of Tagged text (Logical Structure Element type **Span**), and set the **ActualText** string in the **span** to the desired Unicode character.

The only other Distiller solution is to embed the actual font in the PDF file.