Copyright © 1991–1992 by Adobe Systems Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript, the PostScript logo, Adobe, and the Adobe logo are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions. AppleTalk is a registered trademark of Apple Computer, Inc. Other brand or product names are the trademarks or registered trademarks of their respective holders.

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

Supporting Data Compression in PostScript Level 2 and the filter Operator 5

1 Introduction 5

2 Applications of Filters 6
   Analysis of Compression Filters 6
   Analysis of Transmission Filters 7
   Compressing the PostScript Language Stream 8
   Compressing Images 10

3 Details of Using Filters 14
   Concatenating Filters 14

4 LZW Issues 16
   Optional LZW Parameters 16
   Licensing Issues 17

Appendix: Changes Since Earlier Versions 19

Index 21
1 Introduction

The concept of a filter is new to PostScript™ Level 2. A filter is a special kind of file that can be layered on top of another file to manipulate the data being read or written. Filters can either write data (encoding filters) or read data (decoding filters). Filters are generally used to transform the data in some way; either to make it smaller or to make it transmissible over a seven-bit communication line. See section 3.8.4, “Filters,” of the PostScript Language Reference Manual, Second Edition for a more thorough explanation of filters.

During printing, applications might use filters for several reasons:

- compressing images
- compressing an entire job stream
- making a file transmissible over a limited communication medium.

In the past, one of the largest bottlenecks in processing PostScript language files has been the speed of the transmission channels. This is particularly noticeable when dealing with objects such as images, which inherently contain a lot of data, but it is also noticeable when transmitting a text file over a slow serial line.

The standard encoding and decoding filters available in PostScript Level 2 are: LZW, Run Length, CCITT Fax (Group 3 1-D, 3 2-D, and 4), DCT (Discrete Cosine Transform, based on the proposed JPEG specification), ASCII base-85, ASCIIHex, NullEncode, and SubFileDecode filters. This paper will deal with all of the compression and transmission filters except the DCT encoding and decoding filters, which are covered in Technical Note #5116, “Supporting the DCT Filters in PostScript Level 2.”
2 Applications of Filters

One crucial decision when dealing with compression filters is choosing which filter best applies to different kinds of data. There are two reasons to use compression: to compress the entire PostScript language stream or to compress just the image data. The following is an analysis of each of the Level 2 compression filters, and a discussion of using the filters for compressing the PostScript language stream and for compressing image data.

In Level 2 the image operator will accept a file, and, therefore, a filtered file, as the data acquisition parameter. The result of this is that somewhat less PostScript language code is required to generate bitmap images. Sample code that uses this technique is presented in section 3.1.

2.1 Analysis of Compression Filters

Because of the algorithms that the compression filters use, not all of the compression filters will produce good compression results with arbitrary data. Most compression schemes rely on repetitions of patterns in the data. If the data does not contain runs of the same data or patterns that occur more than once, the compression filter will not be able to make the file any smaller; some filters might actually make it larger.

In the following examples, compression will be described as \(x:y\), where \(x\) is proportional to the number of bytes of unencoded source, and \(y\) is proportional to the number of bytes of encoded output.

- The best case for run-length compression is a file of all zeros. Here the compression is approximately 64:1 if the file is long enough. The worst case for run-length compression is the sequence 00 FF 00 FF 00_{16}, where there are no runs of data. Here the compression, 127:128, is actually a slight expansion.

- For CCITT fax Group 3, 1-D (K = 0) compression, the best case is again a file of zeros. Each scan line then compresses to about 4 bytes, so if the scan lines are 300 bytes long (8" × 300 dpi / 8 bits), the compression is about 75:1. The worst case is a high-frequency 50% gray (AA AA AA_{16}) of alternating ones and zeros. On that pattern the compression is 2:9, a large expansion.

- For LZW, as for the other filters, the best case is a file of all zeros. If the file is long enough, then the compression approaches 1365:1. The worst case for LZW is a file where no adjacent pair of characters occurs twice. For example, 00 01 02...FF 00 02 04...FE 01 03 05...FF 03 06..._{16}. In this case, the compression of the current LZW implementation is approximately 8:12.
When analyzing the type of data you expect to compress, it might help to understand the way the compression filters consider the input data. The LZW and run-length filters consider input on a byte basis, so the type of data that will compress well is a run of the same bytes (for both run-length and LZW), or repetitions of patterns of bytes (for LZW). In contrast, the CCITT filters consider the data on the bit level and look for runs of on or off bits. This means that even though the string AA AA AA AA (alternating on and off bits) would compress well with either run-length or LZW encoding, it is the worst case for fax encoding.

2.2 Analysis of Transmission Filters

In Level 1, there were two ways to send data to the printer: ASCII and binary. Both methods could be used for images, strings, and any other form of data. In Level 2, the following methods have also been added: ASCII base-85 and the Adobe™ binary communications protocol (BCP).

Most image data and the output of compression filters contain binary data. In addition, applications that use encoding vectors with characters encoded in the low-ASCII (below 32) or high ASCII (above 128) range can contain strings that contain binary data. Binary data is the most compact representation for data, but it is generally a problem in the following cases.

- **Serial or Parallel Channels** – Some of the low-ASCII characters are used for flow control purposes, and data that uses these characters can cause problems.

- **7-bit channels** – The high bit of data is stripped for binary data above 128.

- **Encapsulated PostScript files** – Because EPS files should be able to print on any channel, they should not contain binary data.

The process of converting binary data to ASCII Hex doubles the size of the data. Clearly, this is a sub-optimal characteristic, because the channels that require ASCII data (serial or parallel) are generally slow to begin with. The ASCII base-85 filter that is part of Level 2 has much more palatable expansion characteristics: whereas ASCII Hex data expands binary by 1:2, ASCII base-85 only expands the data by 4:5.

Another method of sending binary data to a printer is with the Adobe binary communications protocol, which exists in some Level 1 and Level 2 devices. (See the Adobe Binary Communications Protocol Specification, for more information.) The BCP works by quoting characters used for flow control, thus making it suitable for serial and parallel ports. The data is not expanded if it does not contain any of the flow control characters, but in the worst case...
the expansion of data is the same as Hex (1:2). In the average case, BCP has better compression characteristics than the ASCII base-85 filter; however, there are a few caveats.

Using BCP over 7-bit channels is a problem, since the flow control characters quoted with BCP are all in the low-ASCII range. BCP is also not suitable for use in EPS files, because not all devices and not all input channels support it. Additionally, enabling and disabling the BCP must be performed as separate jobs from the print job, which might prohibit its use in highly networked environments or in EPS files.

The BCP is suitable for direct-connection applications where the application is aware of the kind of printer on the other end. In these situations, it clearly improves performance. Note that the BCP is not implemented as a filter, but it is mentioned here since it is useful in conjunction with compression filters which generate binary output.

2.3 Compressing the PostScript Language Stream

To determine how well the compression filters perform over the entire PostScript language file, we compressed three different sample files with each of the three compression filters. The sample files were

- a text-only document
- a line art graphics document
- a document with text and line art mixed.

Images will be explored in section 2.4. The files were generated from commercially available applications.
Table 1 Compression data for samples of Level 1 PostScript language files

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (bytes)</th>
<th>Compression (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text only document</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>33,360</td>
<td>—</td>
</tr>
<tr>
<td>LZW</td>
<td>16,620</td>
<td>50.18</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>33,100</td>
<td>0.78</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>66,755</td>
<td>– 100.10</td>
</tr>
<tr>
<td><strong>Line art graphics document</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>20,938</td>
<td>—</td>
</tr>
<tr>
<td>LZW</td>
<td>8,472</td>
<td>59.75</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>21,080</td>
<td>– 0.68</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>36,204</td>
<td>– 72.91</td>
</tr>
<tr>
<td><strong>Mixed text and graphics document</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>6,479</td>
<td>—</td>
</tr>
<tr>
<td>LZW</td>
<td>2,833</td>
<td>56.27</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>6,505</td>
<td>– 0.40</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>10,771</td>
<td>– 66.24</td>
</tr>
</tbody>
</table>

LZW is a good choice for general data compression, that is, compressing the entire PostScript language stream. It has very good “best-case” and “average-case” behavior, in addition to reasonable “worst-case” behavior. Run-length encoding has good best-case behavior, but LZW seems to win in the average case. Run-length encoding has the best worst-case behavior of any of the compression filters, but because LZW is able to compress patterns of bytes and not simply long runs of the same bytes, there are fewer cases where LZW will not compress a file.

Fax encoding should not be used for general data compression, since its worst case is actually a 4.5 times expansion (2:9). Fax encoding expands most PostScript language files, but does have excellent performance for the types of data it was designed to compress—data where there are few white-black transitions and long runs of white. Fax encoding is best reserved for files that are already in fax format or for very sparse line-art images.

In Table 1, three typical PostScript language files were compressed using the fax, run-length, and LZW filters. Compression is represented as a percentage of the original file size. (Negative compression indicates an expansion of data.) In each of these examples, the LZW filter was able to compress the
PostScript language stream significantly. Although the run-length filter was only able to decrease the size of the text file by less than one percent, it also did not expand the data that it could not compress by much.

The fax filter is not an option for this type of data, because it doubles the size of the text file and expands the other files by at least 66 percent.

Encoding the entire PostScript language stream with LZW is often wise in terms of improving time; however, there are other compromises that are made. If the entire file is compressed, any document structuring conventions (DSC) comments are lost. This can sacrifice document portability in some environments.

Note While it is possible to compress each page of a document individually, thereby retaining some of the DSC comments, this will not give the best results when using the LZW compression method, since it is adaptive and is more effective on longer files.

Additionally, the filter mechanism in Level 2 is one of the few features impossible to emulate in a reasonable way on Level 1 printers; therefore, any document that uses the filter mechanism can only be printed on a Level 2 printer. Because of these compromises, if you choose to support compression of the PostScript language stream from your application, you should also provide a user interface that allows users to disable the use of compression.

Another reason to provide a user interface to disable compression is that it might not always improve performance when printing to a slow device over a fast communication channel. That is, since it necessarily takes some CPU time to decode a file, if your printer is CPU bound rather than I/O bound, using compression might not improve performance.

2.4 Compressing Images

Image data is very different from text and line art data. Inherently, there is much more of it and it is an obvious candidate for compression. We chose two image files—a gray-scale image and a monochrome image that consists largely of line art—to compare the various compression filters.
Figures 1 and 2 are examples of images to which the compression filters might be applied. Figure 1 is a gray-scale image, which means that each sample in the image is represented by several bits—eight in this case. Figure 2 is a one-bit per pixel image consisting of line art. These images have very different characteristics and respond differently to the filters.

Refer to Table 2 and notice that Figure 1 is the most difficult to compress. Even the LZW encoding only compresses the file by about 25 percent, and fax roughly doubles the size of the data. Figure 2, however, is much easier to compress. This is the type of image for which the fax compression scheme is best suited. The fax method outperforms the others slightly, not only in terms of file size, but also in terms of overall execution time. Note, however, that even in the case for which fax is optimized, it is only better by a small percent.
Table 2  Compression data for example image files

<table>
<thead>
<tr>
<th>Figure 1: Gray-scale image</th>
<th>Serial</th>
<th>Print times</th>
<th>AppleTalk®</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>N/A</td>
<td>N/A</td>
<td>21</td>
<td>235,777</td>
</tr>
<tr>
<td>LZW</td>
<td>N/A</td>
<td>N/A</td>
<td>20</td>
<td>170,803</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>N/A</td>
<td>N/A</td>
<td>21</td>
<td>230,593</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>N/A</td>
<td>N/A</td>
<td>57</td>
<td>487,678</td>
</tr>
<tr>
<td><strong>Hex data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>509</td>
<td>79</td>
<td>44</td>
<td>485,381</td>
</tr>
<tr>
<td>LZW</td>
<td>368</td>
<td>59</td>
<td>28</td>
<td>351,327</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>498</td>
<td>77</td>
<td>45</td>
<td>474,639</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>1,056</td>
<td>156</td>
<td>104</td>
<td>1,004,870</td>
</tr>
<tr>
<td><strong>ASCII base-85 data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>318</td>
<td>52</td>
<td>26</td>
<td>303,560</td>
</tr>
<tr>
<td>LZW</td>
<td>320</td>
<td>42</td>
<td>24</td>
<td>291,830</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>311</td>
<td>51</td>
<td>38</td>
<td>296,866</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>659</td>
<td>113</td>
<td>80</td>
<td>628,109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure 2: Monochrome image</th>
<th>Serial</th>
<th>Print times</th>
<th>AppleTalk</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>N/A</td>
<td>N/A</td>
<td>6.2</td>
<td>140,032</td>
</tr>
<tr>
<td>LZW</td>
<td>N/A</td>
<td>N/A</td>
<td>2.9</td>
<td>26,892</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>N/A</td>
<td>N/A</td>
<td>2.8</td>
<td>43,020</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>N/A</td>
<td>N/A</td>
<td>2.9</td>
<td>23,191</td>
</tr>
<tr>
<td><strong>Hex data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>299</td>
<td>40.0</td>
<td>11.8</td>
<td>287,956</td>
</tr>
<tr>
<td>LZW</td>
<td>55</td>
<td>7.2</td>
<td>3.9</td>
<td>54,560</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>90</td>
<td>11.0</td>
<td>4.9</td>
<td>87,816</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>47</td>
<td>6.3</td>
<td>3.6</td>
<td>46,915</td>
</tr>
<tr>
<td><strong>ASCII base-85 data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original file</td>
<td>186</td>
<td>24.0</td>
<td>8.6</td>
<td>180,110</td>
</tr>
<tr>
<td>LZW</td>
<td>34</td>
<td>5.0</td>
<td>3.3</td>
<td>34,421</td>
</tr>
<tr>
<td>Run-length encoding</td>
<td>55</td>
<td>7.1</td>
<td>3.7</td>
<td>55,199</td>
</tr>
<tr>
<td>Fax Group 3</td>
<td>29</td>
<td>4.4</td>
<td>3.3</td>
<td>29,648</td>
</tr>
</tbody>
</table>
Table 2 shows a comparison of different compression and transmission methods for gray-scale eight-bit and one-bit images. Print time does not include time to compress the image.

Because of the poor suitability of the fax filter for most types of data, it is best reserved for sending files that are already in fax format or for applications that generate one-bit images with a large amount of white space.

In addition to the run-length, fax, and LZW filters, the DCT filter can be used to compress color or black and white images. The DCT filter is a lossy filter (that is, when the data is decoded, it might not correspond exactly to the image that was encoded) and, therefore, cannot be used on the PostScript language stream. It can be used on color images and achieve compression ratios of 30:1 with little degradation of the image. This is discussed in section 3.13.3 of the PostScript Language Reference Manual, Second Edition and Technical Note #5116, “Supporting the DCT Filters in PostScript Level 2.” There are also several ways to compress color images without using the DCT filter.

A new feature in PostScript Level 2 is the idea of a color space. One of the color spaces defined in Level 2 is an indexed color space. Programs that deal with colors in terms of palettes might find the indexed color space useful because it is essentially a palette representation, or color lookup table, for images.

The idea behind the indexed color space is that a certain limited number of colors are used in a particular image. Instead of representing the data sent to the printer as 24-bit image data, one could use an indexed color space to represent a limited number of colors in a smaller number of bytes. For instance, 256 colors can be represented in one byte. If the expansion from color index value to image color data is performed on the printer, much less data must be transmitted, improving overall performance.

In addition, the LZW or run-length filters are generally able to compress the indexed data even further. This can result in significant compression of color images, without having to resort to the complexity of the DCT filter. Some of the other color spaces defined in Level 2 are generally suited to using the DCT or LZW filters. They are the RGB, CMYK, HSB, DeviceGray, and CIE color spaces.

For more information on the indexed color space and on emulating its functionality, refer to Technical Note #5121, “Improving Image Performance by Using Color Lookup Tables.”
3 Details of Using Filters

As mentioned earlier, a filter is a special kind of file object that can be layered on top of another file to transform data being read from or written to that file. One of the interesting aspects of filters is that they use not only files as data sources and targets, but also strings and procedures. Below is an example using the ASCII85Decode filter to decode a PostScript language stream.

```
currentfile /ASCII85Decode filter cvx exec
02uY&ATKGSDf%p#++>PVnF(K0!@ru=$DI#UJFV(0JFVdDg#6D
|e*OBo;rD7R#5
E2f;+/bi+EM47G7AV,Dg-/B4V->
```

The earlier sections of this document provide motivation for why you would want to use filters. The following sections cover how you use this new feature.

3.1 Concatenating Filters

Because filters can be layered on top of files, and because filters are actually special kinds of files, it is possible to have several filters operate in series on a given file stream.

One of the most common uses for concatenating filters is to convert the binary output of the compression filters to ASCII so that the data can be transmitted over a 7-bit communication channel, such as a serial line or through electronic mail. In this way, data that would normally pose a problem in terms of data transmission time, space, and format can be sent over restricted channels while maintaining reasonable performance.

It usually does not make sense to concatenate two of the compression filters together (for instance, run-length and LZW), because the output of the compression filters is binary data, which does not compress well.

When concatenating filters, pay attention to the order in which the filters are invoked. If we use the LZW algorithm to compress some data and to transmit the result over a serial channel, but the channel is only 7-bits, we will need to also encode the data in ASCII base-85 format.

When encoding the data on the host, the LZW compression method should be run on the data first. The result of that transformation can then be converted to ASCII base-85. This is fairly straightforward, and the result can be sent over a serial channel to a PostScript interpreter. When decoding the data in the PostScript interpreter, the appropriate filters should be applied in reverse order: using ASCII85Decode and then LZWDecode.

The following is an example of LZW compression and ASCII base-85 encoding applied to image data. Note that it uses the new Level 2 form of the image operator, which accepts a file stream directly.
There is another, less intuitive, example of concatenating filters. Suppose we want to use the PostScript interpreter to encode a file for us, and we want to have the resultant file be both LZW and ASCII base-85 encoded. The order in which the output (encoding) filters are invoked is not necessarily obvious. Below is an example of a PostScript language program that takes as input a file of ASCII text (the remainder of `currentfile`), and prints the ASCII base-85 encoding of the LZW encoding of that input text on the standard output file.

```
/tmpstr 100 string def
/outstream
  (%stdout) (w) file
  /ASCII85Encode filter
  /LZWEncode filter def

/instream
  currentfile def

/convert {
  { instream
    /tmpstr readstring not
    {
      outstream exch writestring
      exit
    } if
    outstream exch writestring
  } loop
  outstream closefile
  flush
} bind def

convert
```

This is some text that I want to convert to LZW and ASCII85.
4 LZW Issues

The LZW compression method is adaptive. It encodes more compactly when its input sequence is drawn from a skewed (non-uniform) input distribution. In sampled natural images, the set of sample-to-sample differences across an image row is usually a more highly skewed distribution than the original set of samples.

The idea of applying LZW encoding to image sample differences, rather than to the samples themselves, is implemented as predictor value 2 of the TIFF LZW compression scheme. This predictor value was not part of TIFF version 5.0, but will likely become part of a later revision.

4.1 Optional LZW Parameters

The PostScript language LZW filters accept optional argument dictionaries that are not documented in the *PostScript Language Reference Manual, Second Edition*. All entries in the *LZWEncode* and *LZWDecode* dictionaries are optional, and have default values. The ones that currently have an effect are:

- **Predictor integer** (default 1). If *Predictor* is 1, normal LZW encoding or decoding is done. If *Predictor* is 2, normal LZW encoding is preceded by differencing the data to be encoded, while normal LZW decoding is followed by a complementary undifferencing operation.

- **Columns integer** (default 1). Only has effect if *Predictor* is 2. *Columns* is the number of samples in a sampled row. The first sample of each row is not differenced; every other one is differenced with the prior sample in its row. Each row begins on a byte boundary. Any extra bits beyond \((\text{Columns} \times \text{Colors} \times \text{BitsPerComponent})\) that might be needed to complete a multiple of 8 are not differenced.

- **Colors integer** (default 1). Only has effect if *Predictor* is 2. The number of interleaved colors per sample. Each color is differenced with the same color in the prior pixel. Legal values are 1, 2, 3, or 4.

- **BitsPerComponent integer** (default 8). Only has effect if *Predictor* is 2. The number of bits used to represent each color component of a sample. Legal values are 1, 2, 4, or 8.
There is another optional LZW parameter that does not relate to **Predictor**:

- **EarlyChange** integer (default 1). The TIFF 5.0 specification can be interpreted to imply that code word length increases are postponed as long as possible. However, the LZW sample code distributed by Aldus increases the code word length one code word earlier than necessary. Both interpretations are implemented in the PostScript interpreter.

  If **EarlyChange** is 0, code word length increases are postponed as long as possible. If it is 1, they occur one code word early. A decode filter’s **EarlyChange** parameter must match the **EarlyChange** parameter used by the encode filter that generated its input data.

### 4.2 Licensing Issues

The LZW compression method is the subject of United States patent number 4,558,302 and corresponding foreign patents owned by the Unisys Corporation. Adobe Systems has licensed this patent for use in its products. Independent software vendors (ISVs) may be required to license this patent to develop software using the LZW method to compress PostScript language programs or data for use with Adobe products. Unisys has agreed that ISVs may obtain such a license for a modest one-time fee.

More information can be obtained from:

Welch Licensing Department  
Law Department  
M/S C2SW1  
Unisys Corporation  
BlueBell, PA 19424
Appendix: Changes Since Earlier Versions

Changes since August 13, 1991 version

- Document was reformatted in the new document layout and minor editorial changes were made.
Index

B

BCP 7
binary communications protocol. See BCP 7

C

color space 13
indexed 13
compression
black and white image 13
disable 10
filter 5–17
gray-scale image 11
images 10–13
monochrome image 11
PostScript Level 2 5–17

currentfile 15

F

filters 5
application 6–15
compression 6, 14
CCITT fax Group 3 6
LZW 6
run-length 6
concatenating 14–15
encoding 14
decoding 5
encoding 5
LZW 7, 13

Parameters 16
Predictor 16
LZW encoding
fax 7
run-length 7
transmission 7

7-bit channel 7
7-bit channel using BCP 8
ASCII Hex 7
encapsulated PostScript files 7
serial or parallel channel 7
using 14–15
Level 2 10

I

image 6, 14
image file
compression data 12
indexed color space 13

L

lossy filter 13
LZWDecode 16
LZWEncode 16

P

palettes 13
PostScript language stream
compressing 8–10
fax encoding 9
LZW 9
run length encoding 9