Inhalt

Teil 2: Textverarbeitung

22 Graphics Interchange Format (GIF)

Das 1987 von der Firma CompuServe definierte GIF89a-Format wurde 1989 um einige Zusätze unter dem Begriff GIF89a erweitert. In diesen GIF-Dateien lassen sich mehrere Bilder (wird für animierte GIF-Bilder benutzt) und transparente Bilder hinterlegen. Populär wurde das GIF-Format durch das World Wide Web, da die HTML-Spezifikation Bilder in diesem Format unterstützt. Nachfolgend finden Sie die Beschreibung des GIF89a-Formats, beim GIF87a-Format fehlen einige dieser Bildelemente.

Eine GIF89a-Datei besteht (wie die GIF98a-Datei) aus mehreren Blöcken zur Aufnahme der Grafiken und zusätzlich benötigter Daten. Dabei werden ähnlich wie beim TIFF-Format Tag-Felder benutzt. Die Blöcke lassen sich in drei Gruppen zusammenfassen:

- Control-Blocks
- Graphic Rendering-Blocks
- Special Purpose-Blocks

Die Control-Blocks (z. B. *Header*, *Logical Screen Descriptor*, *Graphics Control Extension*, *Trailor*) enthalten Informationen zur Steuerung der Bildwiedergabe. In den *Graphic Rendering*-Blocks (z. B. *Image Descriptor*, *Plain Text Extension* etc.) finden sich die eigentlichen Daten zur Ausgabe der Grafiken. Die *Special Purpose*-Blocks (z. B. *Comment Extension*, *Application Extension*) dienen zur Aufnahme herstellerspezifischer Informationen und sollten vom GIF-Decoder überlesen werden.

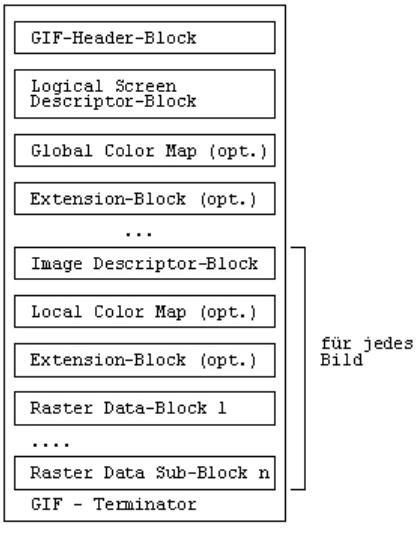

Abbildung 22.1 Struktur einer GIF-Datei

Die Blockgröße ist mit Ausnahme der Subblocks zur Aufnahme von Daten fest definiert. Enthält der Block ein *Block Size*-Feld, gibt dieser Wert die Zahl der folgenden Bytes im Block an. Diese Größe umfaßt nicht einen eventuell folgenden Terminator. Abbildung 22.1 gibt den strukturellen Aufbau einer GIF-Datei wieder.

Für jedes Bild der GIF-Datei werden ein Image Descriptor und ein oder mehrere *Raster Data*-Blocks abgelegt. Ein *Local Color Map*-Block kann optional auftreten. Innerhalb der Blockstruktur dürfen sogenannte Subblocks auftreten (z.B. Blöcke mit Bilddaten). Diese Subblocks bestehen aus einem Längenbyte, welches die Zahl der Folgebytes im Block (0–255) definiert. Daran schließt sich der Datenbereich mit 0 bis 255 Byte an.

Der GIF-Header

Eine GIF-Datei muß immer mit einem Header beginnen, der in beiden GIF-Versionen gleich ist. Dieser Header umfaßt 6 Byte und besitzt die in Tabelle 22.1 definierte Struktur (siehe auch vorhergehendes Kapitel).

Offset	Bytes	Feld
00H	3	Signatur 'GIF'
03H	3	Version '87a' oder '89a'

Tabelle 22.1 GIF87a- bzw. GIF89a-Header

In den ersten drei Bytes steht eine Signatur für den Header ('GIF'). Daran schließen sich drei Bytes mit der GIF-Version (GIF87a oder GIF89a) an. Der Header darf nur einmal innerhalb der GIF-Datei auftreten, und es muß der erste Block in der Datei sein. Die folgende Blockstruktur ist für GIF87a und GIF89a gleich, wobei in GIF89a zusätzliche Blocktypen definiert wurden. Weiterhin wurden einige Flags in der GIF89a-Version etwas modifiziert. An den betreffenden Stellen der Beschreibung finden Sie Hinweise auf die Änderungen. Auch ältere GIF87a-Decoder können eine GIF89a-Datei bearbeiten, wobei aber die erweiterten Blocktypen zu überlesen sind und eventuell Informationen verlorengehen.

Die GIF-Blocks

An den Header schließen sich verschiedene Blöcke mit Informationen an. Die verschiedenen Blocktypen werden nachfolgend beschrieben.

Der Logical Screen Descriptor-Block

An den Header muß sich (ab Offset 06H) der 7 Byte große *Logical Screen Desciptor*-Block mit den Daten des logischen Bildschirms anschließen. Auch dieser Block wird in GIF87a und GIF89a verwendet. Die Daten des Blocks gelten für die komplette GIF-Datei und sind in folgender Struktur abgelegt:

Bytes	Feld
2	Logical Screen Width
2	Logical Screen Height
1	Resolution-Flag
1	Background Color Index
1	Pixel Aspect Ratio

Tabelle 22.2 Struktur des Logical Screen Descriptor-Blocks

Der erste Eintrag umfaßt einen 16-Bit-Wert und gibt die Breite des logischen Bildschirms in Pixel an. Dabei wird die Intel-Notation (Low-Byte first) zur Speicherung benutzt. Das nächste Feld mit der Bildschirmhöhe enthält ebenfalls einen 16-Bit-Wert. Die beiden Werte für die Bildabmessungen beziehen sich auf einen virtuellen Bildschirm, dessen Nullpunkt sich in der oberen linken Ecke befindet.

Ab Offset 04H findet sich im *Logical Screen Descriptor*-Block ein Bitfeld (Resolution-Flag) mit der Kodierung gemäß Abbildung 22.2.

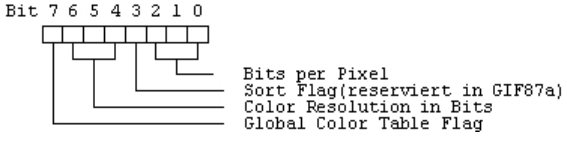

Abbildung 22.2 Kodierung des Resolution-Flag

Das oberste Bit 7 markiert, ob eine *Global Color Map* (Definition der Farbpalette) existiert. Ist Bit 7 auf 1 gesetzt, folgt auf den *Logical Screen Descriptor*-Block die *Global Color Map*. Ist dieses Bit auf 0 gesetzt, fehlt die *Global Color Map*, und die Bits für den *Background Color Index* (Hintergrundfarbe) besitzen keine Bedeutung.

In den Bits 4 bis 6 wird festgehalten, wie viele Bits zur RGB-Darstellung einer primären Farbe (Color Resolution) in der Farbtabelle zur Verfügung stehen. Der Wert in den Bits ist jeweils um 1 zu erhöhen. Der Wert 3 besagt, daß pro primäre Farbe 4 Bit in der entsprechenden Palette benutzt werden.

Bit 3 ist in der GIF87a-Spezifikation noch als reserviert markiert und muß auf 0 gesetzt werden. In der GIF89a-Spezifikation enthält dieses Bit das Sort-Flag. Der Wert 1 signalisiert, daß die *Global Color Table* sortiert vorliegt. Die Sortierung erfolgt dabei nach den Farben mit absteigender Bedeutung, d.h., die häufiger verwendeten Farben befinden sich am Beginn der Palette. Dies ist für Decoder, die nur wenige Farben unterstützen, hilfreich. Mit dem Wert 0 liegt die *Global Color Map* unsortiert vor.

In den Bits 0 bis 2 wird in GIF87a die Zahl der Bit pro Pixel angegeben. Der Wert ist dabei um 1 zu erhöhen. Der maximale Wert 7 bedeutet, daß 8 Bit pro Pixel verfügbar sind. Damit lassen sich 256 unterschiedliche Farben in einem Bild verwenden. In der GIF89a-

Spezifikation wird angegeben, daß bei gesetztem *Color Table*-Flag die drei Bits die Größe der globalen Farbpalette (global color table size) in Byte definieren. Letztlich handelt es sich aber um den gleichen Wert (Bit pro Pixel), da sich die Größe der Farbpalette zu

```
Color Table Size := 2 ** (value + 1)
```

berechnen läßt. Der Wert sollte auch dann gesetzt werden, wenn die GIF-Datei keine *Global Color Map* enthält. Dies ermöglicht dem Decoder, den entsprechenden Graphikmodus einzustellen.

Ab Offset 05H findet sich im *Logical Screen Descriptor*-Block ein Byte mit der Kodierung der Hintergrundfarbe (Background Color Index). Diese Hintergrundfarbe wird aus den 256 möglichen Farben ausgewählt. Die Hintergrundfarbe wird für die Teile des Bildschirms benutzt, die nicht durch die Bitmap belegt werden. Dieser Wert wird zum Beispiel benutzt, um transparente GIF-Grafiken für HTML-Dokumente zu erstellen. Falls das *Global Color-Flag* gelöscht (0) ist, sollte das Byte auf 0 gesetzt und vom Decoder überlesen werden.

Das Byte ab Offset 06H im *Logical Screen Descriptor*-Block definiert das *Pixel Aspect Ratio*. Die Belegung wird jedoch in GIF87a und GIF89a unterschiedlich gehandhabt. In GIF87a gilt die Kodierung gemäß Abbildung 22.3.

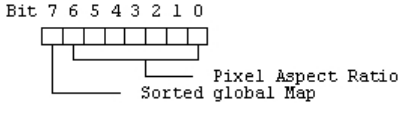

Abbildung 22.3 Kodierung des Pixel Aspect Ratio-Flag (GIF87a)

In GIF87a wird Bit 7 als Sorted *Global Color Map*-Flag verwendet. Die restlichen Bits geben das *Pixel Aspect Ratio* des ursprünglichen Bildes an.

In GIF89a ist das Sorted *Global Color Map*-Flag im *Resolution*-Flag integriert (siehe Abbildung 22.2). Daher werden alle Bits des *Pixel Aspect Ratio*-Feldes benutzt. Ist der Wert des Feldes ungleich 0, läßt sich das Verhältnis der Bildabmessungen zu:

```
Aspect Ratio = (Pixel Aspect Ratio + 15) / 64
```

berechnen. Das + ist als Quotient der Bildbreite zur Bildhöhe definiert. Die Spezifikation erlaubt einen Bereich zwischen 4:1 bis 1:4 in 1/64 Schritten.

Der Global Color Map-Block

Im Anschluß an den Logical *Screen Decriptor*-Block kann optional ein Block mit der *Global Color Map* (globale Farbpalette) gespeichert sein. Dies ist immer dann der Fall, wenn im vorhergehenden *Logical Screen Descriptor*-Block das Bit 7 des *Resolution*-Flag gesetzt ist.

Im *Color Map*-Block wird die globale Farbtabelle für die nachfolgenden Bilder spezifiziert. Diese globale Farbpalette wird immer dann verwendet, wenn ein Bild keine lokale Palette besitzt.

Für jeden Bildpunkt sind maximal 8 Bit vorgesehen, womit sich lediglich 256 Farben oder Graustufen abbilden lassen. Eine True-Color-Darstellung (Echtfarbenanzeige) ist nicht vorgesehen. Die *Global Color Map* enthält für jede der 256 Farben ein Tripel (3 Byte) mit den Grundfarben Rot, Grün und Blau (Abbildung 22.4).

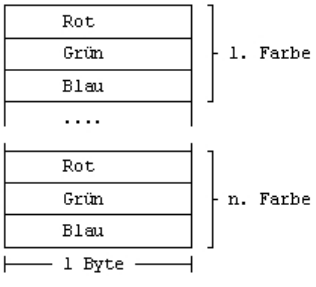

Abbildung 22.4 Struktur der Global Color Map

Jeder dieser drei Grundfarben ist ein Byte zugeordnet. Der in dem Byte abgelegte Wert bestimmt den Farbanteil (Intensität) in der Mischfarbe. Mit drei Byte pro Farbe lassen sich 16 Millionen Farben darstellen, wobei jedoch nur 256 Farben über die Palette im Bild genutzt werden können. Der Wert eines Bildpunktes wird als Offset in die Farbtabelle interpretiert, und die Grafikkarte generiert dann den zugehörigen Farbwert. Die Größe der Farbtabelle und die Zahl der Bits pro Farbe werden ebenfalls im *Resolution*-Flag spezifiziert *(Palette Size = 3 Byte * 2 ** Bits pro Pixel)*.

Der Image Descriptor-Block

Jedes Bild innerhalb der GIF-Datei muß durch einen 10 Byte großen *Image Descriptor*-Block eingeleitet werden. Der Block wird in beiden GIF-Varianten verwendet. Die Struktur dieses Blocks ist Tabelle 22.3 zu entnehmen.

Bytes	Feld
1	Image Separator Header (ASCII 2CH = ',')
2	Koordinate linker Rand
2	Koordinate oberer Rand
2	Bildbreite
2	Bildhöhe
1	Flags

Tabelle 22.3 Struktur eines Image Descriptor-Blocks

Der Block enthält die wichtigsten Daten eines Bildes wie Abmessungen, Koordinaten der linken oberen Ecke etc. Das erste Byte wird mit dem Separator (',' 2CH) gefüllt. Die beiden folgenden Felder geben die Bildkoordinaten für die obere linke Ecke des Bildes in Pixel an und umfassen jeweils 16 Bit (Unsigned Word). Diese Daten beziehen sich auf den logischen Bildschirm.

Ab dem dritten Feld (Offset 05H im Block) wird die Breite des Bildes (Image Width) in Pixel angegeben. Das folgende Wort definiert die Bildhöhe in Pixel. Beide Werte werden als unsigned Word definiert. Das letzte Byte dient zur Aufnahme verschiedener Flags, die gemäß Abbildung 22.5 kodiert sind.

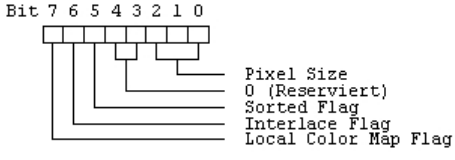

Abbildung 22.5 Kodierung des Flags im Image Descriptor-Block (IDB)

Das oberste Bit spezifiziert, ob sich eine lokale Palette (Local Color Map) an den *Image Descriptor*-Block anschließt. Ist Bit 7 = 1, folgt dem *Image Descriptor*-Block eine *Local Color Map*. In diesem Fall werden diese Daten für die Farbpalette des folgenden Teilbildes verwendet. Ein Decoder muß dann die *Global Color Map* sichern und die neuen Daten benutzen. Nach Bearbeitung des Bildes sind die Daten der *Global Color Map* zu restaurieren.

Bit 6 definiert das *Interlaced*-Flag, d.h., das Grafikbild kann sequentiell (Bit 6 = 0) oder interlaced (Bit 6 = 1) gespeichert sein. Der sequentielle Modus gibt Zeile für Zeile des Bildes aus. Im Gegensatz dazu wurde der *Interlaced Mode* geschaffen, um bei einer Übertragung per Telefonleitung möglichst schnell über ein Rohbild zu verfügen. In diesem Modus wird bei jedem Durchlauf nur jede achte Zeile übertragen und ausgegeben, so daß nach dem ersten Durchlauf zunächst die 0., 8., 16. usw. Zeile vorliegt. Die fehlenden Zeilen werden dann in den folgenden Durchläufen in der Folge der Anfangsreihen 4, 2, 1, 3, 5, 7 ergänzt. Beim zweiten Durchlauf erhält man demnach die 4., 12., 18. usw. Zeile und beim dritten Durchlauf die 2., 6., 10. usw. Zeile.

Bit 5 gibt in GIF89a an, ob die lokale Farbtabelle nach Farben sortiert ist. Bei gesetztem Bit werden die wichtigsten Farben zuerst in der Palette abgelegt. Die am häufigsten verwendete Farbe sollte zuerst gespeichert werden. Beim Flag = 0 werden die Farben der Palette unsortiert abgelegt. Bit 5 wird aber nur in wenigen Anwendungen benutzt, da die offizielle GIF87a-Dokumentation dieses Bit nicht erwähnt und zu 0 setzt.

Anmerkung: Die Bits 3 bis 5 sind in GIF87a als reserviert markiert und müssen auf 0 gesetzt werden.

Die unteren Bits 0 bis 2 definierten die Zahl der Bits pro Bildpunkt (GIF87a und GIF89a). Der Wert der Bits ist um 1 zu erhöhen. Mit einem Eintrag von 7 werden 8 Bit pro Pixel verwendet. Aus diesem Wert läßt sich die Zahl der Einträge in der Farbpalette (2**n) und damit deren Größe (Einträge * 3 Byte) berechnen. Der Wert dieser drei Bits sollte auf 0 gesetzt werden, falls keine *Local Color Map* folgt.

Der Local Color Map-Block

Neben der globalen Farbtabelle lassen sich auch vor den *Raster Image*-Blöcken eines Teilbildes lokale Farbpaletten definieren. Ist das *Local Color Table*-Flag im *Image Descriptor*-Block gesetzt, folgt auf diesen Block der *Local Color Map*-Block. Dann muß der Decoder die Daten der *Global Color Map* sichern. Erst nach Bearbeitung der Bilddaten ist die *Global Color Map* zu restaurieren. Der Aufbau der *Local Color Map* stimmt mit dem der *Global Color Map* überein (siehe Abbildung 22.4). Eine GIF-Datei kann mehrere *Local Color Map*-Blocks enthalten. Die *Local Color Map*-Blocks sind in beiden GIF-Spezifikationen definiert.

Der Extension-Block

An den Block mit der Local Color Map kann sich ein optionaler *Extension*-Block (Erweiterungsblock) anschließen. Über diese *Extension*-Blocks wurde in GIF89a ein Weg für zukünftige Erweiterungen des Formats geschaffen. In GIF87a waren diese *Extension*-Blocks zum Speichern von Informationen über das bilderzeugende Gerät, die benutzte Software, die Ausrüstung zur Bildabtastung (Scanner) etc. vorgesehen. Die *Extension*-Blocks haben den in Tabelle 22.4 beschriebenen Aufbau.

Bytes	Feld
1	Extension-Block Header (ASCII 21H = '!')
1	Funktionscode (0 .. 255)
1	Länge Datenblock 1 (in Byte)
n	Datenblock 1
1	Länge Datenblock 2
n	Datenblock 2
...	...
1	Länge Datenblock n
n	Datenblock n
1	00H als Terminator

Tabelle 22.4 Struktur eines Extension-Blocks (GIF89a)

Das erste Byte des Erweiterungsblocks enthält das Zeichen ! als Signatur. Darauf folgt ein Byte mit dem Funktionscode, der die Art der nachfolgenden Daten definiert.

Anschließend folgt der Datenbereich, der mehrere Records der gleichen Struktur enthalten kann. In jedem dieser Records findet sich im ersten Byte die Angabe über die Anzahl der nachfolgenden Datenbytes, womit ein Datenbereich die maximale Länge von 255 Byte besitzen kann.

Bei längeren Datensequenzen sind mehrere Subblocks zu speichern. Das Ende des *Extension*-Blocks wird durch ein Nullbyte (00H) markiert.

Die Belegung der Funktionscodes ist in GIF87a nicht definiert. Auch der interne Aufbau wurde dem jeweiligen Entwickler des Encoders überlassen. In GIF89a ändert sich die Situation, denn die Spezifikation beschreibt verschiedene *Extension*-Blocks. Am Ende dieses Kapitels wird der Aufbau dieser *Extension*-Blocks für GIF89a vorgestellt.

Der Raster Data-Block

Die eigentlichen Bilddaten (Image Data) werden in einem oder mehreren *Raster Data*-Blocks abgelegt. Ist im *Image Descriptor*-Flag (Abbildung 22.5) das Flag für die *Local Color Map* gesetzt, folgen die Bilddaten im Anschluß an die Farbtabelle. Fehlt die *Local Color Map*, schließen sich die Daten an den *Image Descriptor*-Block oder an einen *Extension*-Block an. Der erste *Raster Data*-Block besitzt den Aufbau gemäß Tabelle 22.5.

Bytes	Feld
1	Code Size
1	Bytes im Datenblock
n	Datenbytes

Tabelle 22.5 Struktur eines Raster Data-Blocks

Im ersten Byte des ersten Blocks steht ein Byte, welches als *Code Size* bezeichnet wird. Dieser Wert definiert die minimale Codelänge, die für die Darstellung der Pixel bei der LZW-Komprimierung benötigt wird. Dieses Byte wird zur Initialisierung des Decoders benutzt. In der Regel stimmt der Wert mit der Zahl der Farbbits pro Bildpunkt überein. Nur bei Schwarzweißbildern (Bit pro Pixel = 1) muß *Code Size* = 2 gewählt werden.

Das zweite Byte definiert die Zahl der Bytes im nachfolgenden Datenblock. Der Wert kann dabei zwischen 0 und 255 liegen.

Ab dem dritten Byte folgen die komprimierten Bilddaten. Umfassen die Bilddaten mehr als 255 Byte, folgen weitere Subblocks mit den restlichen Daten (siehe Abschnitt Subblocks mit Rasterdaten). Nach der Dekodierung ist das Bild, beginnend in der linken oberen Ecke, von links nach rechts und von oben nach unten aufzubauen. Zur Komprimierung wird der LZW-Algorithmus von Lempel Ziv und Welch in leicht modifizierter Form benutzt (siehe folgenden Abschnitt).

Die LZW-Komprimierung

In Programmen zur Bearbeitung von GIF-Grafikdateien wird die Technik des LZW-Verfahrens (Lempel-Ziv-Welch) benutzt. Der Algorithmus versucht den zu komprimierenden Zeichenstrom in Teilketten zu zerlegen und diese in einer Tabelle zu speichern. Anschließend werden nur die Indizes in die betreffende Tabelle als Ausgabecodes gespeichert. Anhand dieser Ausgabecodes läßt sich dann der ursprüngliche Zeichenstrom wieder generieren. Der prinzipielle Aufbau ist in Abbildung 22.6 wiedergegeben.

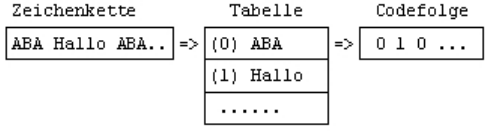

Abbildung 22.6 LZW-Encoding

Die Zeichenfolge aus Abbildung 22.6 läßt sich in Teilstrings (ABA, Hallo etc.) aufteilen. Diese Zeichenfolgen tauchen auch in der Kodierungstabelle auf. An Stelle der Teilstrings wird dann jeweils der zugehörige Tabellenindex als Ausgangscode übertragen. Aus der Ursprungszeichenkette mit 11 Zeichen (ABAHalloABA) wird dann eine Codefolge mit 3 Zeichen (010). Mit diesem recht einfachen Prinzip lassen sich beliebige Zeichenfolgen komprimieren.

Allerdings sind noch zwei Probleme zu beheben. Einmal wird die Codetabelle sich nur in den seltensten Fällen a priori bestimmen lassen. Sie wird aber auch nicht mit dem Code gespeichert und steht bei der Dekomprimierung nicht mehr zur Verfügung. Deshalb muß der Algorithmus bei der Komprimierung und Dekomprimierung die Tabelle jeweils selbst aufbauen. Die zweite Schwierigkeit betrifft die Größe der Tabelle. Theoretisch muß diese Tabelle unendlich groß sein, um alle Zeichenkombinationen aufzunehmen. Aus praktischen Gründen wird jedoch die Größe der Tabelle limitiert. Mit der Implementierung des LZW-Algorithmus für GIF-Dateien werden diese Probleme behoben.

Zur Diskussion des Verfahrens möchte ich vorher noch einige Begriffe erläutern.

<new>: Speicherzelle mit dem letzten gelesenen Zeichen

<old> : Speicherzelle mit dem vorletzten gelesenen Zeichen

[..] : Zeichenpuffer zum Aufbau der Tabelle

[..]K : Zeichenpuffer mit angefügtem Zeichen *K*

Für den Aufbau der Tabelle ist dabei der Puffer *[..]* von Bedeutung. Dieser Puffer kann einzelne Zeichen oder komplette Zeichenketten aufnehmen. Ziel des Algorithmus ist es, komplette Zeichenketten zu bilden, die noch nicht in der Tabelle abgespeichert sind. Tritt dieser Fall auf, wird der betreffende Teilstring in die Tabelle angefügt und ein Code für die

letzte vorhandene Teilkette ausgegeben. So baut sich die Kodierungstabelle selbst auf und kann anhand des Ausgabecodes jederzeit bei der Dekomprimierung rekonstruiert werden.

Weiterhin stellt sich die Frage nach der Initialisierung der Tabelle zu Beginn der Komprimierung. Die Größe muß willkürlich festgelegt werden. Mit 12 Bit lassen sich zum Beispiel 4096 Einträge kodieren. Jeder Eintrag speichert später eine Zeichenfolge. Als Ausgabecode werden dagegen nur die 12-Bit-Indizes der Tabelle gespeichert. Dies führt zu der gewünschten Komprimierung. Mit etwas Kenntnis über die möglichen Zeichen im Eingabecode läßt sich die Tabelle teilweise mit Anfangswerten initialisieren. Nehmen wir an, die Eingangszeichen stammen aus dem Alphabet der Großbuchstaben. Dann können die ersten 26 Einträge der Tabelle mit den Codes der Zeichen (0 = A, 1 = B, 2 = C etc.) belegt werden.

Der LZW-Algorithmus zur Komprimierung läßt sich dann mit folgenden Pseudocodeanweisungen beschreiben.

```
initialize table
Clear Buffer [..]
WHILE Not EOF DO
 Read Code in K
 IF [..]K in Tabelle?
  [..] <- [..]K
 ELSE
  Add [..]K in Tabelle
  Write Tabellenindex von [..]
  [..] <- K
 ENDIF
WEND
Write Tabellenindex von [..]
```

Zuerst ist die Tabelle zu initialisieren und der Puffer *[..]* zu leeren. Dann wird die Folge von Eingabecodes zeichenweise gelesen. Das gerade gelesene Zeichen der Eingabefolge wird rechts vom Puffer als Postfix beigestellt (*[..]K*). Nun ist zu prüfen, ob der so gebildete Teilstring *[..]K* bereits in der Tabelle vorkommt. In diesem Fall bearbeitet der Algorithmus das nächste Zeichen. Kommt der String noch nicht in der Tabelle vor, beginnt der nächste Schritt. Als erstes wird der neue String *[..]K* in der Tabelle an der ersten freien Position angehängt. Dann wird der Tabellenindex des Teilstrings *[..] (not [..]K)* als Ausgabecode gespeichert. Abschließend ist der alte Pufferinhalt durch das zuletzt gelesene Zeichen *K* zu ersetzen. Dann wird das nächste Zeichen gelesen, und die Bearbeitung beginnt erneut. Erst wenn alle Zeichen gelesen wurden, ist der Code des Puffers mit dem Reststring aus der Tabelle zu ermitteln und auszugeben.

Dieser Sachverhalt soll nochmals an einem kleinen Beispiel verdeutlicht werden. Aus der Menge der Buchstaben *A*, *B*, *C*, *D* soll die Zeichenkette *ABACABA* gebildet werden. Die Tabelle läßt sich dann in den ersten vier Einträgen mit den Buchstaben *A*, *B*, *C*, *D* initialisieren. Die folgende Abbildung gibt die einzelnen Schritte bei der Komprimierung wieder:

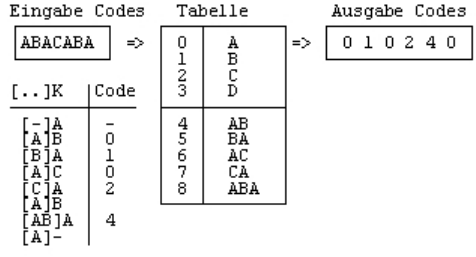

Abbildung 22.7 LZW-Kompression

Aus der Zeichenfolge ABACABA wird dann die Codefolge 010240. Diese Folge umfaßt nur noch 6 Codes, während der ursprüngliche String 7 Zeichen enthält. Bei längeren Zeichenketten ergeben sich wesentlich bessere Komprimierungsverhältnisse.

Bei der Dekomprimierung muß nun die Codefolge wieder in den Ausgabestring überführt werden. Hierzu wird ein Algorithmus verwendet, der sich mit folgenden Pseudocodeanweisungen beschreiben läßt:

```
Initialize Stringtabelle
<code> := 1. Code Byte
Tabelle[code] -> Ausgabestring
<old> := <code>
WHILE NOT EOF DO
 <code> := Next Code Byte
 IF Tabelle[code] belegt?
  Tabelle[code] -> Ausgabestring
  [..] <- Tabelle[old]
  K    <- 1. Zeichen (Tabelle[code])
  Write [..]K in Tabelle
  <old> := <code>
 ELSE
  [..] <- Tabelle[old]
  K    <- 1. Zeichen (..)
  [..]K -> in Ausgabestring
  [..]K -> Tabelle
  <old> := <code>
 END IF
WEND
```

Bezüglich der verwendeten Nomenklatur gilt: *<old>* und *<code>* sind zwei Variablen, in denen das vorletzte und das letzte gelesene Zeichen stehen. Mit *[..]* wird ein Zeichenpuffer definiert, der Strings aus der Stringtabelle enthält. Die Stringtabelle wird mit *Tabelle[..]* bezeichnet. Mit *-> Ausgabestring* werden Strings in die Ausgabeeinheit geschrieben.

Der Algorithmus initialisiert zuerst die Stringtabelle mit Basiswerten. Dann wird das erste Element der Codefolge gelesen. Der Wert dient als Index in die Stringtabelle. Der betreffende String (der Index zeigt auf den initialisierten Teil der Tabelle) wird dann in den Ausgabestrom geschrieben. Nun wird der gelesene Code in der Variablen *<old>* gespeichert. Die WHILE-Schleife sorgt dafür, daß alle Elemente der Codefolge bearbeitet werden. Sobald ein Codeelement gelesen wurde, prüft der Algorithmus, ob der Index auf einen bereits belegten Tabelleneintrag zeigt. Falls dies der Fall ist, wird der String in den Ausgabestrom angehängt. Anschließend wird der String aus *Tabelle[old]* in einen Zeichenpuffer übernommen und das erste Zeichen des Eintrags von *Tabelle[code]* angehängt *([..]K)*. Der neue Teilstring *[..]K* wird in der Tabelle an der ersten freien Position eingetragen. Dann muß nur noch das zuletzt gelesene Zeichen von *<code>* nach *<old>* kopiert werden.

Ist der Tabellenplatz für *<code>* noch unbelegt, muß der zugehörige String generiert werden. Hierzu wird der bereits existierende String aus *Tabelle[old]* in den Puffer *[..]* kopiert. Dann wird das erste Zeichen des Puffers *[..]* in die Variable *K* kopiert und an den Puffer angehängt *[..]K.* Dieser neue String wird sowohl in die Tabelle eingetragen als auch in den Ausgabestrom kopiert. Schließlich wird noch der Inhalt von *<code>* in *<old>* gesichert. Der Vorgang wiederholt sich so lange, bis alle Eingangscodes gelesen sind. Anschließend findet sich die ursprüngliche Zeichenfolge wieder im Ausgabestrom. Die folgende kleine Abbildung gibt die einzelnen Schritte der Dekomprimierung wieder.

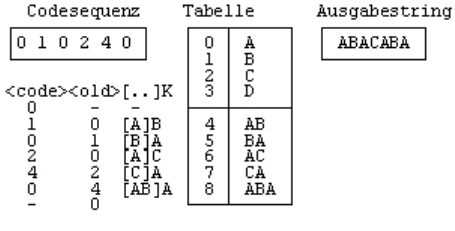

Abbildung 22.8 LZW-Dekodierung

Aus der Codefolge 010240 wurde wieder die ursprüngliche Zeichenfolge ABACABA generiert. Zu Beginn der Dekodierung wird die Stringtabelle mit den Zeichen A, B, C, D initialisiert. Dies waren die ursprünglichen Initialisierungszeichen der Komprimierungstabelle. Während der Bearbeitung der Eingabecodes wird die komplette Zeichentabelle wieder aufgebaut. Zum Abschluß liegt dann die komplette Tabelle mit allen Einträgen der Komprimierungstabelle vor (vergleichen Sie die beiden Beispiele).

Achtung: So elegant das LZW-Komprimierungsverfahren im GIF-Format funktioniert, gibt es einen gravierenden Nachteil, die Firma Unisys hält ein Patent auf dieses Verfahren. Dies führt dazu, dass Softwareentwickler Lizenzgebühren für Programme zahlen müssen, die das GIF-Format unterstützen. Selbst Webseitenanbieter werden mittlerweile von Unisys zur Kasse gebeten, da die GIF-Dateien häufig nicht mit lizenzierter Software erstellt wurden. Daher ist das PNG-Format wesentlich besser für Webgrafiken geeignet.

Modifiziertes LZW-Verfahren für GIF-Dateien

Bei der Komprimierung von GIF-Dateien gelangt das oben beschriebene Verfahren mit leichten Modifikationen zum Einsatz. Da es sich bei den Codes um Bitfolgen handelt, muß die Zahl der Bits pro Lesezugriff definiert werden. Es ist nicht sinnvoll, jeweils nur ein Bit (Pixel) zu lesen. Auch die Länge von 8 Bit pro Lesezugriff ist problematisch. Bei der GIF-Komprimierung wird vielmehr eine variable Codelänge verwendet, die zwischen 3 und 12 Bit lang sein darf. Im *Raster Data*-Block enthält das erste Byte die Größe Code Size. Dieser Wert entspricht zwar der Zahl der Bits pro Pixel. Der Wert wird aber als Startwert für die Codelänge interpretiert. Bei vier Bit pro Pixel wird in dem betreffenden Feld die Zahl N = 3 gespeichert. Dies bedeutet, daß bei jedem Zugriff auf die Eingangsdaten immer N + 1 = 4 Bit zu lesen sind. Während der weiteren Bearbeitung kann sich die Codelänge pro Zugriff auf 12 Bit erweitern. Sobald diese Länge erreicht wird, ist die Tabelle mit den Ausgabemustern voll, d.h., diese muß zurückgesetzt werden.

Der GIF-Komprimierer muß also eine Tabelle mit 4096 Einträgen anlegen. Zu Beginn wird diese Tabelle mit einigen Codes initialisiert. Die Größe des Initialisierungsbereiches wird dabei durch die Größe Code Size definiert. Bei 1 Bit pro Pixel muß N = 2 gesetzt werden. Dann werden die Einträge #0 und #1 der Tabelle initialisiert. Weiterhin werden an der Position 2**N und 2**N+1 zwei Spezialcodes abgelegt. Bei N = 2 sind dies die Positionen #4 und #5. Der erste Eintrag an der Position #4 wird als *clear code* <*CC*> bezeichnet. Wird dieser Eintrag bei der Dekomprimierung erkannt, muß die Tabelle neu initialisiert werden. Der Komprimierer wird diesen Code immer dann ausgeben, wenn die Tabelle voll ist. Weiterhin kann der Code als erstes Zeichen des Ausgangsstroms auftreten, um im Dekomprimierer einen Reset auszulösen. Der zweite Eintrag wird als *end of information* <*EOI*> bezeichnet. Er signalisiert dem Dekomprimierer, daß das Ende des Codestromes erreicht ist und keine weiteren Daten folgen.

Bei der Komprimierung/Dekomprimierung sollten neue Strings ab der Position <CC>+2 gespeichert werden. Der Code <CC> wird zu Beginn der Komprimierung und bei jedem Tabellenüberlauf in den Ausgabestrom geschrieben. Dann muß der Leser die Tabelle jeweils neu initialisieren. Die variable Codelänge der zu lesenden Daten sollte kein größeres Problem sein. Bei der Komprimierung wird mit der in *code size + 1* angegebenen Größe begonnen. Immer wenn ein Code aus der Tabellenposition *(2**(Codelänge) – 1)* ausgegeben wird, ist die Codelänge um 1 zu erhöhen. Dies wird so lange weitergeführt,

bis die Codelänge von 12 Bit erreicht ist. Dann muß die Tabelle neu initialisiert werden. Bei der Dekomprimierung beginnt man ebenfalls mit der in *code size + 1* vereinbarten Codelänge. Die Codelänge ist immer um 1 zu erhöhen, sobald der Tabelleneintrag *(2**(Codelänge) -1)* in die Ausgabe geschrieben wird. Beachten Sie auch, daß die Bits in der Codefolge mit den Bits der Stringcodes der Tabelle übereinstimmen.

Subblocks mit Raster-Daten

Da ein Bild in der Regel mehr als 255 Datenbytes umfaßt, werden die komprimierten Daten in einen *Raster Data*-Block und mehrere (Raster Data-)Subblocks unterteilt. Die Subblocks besitzen einen Aufbau ähnlich Tabelle 22.5, wobei das erste Byte (Code Size) fehlt. Ein Subblock beginnt mit einem Längenbyte, welches die Zahl der Folgebytes im Block definiert. Es können dann zwischen 0 und 255 Datenbytes folgen. Der Subblock kann damit maximal 256 Byte umfassen.

Terminator-Block

Das Ende des Bilddatenbereichs wird durch einen *Terminator*-Block (00H) angezeigt. Dies ist nichts anderes als ein Subblock, bei dem das Byte mit der Längenangabe auf 0 gesetzt wird. Dieser Block umfaßt dann nur das Längenbyte.

Die Blöcke *Image Descriptor*, *Local Color Map* und *Raster Data* lassen sich bei Bedarf mehrfach in einer Datei anlegen. Dies erlaubt es, mehrere Bilder in einer Datei zu speichern.

Der Graphic Control Extension-Block (GIF89a)

Dieser Block wurde in GIF89a neu definiert und enthält zusätzliche Parameter für ein Bild. Der Block ist optional und muß nach dem *Image Descriptor*-Block, aber vor den eigentlichen *Raster Data*-Blocks stehen. Die Daten gelten nur für das folgende (Teil-)Bild. Der Block besitzt den in Tabelle 22.6 gezeigten Aufbau.

Bytes	Bemerkungen
1	Extension-Block-Signatur (21H)
1	Graphic Control Label (F9H)
1	Block Size (4)
1	Flags
2	Delay Time
1	Transparent Color Index
1	Block-Terminator (00H)

Tabelle 22.6 Struktur eines Graphic Control Extension-Blocks (GIF89a)

Das erste Byte enthält die Signatur für den *Extension*-Block. Das Byte wird fest auf den Wert 21H (entspricht !) gesetzt.

Byte 2 enthält das *Graphic Control Label*. Dies ist nichts anderes als die Signatur für einen *Graphic Control Extension*-Block und wird fest auf den Wert F9H gesetzt. Das Feld *Block Size* gibt die Zahl der folgenden Datenbytes im Block an. Der Block-Terminator wird jedoch nicht eingerechnet. Das Feld enthält bei einem *Graphic Control Extension*-Block immer den Wert 4. Das folgende Flagbyte (Offset 03H) besitzt einen Aufbau gemäß Abbildung 22.9.

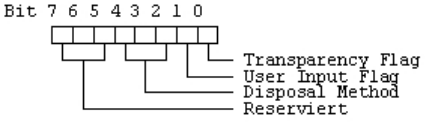

Abbildung 22.9 Kodierung des Flag im Graphics Control Extension-Block

Das *Transparency*-Flag signalisiert, ob das *Transparency Index*-Feld belegt ist. In diesem Fall wird das Flag auf 1 gesetzt. Ist das Flag auf 0 gesetzt, enthält das *Transparency Index*-Feld keinen gültigen Wert. (Dies wird in HTML genutzt, um transparente Bilder darzustellen.)

Das *User Input*-Flag definiert, ob auf eine Benutzereingabe nach der Ausgabe des Bildes gewartet wird. Ist das Flag auf 1 gesetzt, wird die Bearbeitung erst nach einer Benutzereingabe fortgesetzt. Die Art der Benutzereingabe (Return, Mausklick etc.) wird durch die Anwendung definiert. Ist eine Verzögerungszeit (Delay Time) definiert, wird die Bearbeitung nach dieser Wartezeit auch ohne Benutzereingabe fortgesetzt.

Die *Disposal*-Methode definiert, was mit der Grafik nach der Ausgabe passiert. Hierbei gibt es folgende Möglichkeiten:

- 0 No disposal specified
- 1 Do not dispose
- 2 Restore to background color
- 3 Restore to previous

Bei der Option 1 bleibt die Grafik erhalten. Beim Code 2 ist der Bereich der Grafik auf die Hintergrundfarbe zu setzen. Beim Code 3 ist die vorherige Einstellung zu restaurieren. Alle anderen Werte sind undefiniert.

Das Feld *Delay Time* ist als vorzeichenlose Zahl (unsigned Word) definiert. Ist das *User*-Flag gesetzt, definiert dieses Feld die Wartezeit in 1/100 Sekunden, ab der die Weiterbearbeitung der Datenfolge wieder aufgenommen wird. Diese Wartezeit kann durch eine Benutzereingabe abgebrochen werden. Die Wartezeit beginnt sofort nach der Ausgabe der Grafik.

Im Feld *Transparency Index* findet sich ein Bytewert. Tritt dieser Bytewert im Datenstrom für die Grafik auf, ist der betreffende Bildpunkt nicht auszugeben. Der Decoder kann zum

nächsten Bildpunkt weitergehen. Hierdurch bleibt der Bildschirmhintergrund erhalten. Dieser Index ist nur dann gültig, falls das *Transparency*-Flag gesetzt ist.

Das letzte Byte dient als Block-Terminator und besitzt den Wert 00H. Dieses Byte wird in der Längenangabe des *Extension*-Blocks nicht berücksichtigt. Im Grunde genommen handelt es sich um einen leeren Block, der generell das Ende mehrerer Subblocks definiert.

Anmerkung: Dieser Block wird für animierte GIF-Dateien benutzt, die von Browsern wie Internet Explorer oder Netscape Navigator unterstützt werden.

Der Comment Extension-Block (GIF89a)

Dieser optionale Block ist ebenfalls erst ab GIF89a definiert und kann Kommentare zur GIF-Datei (z. B. Autor, Credits etc.) enthalten. Der Block darf an jeder Stelle in der GIF-Datei auftreten, an der ein Block beginnen kann. Er sollte jedoch nicht zwischen Subblocks eingefügt werden. Empfohlen wird, den Block zu Beginn oder am Ende der GIF-Datei einzufügen. Der Inhalt des Blocks besitzt keinen Einfluß auf die GIF-Bilder. Tabelle 22.7 gibt den Aufbau des Blocks wieder.

Bytes	Bemerkungen
1	Extension-Block-Signatur (21H)
1	Comment Label (FEH)
1	Block Size
n	Kommentar als Subblocks mit 1 Länge Subblock n Datenbereich Subblock
1	Block-Terminator (00H)

Tabelle 22.7 Struktur eines Comment Extension-Blocks (GIF89a)

Das erste Byte enthält die Signatur für den *Extension*-Block. Das Byte wird fest auf den Wert 21H (entspricht !) gesetzt.

Das zweite Byte enthält die Signatur für den *Comment Extension*-Block und wird immer mit dem Code FEH belegt.

An diese Signatur schließt sich eine Sequenz von Subblocks mit dem eigentlichen Kommentartext an. Jeder Subblock enthält im ersten Byte die Zahl der folgenden Datenbytes. An dieses Byte schließen sich zwischen 0 und 255 Datenbytes an. Ist der Kommentarstring länger als ein Block, wird er auf mehrere Subblocks aufgeteilt. Das Ende des *Comment Extension*-Blocks wird durch einen Terminator markiert. Hierbei handelt es sich um einen Subblock mit einem Byte Länge, der den Wert 00H enthält.

Der Kommentarstring sollte mit 7-Bit-ASCII-Zeichen erstellt werden. Eine Darstellung multilingualer Zeichen (ä, ö, ü etc.) ist nicht vorgesehen. Der Inhalt des Blocks sollte nicht zur Speicherung von Dekodierinformationen benutzt werden, d.h., der Decoder kann den Block übergehen.

Der Plain Text Extension-Block (GIF89a)

Dieser optionale Block ist ebenfalls erst ab GIF89a definiert und kann Texte und Parameter zur grafischen Darstellung dieser Texte enthalten. Tabelle 22.8 gibt den Aufbau des Blocks wieder.

Bytes	Bemerkungen
1	Extension-Block-Signatur (21H)
1	Plain Text (01H)
1	Block Size
2	Text Grid Left Position
2	Text Grid Top Position
2	Text Grid Width
2	Text Grid Height
1	Character Cell Width
1	Character Cell Height
1	Text Foreground Color
1	Text Background Color
n	Sequenz von Text-Subblocks mit 1 Byte Länge Subblock n Byte Subblock mit Text
1	Block-Terminator (00H)

Tabelle 22.8 Struktur eines Plain Text Extension-Blocks (GIF89a)

Das erste Byte enthält die Signatur für den *Extension*-Block. Das Byte wird fest auf den Wert 21H (entspricht !) gesetzt. Das zweite Byte enthält die Signatur für den *Plain Text Extension*-Block und wird immer mit dem Code 01H belegt. Das dritte Byte enthält die Längenangabe für den folgenden Datenbereich. Beim *Plain Text Extension*-Block wird dieser Wert fest auf 12 (0CH) gesetzt. Der Text wird dann in Subblocks mit eigenen Längenangaben gespeichert.

Ab Offset 03H vom Blockanfang beginnt eine Sequenz von Feldern, die als unsigned Word interpretiert werden. Das erste Feld definiert den linken Rand (column number) für das Gitter zur Textausgabe. Die Position wird dabei in Pixel vom linken Fensterrand definiert. Das folgende Feld legt die obere Gitterposition (row number) für die Textausgabe in Pixel fest und bezieht sich ebenfalls auf den logischen Bildschirm.

Das Feld *Character Cell Width* definiert die Breite einer Gitterzelle in Pixel. Diese Gitterzelle dient zur Aufnahme eines Zeichens. In *Character Cell Height* wird die Höhe einer Gitterzelle definiert. Auch dieser Wert wird als unsigned Word angegeben. Der Decoder muß diese Werte auf die Abmessungen des virtuellen Bildschirms umrechnen, wobei das Ergebnis nur ganzzahlig sein darf.

Die letzten beiden Felder belegen jeweils nur ein Byte und geben den Farbwert (Index in die Farbpalette) für die Vordergrund- und die Hintergrundfarbe des Textes an. Der eigentliche Text wird in einer Sequenz von Subblocks gespeichert. Diese Subblocks schließen sich direkt an die obige Struktur an. Jeder Subblock enthält im ersten Byte die Zahl der folgenden Datenbytes. An dieses Byte schließen sich zwischen 0 und 255 Datenbytes mit dem auszugebenden Text an. Ist der Text länger als ein Block, wird er auf mehrere Subblocks aufgeteilt. Das Ende des *Plain Text Extension*-Blocks wird durch einen Terminator markiert. Hierbei handelt es sich um einen Subblock mit einem Byte Länge, der den Wert 00H enthält.

Zur Ausgabe des Textes wird ein Gittermuster (grid of character cells) definiert, wobei jede Gitterzelle ein einzelnes Zeichen aufnimmt. Die Parameter für das Gittermuster finden sich im *Plain Text Extension*-Block. Der Decoder muß die Parameter so umsetzen, daß die Gitterabmessungen Ganzzahlen (Integer) ergeben. Ein auftretender Nachkommaanteil ist abzuschneiden. Aus Kompatibilitätsgründen sollten die Zellabmessungen mit 8 x 8 oder 8 x 16 Punkten (Breite x Höhe) gewählt werden.

Die einzelnen Zeichen sind sequentiell zu lesen und beginnend in der oberen linken Ecke des Gitters zeilenweise in die einzelnen Zellen einzutragen. Zur Anzeige ist der bestmögliche Monospace-Font mit der passenden Größe vom Decoder zu wählen. Der auszugebende Text muß mit 7-Bit-ASCII-Zeichen kodiert werden. Eine Darstellung multilingualer Zeichen (ä, ö, ü etc.) ist nicht vorgesehen. Treten Zeichencodes unterhalb 20H und oberhalb von 7FH auf, muß der Decoder ein Leerzeichen (Space, 20H) ausgeben.

Der Application Extension-Block (GIF89a)

Dieser optionale Block wurde erst in GIF89a definiert. Der Block dient zur Aufnahme anwendungsspezifischer Informationen. Tabelle 22.9 gibt den Aufbau des Blocks wieder.

Bytes	Bemerkungen
1	Extension-Block-Signatur (21H)
1	Application Extension (FFH)
1	Block Size 11
8	Application Identifier
3	Application Authentication Code

Bytes	Bemerkungen
n	Sequenz von Subblocks mit 1 Byte Länge Subblock n Byte Daten Subblock
1	Block-Terminator (00H)

Tabelle 22.9 Struktur eines Application Extension-Blocks (GIF89a)

Das erste Byte enthält die Signatur für den *Extension*-Block. Das Byte wird fest auf den Wert 21H (entspricht !) gesetzt. Das zweite Byte enthält die Signatur für den *Application Extension*-Block und wird immer mit dem Code FFH belegt.

Das dritte Byte enthält die Längenangabe für den folgenden Datenbereich. Beim *Application Extension*-Block wird dieser Wert fest auf 11 (0BH) gesetzt. Die eigentlichen Parameter werden dann in Subblocks mit eigenen Längenangaben gespeichert. Ab Offset 03H folgen 8 Byte mit dem *Application Identifier*. Dieser Identifier muß aus druckbaren ASCII-Zeichen bestehen und dient zur Bezeichnung der Anwendung, die die Daten erzeugt hat.

Anschließend folgen drei Byte für den *Application Authentication Code*. Hier kann ein Binärcode gespeichert werden, der durch die Anwendung berechnet wird. Damit ist eine eindeutige Identifizierung der erzeugenden Anwendung möglich.

An dieses Feld schließen sich die Subblocks mit den anwendungsspezifischen Daten an. Jeder Subblock beginnt mit einem Längenbyte, gefolgt von bis zu 255 Datenbytes. Der letzte Block enthält nur das Längenbyte mit dem Wert 00H und dient als Terminator.

Der GIF-Terminator

Der Abschluß einer GIF-Datei wird durch einen *Terminator*-Block markiert. Hierbei handelt es sich wieder um einen 1-Byte-Block. Dieses Byte enthält ein Semikolon (Code 3BH) als Terminator.

48 PostScript und Encapsulated PostScript (EPS)

Mit PostScript hat der Hersteller Adobe eine druckerunabhängige Beschreibungssprache geschaffen, die sich als Standard durchsetzt. Viele Softwareprodukte unterstützen deshalb den Import von PostScript-kodierten Bildern und Texten. Um einen Austausch zwischen verschiedenen Systemen zu erleichtern, verlangt Adobe bestimmte Strukturkonventionen (EPS), die von PostScript-Programmen einzuhalten sind. Abbildung 48.1 enthält eine solche Datei, die mit dem Lotus-Produkt *Freelance* erstellt wurde.

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title: born1.EPS
%%Creator: Freelance Plus 3.01
%%CreationDate: 8/1/1990
%%Pages: 1
%%DocumentFonts: (atend)
%%BoundingBox: 0 0 648 468
%%EndComments
%%BeginProcSet: Freelance Plus
/Freelance_Plus dup 100 dict def load begin
[ ] {bind} stopped { (patching the bind operator...) = flush
/*bind /bind load def /bind { dup xcheck { *bind } if } *bind def }
if pop /bdf {bind def} bind def /ldf {load def} bdf
/mt /moveto ldf /rt /rmoveto ldf /l2 /lineto ldf /c2 /curveto ldf
/sg /setgray ldf /gs /gsave ldf /ef /eofill ldf /rl2 /rlineto ldf
/st /stroke ldf /gr /grestore ldf /np /newpath ldf
/sv /save ldf /su /statusdict ldf /rs /restore ldf
/sw /setlinewidth ldf /sd /setdash ldf /cp /closepath ldf /ed
{exch def } bdf /cfnt {findfont exch makefont setfont} bdf
/itr {transform round exch round exch itransform} bdf
/fres 72 0 matrix currentmatrix dtransform
exch dup mul exch dup mul add sqrt def
/res fres def /mcm matrix currentmatrix bdf
%%EndProcSet
end
%%EndProlog
%%BeginSetup
Freelance_Plus begin
```

```
save newpath
.1 .1 scale
/ecm matrix currentmatrix bdf
/sem {ecm setmatrix} bdf
-720 -720 translate
0 setlinecap
0 setlinejoin
1.42 setmiterlimit%%EndSetup
/Ich 256 array def StandardEncoding Ich copy pop /bullet
/paragraph/section/dieresis/tilde/ring
/circumflex/grave/acute/quotedblleft/quotesingle
/ellipsis/endash/emdash/guilsinglleft/guilsinglright
/quotedblbase/quotesinglbase/quotedblright/OE/oe
/Ydieresis/fi/fl/dagger/daggerdbl/Ccedilla/udieresis
/eacute/acircumflex/adieresis/agrave/aring/ccedilla
/ecircumflex/edieresis/egrave/idieresis/icircumflex
/igrave/Adieresis/Aring/Eacute/ae/AE/ocircumflex
/odieresis/ograve/ucircumflex/ugrave/ydieresis
/Odieresis/Udieresis/oslash/sterling/Oslash/florin
/aacute/iacute/oacute/uacute/ntilde/Ntilde/ordfeminine
/ordmasculine/questiondown/exclamdown/guillemotleft
/guillemotright/Aacute/Acircumflex/Agrave/cent/yen
/atilde/Atilde/currency/Ecircumflex/Edieresis/Egrave
/dotlessi/Iacute/Icircumflex/Idieresis/Igrave/Oacute
/germandbls/Ocircumflex/Ograve/otilde/Otilde/Uacute
/Ucircumflex/Ugrave/macron/cedilla/periodcentered
Ich 127 97 getinterval astore pop
/Ienc { /ncs Ich def /nfn ed /bfn ed /bfd bfn findfont def
/nf bfd maxlength dict def bfd{exch dup dup /FID ne exch
/Encoding ne and {exch nf 3 1 roll put}{pop pop} ifelse }
forall nf/FontName nfn put nf/Encoding ncs put nfn nf
definefont pop}bdf
/IencO { /ncs Ich def /nfn ed /bfn ed /lnw ed /bfd bfn
findfont def /nf bfd maxlength 4 add dict def bfd{exch dup
dup /FID ne exch /Encoding ne and
{exch nf 3 1 roll put}{pop pop} ifelse }forall
```

```
nf/FontName nfn put nf/Encoding ncs put
nf/PaintType 2 put nf/StrokeWidth lnw put
nfn nf definefont pop}bdf
/IencSO { /nfn ed /bfn ed /lnw ed /bfd bfn findfont def
/nf bfd maxlength 4 add dict def bfd{exch dup
/FID ne { exch nf 3 1 roll put}{pop pop} ifelse }forall
nf/FontName nfn put nf/PaintType 2 put
nf/StrokeWidth lnw put nfn nf definefont pop}bdf
/Helvetica /Flfon1 Ienc
[191 0 0.00 191 -18 -142]  /Flfon1 cfnt
0.000 sg
2737 3026 itr mt
(Dies ist ein Text)
show
6 sw
sv np [] 0 sd
2764 3749 itr mt
4071 3749 itr l2
st rs
sv np
1352 2572 itr mt
2424 2572 itr l2
2424 3513 itr l2
1352 3513 itr l2
1352 2572 itr l2
cp
st rs
sv np
3456 4438 itr mt
currentpoint translate
np 1.000 1.000 scale
0 0 549 0.000 360.000 arc sem
st rs
sv np
5156 4438 itr mt
5823 4573 itr l2
```

```
5823 3984 itr l2
4973 3984 itr l2
4973 4354 itr l2
5156 4438 itr l2
gs 0.000 sg
ef gr
cp
st rs
rs end
%%Trailer
%%DocumentFonts: Helvetica
```

Abbildung 48.1 EPS-Datei, erzeugt mit Freelance Plus

Die EPS-Datei besteht aus den eigentlichen PostScript-Anweisungen und den Strukturkonventionen zur Programmsteuerung.

Die EPS-Strukturkonventionen

Die Strukturkonventionen von EPS sehen für jedes Programm einen *Prolog* und ein *Script* vor. Im Prolog finden sich anwendungsabhängige Definitionen, während im Script die eigentlichen PostScript-Anweisungen stehen. Die Teile werden durch verschiedene Strukturinformationen beschrieben, die als PostScript-Kommentare mit dem Zeichen % eingeleitet werden. Der Interpreter ignoriert dann die komplette Zeile, während ein lesendes Programm die darin enthaltenen Informationen auswerten kann.

Um die Strukturinformationen von anderen Kommentarzeilen zu unterscheiden, beginnen die Infos mit den Zeichen:

```
%!
%%
```

gefolgt von einem beliebigen ASCII-String. Eine EPS-Datei beginnt immer mit der folgenden Zeile, die eine Datei als PostScript-Datei gemäß den Adobe-Konventionen auszeichnet:

```
%!PS-Adobe-2.0 EPSF-1.2
```

Weiterhin enthält die Zeile noch das Schlüsselwort:

```
PS-Adobe-
```

gefolgt von einer Versionsnummer.

Header Comments

Unmittelbar nach der Versionszeile finden sich weitere Kommentare mit zusätzlichen Informationen.

```
%%DocumentFonts: font1, font2 ...
```

Mit diesem Kommentar lassen sich die benutzten Fontnamen angeben. Diese Anweisung kann aber auch an das Ende der Datei verschoben werden. Dann muß im Header folgende Zeile erscheinen (Abbildung 48.1):

```
%%DocumentFonts: (atend)
```

Diese Anweisung legt den Dokumententitel fest:

```
%%Title: title
```

Hier wird häufig der Dateiname mitgespeichert. Das dateierzeugende Programm oder die zuständige Person sowie das Erzeugungsdatum sind in diesen Zeilen notiert:

```
%Creator: text
%%CreationDate: datum
```

Im Datum kann auch die Zeit als String mit aufgenommen werden. Diese Anweisung benennt einen Empfänger für das Dokument:

```
%%For: text
```

Die Zahl der Seiten wird mit der nächsten Sequenz gesetzt:

```
%%Pages: xx
```

Der Wert *xx* sollte eine nicht negative Dezimalzahl umfassen und die Zahl der auf dem Drucker auszugebenden Seiten spezifizieren. Erzeugt das Programm keine Seiten, ist eine 0 einzutragen. Die Spezifikation *attend* ist ebenfalls erlaubt.

Die Bounding-Box umschließt alle Markierungen des Programmes mit einem Rechteck. Die nächste Anweisung enthält als Parameter vier Integerwerte, die die linke obere und die rechte untere Ecke der Bounding-Box im Initialkoordinatensystem angeben:

```
%%BoundingBox: 0 0 648 468
```

Folgende Anweisung schließt den Header ab:

```
%%EndComments
```

Body Comments

Die auftretenden Kommentare dienen im wesentlichen zur Markierung verschiedener Abschnitte im eigentlichen Programmteil. Nachfolgend findet sich eine Zusammenstellung möglicher Kommentare (aus Abbildung 48.1):

```
%%BeginProcSet:
%%EndProcSet
%%EndProlog
%%BeginSetup
%%EndSetup
```

Der Prolog wird durch einen entsprechenden Kommentar (*EndProlog*) abgeschlossen. Der Beginn der Beschreibung einer Seite läßt sich durch diese Sequenz markieren:

```
%%Page: label ordial
```

In *label* und *ordial* werden die entsprechenden Nummern eingetragen. Zeichensätze für einzelne Seiten, die durch ein Leseprogramm ausgewertet werden können, lassen sich so spezifizieren:

```
%%PageFonts: font1, font2 ...
```

Trailer Comments

Den Abschluß einer EPS-Datei bildet der Anhang (*Trailer*), eingeleitet durch den Kommentar

```
%%Trailer
```

Falls im Kopf die Anweisung mit *attend* an das Ende verschoben wurde, können mit der folgenden Anweisung noch die im Dokument verwendeten Zeichensätze aufgeführt werden:

```
%%DocumentFonts: font1, font2 ...
```

Entsprechende Beispiele finden sich in Abbildung 48.1.

PostScript-Anweisungen (Level 1)

Die Sprache PostScript arbeitet nach dem Prinzip der umgekehrten polnischen Notation und besitzt verschiedene Anweisungen zur Ausgabe von Schrift und Grafik. Für den Umgang mit der Sprache seien die von Adobe bei Addison Wesley publizierten Titel (z. B. PostScript Language Reference) als Lektüre empfohlen. Nachfolgend werden die wichtigsten PostScript-Anweisungen kurz beschrieben.

abs

Format: `num abs`

Wandelt den Inhalt von *num* in eine absolute Zahl um und legt das Ergebnis auf dem Stack ab.

add

Format: `num1 num2 add`

Addiert die beiden Parameter *num1* und *num2* und legt das Ergebnis auf dem Stack ab.

and

Format: `bool1 bool2 and`

Verknüpft die beiden logischen Werte *bool1* und *bool2* über die And-Funktion und legt das Ergebnis auf dem Stack ab.

arc

Format: `x y r ang1 ang2 arc`

Zeichnet am Punkt *x,y* einen Kreisbogen mit dem Radius *r*. Die Parameter *ang1* und *ang2* geben die Winkel des Kreisbogens an.

arcn

Format: `x y r ang1 ang2 arcn`

Zeichnet am Punkt *x,y* im Uhrzeigersinn einen Kreisbogen mit dem Radius *r*. Die Parameter *ang1* und *ang2* geben die Winkel des Kreisbogens an.

arcto

Format: `x1 y1 x2 y2 arcto`

Zeichnet vom aktuellen Punkt *x0,y0* eine Gerade zum Punkt *x1,y1* und von dort einen Kreisbogen zum Punkt *x2,y2*. Nach dem Aufruf stehen auf dem Stack vier Parameter mit den Tangentenpunkten.

ashow

Format: `ax ay string ashow`

Gibt einen Text (*string*) aus, wobei die Dickten mit *ax* und *ay* korrigiert werden.

atan

Format: `num den atan0`

Berechnet aus dem Parameter *num* den Winkel in Grad und legt das Ergebnis auf dem Stack ab.

awidthshow

Format: `cx cy char ax ay string awidthshow`

Gibt einen Text (*string*) aus, wobei die Dickten mit *ax* und *ay* korrigiert werden. Zusätzlich wird jedes Zeichen *char* im Text mit den Parametern *cx* und *cy* korrigiert.

bind

Format: `proc bind`

Ersetzt den Operatornamen einer Prozedur durch deren Werte und legt sie auf dem Stack ab.

bitshift

Format: `int1 shift bitshift`

Verschiebt den Wert von *int1* um *shift* nach links. Ist der Wert negativ, werden die Bits nach rechts verschoben. Das Ergebnis findet sich anschließend auf dem Stack.

ceiling

Format: `num ceiling`

Liefert den nächsten benachbarten Wert von *num* auf dem Stack zurück.

charpath

Format: `string bool charpath`

Liefert den Pfad, der durch die Anwendung von *show* erzeugt wurde.

clip

Format: `clip`

Schneidet die inneren Flächen eines Clipping-Pfades aus.

clippath

Format: `clippath`

Definiert den aktuellen Clipping-Pfad.

closepath

Format: `closepath`

Schließt den aktuellen Subpfad durch ein gerades Liniensegment vom aktuellen Punkt zum Startpunkt.

copypage

Format: `copypage`

Gibt den Inhalt der aktuellen Seite auf dem Drucker aus, ohne die Seite neu zu initialisieren. Dadurch kann ein Teilbild probeweise ausgegeben werden.

cos

Format: `angle cos`

Berechnet den Cosinuswert des Winkels und gibt das Ergebnis auf dem Stack zurück.

currentpoint

Format: `currentpoint`

Liefert die aktuellen XY-Koordinaten auf dem Stack zurück.

curveto

Format: `x1 y1 x2 y2 x3 y3 curveto`

Zeichnet eine Bézierkurve durch die drei angegebenen Punkte.

cvi

Format: `num cvi`

Konvertiert den Wert von *num* in eine Integergröße und gibt diese auf dem Stack zurück.

cvn

Format: `string cvn`

Konvertiert eine Zahl in Form eines Strings in eine Integergröße und gibt diese auf dem Stack zurück.

cvr

Format: `string cvr`
`num cvr`

Konvertiert den Wert von *num* oder *string* in eine Fließkommagröße und gibt diese auf dem Stack zurück.

cvrs

Format: `num string radix cvrs`

Konvertiert eine Zahl *num* in einen String mit der Basis *radix* und gibt das Ergebnis auf dem Stack zurück.

cvs

Format: `num string cvs`

Konvertiert den Wert eines beliebigen Elements in einen Text und gibt diesen auf dem Stack zurück.

div

Format: `num1 num2 div`

Dividiert die Zahl *num1* durch *num2* und legt das Ergebnis auf dem Stack ab.

dup

Format: `element dup`

Dupliziert das oberste Element des Stacks.

eoclip

Format: `eoclip`

Schneidet die inneren Flächen des aktuellen Clipping-Pfades mit den inneren Flächen des aktuellen Pfades.

eofill

Format: `eofill`

Füllt die inneren Flächen des aktuellen Pfades mit der aktuellen Farbe aus.

eq

Format: `elem1 elem2 eq`

Vergleicht zwei beliebige Elemente miteinander und legt als Ergebnis *True* oder *False* auf dem Stack ab.

exch

Format: `elem1 elem2 exch`

Vertauscht die beiden obersten Stackelemente.

exec

Format: `oper exec`

Legt den Operanden auf den Stack und führt ihn direkt aus.

exit

Format: `exit`

Beendet die Ausführung einer Schleife (*for*, *loop*, *repeat*, *forall*).

exp

Format: `base exp exp`

Berechnet aus der Basis *base* und dem Exponenten *exp* einen Fließkommawert.

fill

Format: `fill`

Füllt die Fläche des aktuellen Pfades mit der aktuellen Farbe.

findfont

Format: `font findfont`

Sucht einen durch *font* beschriebenen Namen aus dem Font-Dictionary.

for

Format: `start step end { proc } for`

Konstruiert eine Schleife, die mit dem Wert von *start* beginnt. Der Schleifenindex wird mit der Schrittweite *step* so lange erhöht, bis der Endwert *end* erreicht ist. Bei jedem Durchlauf werden die in den geschweiften Klammern ({}) stehenden Anweisungen ausgeführt.

ge

Format: `num1 num2 ge`

Führt zwischen *num1* und *num2* einen Vergleich auf *greater equal* aus und legt das Ergebnis als Boole-Wert auf dem Stack ab.

grestore

Format: `grestore`

Restauriert den mit *gsave* gesicherten Grafikstatus.

gsave

Format: `gsave`

Sichert den Grafikstatus.

gt

Format: `num1 num2 gt`

Führt zwischen *num1* und *num2* einen Vergleich auf *greater* aus und legt das Ergebnis als Boole-Wert auf dem Stack ab.

idiv

Format: `int1 int2 idiv`

Führt eine Integerdivision zwischen *int1/int2* aus und legt das Ergebnis auf dem Stack ab.

if

Format: `bool { proc } if`

Bearbeitet die in den Klammern stehenden Anweisungen, falls *bool* gleich *True* ist.

ifelse

Format: `bool { proc1 } { proc2 } ifelse`

Bearbeitet die in den Klammern stehenden Anweisungen *proc1*, falls:

`bool = True`

andernfalls werden die Anweisungen *proc2* ausgeführt.

image

Format: `width height bits_per_sample matrix proc image`

Überträgt ein Rasterbild auf die aktuelle Seite. Die Parameter *width* und *height* spezifizieren die Abmessungen des Bildes; mit *proc* werden die Bilddaten eingelesen.

imagemask

Format: `width height invert matrix proc imagemask`

Überträgt ein Rasterbild auf die aktuelle Seite. Die Parameter *width* und *height* spezifizieren die Abmessungen des Bildes, mit *proc* werden die Bilddaten eingelesen. Der logische Wert *invert* definiert dabei, ob die Bits direkt oder invertiert (*0 Bits zeichnen -> invert = False*) gezeichnet werden.

index

Format: `a1 a2...ax n index`

Kopiert das *n-te* Stackelement oben auf den Stack, gezählt vom obersten Element (0).

le

Format: `num1 num2 le`

Führt zwischen *num1* und *num2* einen Vergleich auf *less or equal* aus und legt das Ergebnis als Booleschen Wert auf dem Stack ab.

lineto

Format: `x y lineto`

Fügt an den aktuellen Pfad ein gerades Liniensegment von der aktuellen Position zum Punkt *x,y* an.

ln

Format: `num1 ln`

Liefert den Fließkommawert von *ln(num1)* auf den Stack zurück.

log

Format: `num1 log`

Liefert den Fließkommawert von *log(num1)* auf den Stack zurück.

loop

Format: `{ proc } loop`

Führt die Anweisungen der Schleife *{ proc }* so lange aus, bis ein *exit*-Operator geladen wird.

lt

Format: `num1 num2 lt`

Führt einen Vergleich auf *less then num1* und *num2* aus und legt das Ergebnis als Boole-Wert auf dem Stack ab.

mod

Format: `int1 int2 mod`

Liefert den Rest der Division von *int1/int2* auf dem Stack zurück (Modulofunktion).

moveto

Format: `x y moveto`

Beginnt einen neuen Pfad und setzt den aktuellen Punkt auf die mit *x,y* angegebenen Koordinaten.

mul

Format: `num1 num2 mul`

Liefert das Ergebnis der Multiplikation von *num1* und *num2* auf dem Stack ab.

neg

Format: `num1 neg`

Invertiert den Wert von *num1* und legt das Ergebnis auf dem Stack ab.

newpath

Format: `newpath`

Legt einen neuen Pfad auf der aktuellen Seite an; der aktuelle Punkt ist dann undefiniert.

not

Format: `bool1 not`

Invertiert den Wert der logischen Variablen und gibt das Ergebnis auf dem Stack zurück.

or

Format: `bool1 bool2 or`

Verknüpft die beiden Operatoren mit *or* und liefert das Ergebnis auf dem Stack ab.

pop

Format: `pop`

Entfernt das oberste Element vom Stack.

quit

Format: `quit`

Beendet die Arbeit des Interpreters.

rand

Format: `rand`

Liefert eine Zufallszahl (Integer) auf dem Stack zurück.

rcurveto

Format: `dx1, dy1 dx2 dy2 dx3 dy3 rcurveto`

Zeichnet eine Bézierkurve vom aktuellen Pfad relativ zu den Koordinatenangaben *dx,dy*.

repeat

Format: `count { proc } repeat`

Führt die Anweisungen in *proc count*-mal aus.

rlinto

Format: `dx dy rlineto`

Erzeugt einen neuen Subpfad durch relative Verschiebung um die Strecke *dx,dy*.

roll

Format: `elem1...elemx n j roll`

Verschiebt *n* Elemente des Stacks um *j* Positionen im Kreis.

round

Format: `num1 round`

Rundet den Wert *num1* auf den nächstliegenden Integerwert.

scale

Format: `x y scale`

Verändert die Skalierung der x- und y-Achsen.

scalefont

Format: `font scale scalefont`

Skaliert die Größe eines Zeichensatzes vor der Ausgabe.

search

Format: `string such search`

Sucht *string* nach der Zeichenkette *such* ab und legt das Ergebnis als:

`Reststring, such, Anfangsteil, bool`

auf dem Stack ab. Der oberste Wert auf dem Stack gibt an, ob der String gefunden wurde (*bool* = *True*).

setfont

Format: `font setfont`

Aktiviert den angegebenen Zeichensatz als aktuellen Font.

setgray

Format: `num setgray`

Legt die Schwärzung für die Zeichenoperationen fest. Der Wert von *num* kann zwischen 0 (Weiß) und 1 (Schwarz) liegen.

setlinewidth

Format: `num setlinewidth`

Setzt die Breite von Linien auf einen definierten Wert.

show

Format: `(text) show`

Gibt den auf dem Stack stehenden Text im aktuellen Pfad aus.

showpage

Format: `showpage`

Kopiert die aktuelle Seite auf das Ausgabegerät.

sin

Format: `angle sin`

Berechnet den Sinus des angegebenen Winkels und gibt das Ergebnis als Fließkommawert auf dem Stack zurück.

sqrt

Format: `num sqrt`

Berechnet die Wurzel aus *num* und gibt das Ergebnis als Fließkommawert auf dem Stack zurück.

string

Format: `int string`

Erzeugt ein Stringobjekt der Länge *int*.

stringwidth

Format: `string stringwidth`

Ermittelt die Länge des angegebenen Strings.

stroke

Format: `stroke`

Zieht entlang des aktuellen Pfades eine Linie. Erst damit werden Objekte in der aktuellen Farbe gezeichnet.

sub

Format: `num1 num2 sub`

Subtrahiert *num1-num2* und liefert das Ergebnis auf dem Stack ab.

truncate

Format: `num truncate`

Entfernt den Nachkommateil von *num* und liefert das Ergebnis auf dem Stack zurück.

xor

Format: `bool1 bool2 xor`

Verknüpft die beiden Werte *bool1* und *bool2* mit Xor und legt das Ergebnis auf dem Stack ab.

PostScript Level 2

PostScript Level 2 stellt eine Erweiterung des Befehlsvorrats in Level 1 dar. Neben verbesserten Möglichkeiten zur Komprimierung von PostScript-Programmen wurden neuere Verfahren implementiert:

- Verbesserungen in der Speicherverwaltung und Definition eines Dictionary-Operands.
- Einführung von Resources (Sammlung von benannten Objekten) und Filtern zum Import/Export von Dateien.
- Binäre Kodierung der Programme zur optimalen Übertragung der Programme; User Paths, um die Pfade für Operatoren zu definieren.
- Befehle, um Formulare (FORMS) zu erzeugen und wiederzuverwenden; Definition von Mustern (Pattern).
- Definition verschiedener Farbräume, um Farbausgaben unter PostScript zu unterstützen; Unterstützung des CIE-Farbmodells und Befehle zur Ausgabe von Bildern.
- Erweiterungen in den Text- und Grafikoperatoren; Parameter für den Interpreter und Device-Setup-Befehle.

Nachfolgend finden Sie eine kurze Übersicht über die neuen Befehle in Level 2.

<< mark

Der Befehl legt das Objekt *mark* auf dem Operandenstack ab; dieser Befehl verhält sich wie der *mark*- oder *[*-Operator.

>> mark

Format: `mark key1 value1 .... keyn valuen >> dict`

Erzeugt ein Dictionary mit den spezifizierten *key*-Werten. Als Operanden werden *mark* und eine gerade Anzahl von Operanden (key) erwartet. Diese Schlüssel werden in das Dictionary eingefügt. Der folgende Befehl:

```
<< /Duplex true /PageSize [612 792] /Collate false >> setpagedevice
```

erzeugt ein Dictionary mit drei Einträgen, welches sofort zum *setpagedevice*-Operator weitergegeben wird.

arct

Format: `x1 y1 x2 y2 r arct`

Der Befehl fügt ein Kreissegment an den aktuellen Pfad an. Vorher kann eine gerade Linie auftreten, damit ein Kreissegment erzeugt wird.

Die Punkte x1 y1 und x2 y2 definieren die Ecken, in die der Kreis einzufügen ist. Vom Startpunkt x0 y0 wird ein Pfad zu x1 y1 und dann zu x2 y2 konstruiert. Anschließend erzeugt der Befehl ein Kreissegment mit dem Radius *r* an den Tangenten. Letztlich erzeugt

der Befehl eine abgerundete Ecke in einem Viereck. Das folgende Beispiel verdeutlicht die Anwendung des Befehls:

```
newpath
0 0 moveto          % auf Startpunkt
0 10 10 10 4 arct   % runde Ecke zeichnen
10 10 lineto        % Linie zum Endpunkt zeichnen
```

Der Pfad wird dabei vom Startpunkt bis zum angegebenen Endpunkt konstruiert.

cshow

Format: `proc string cshow`

Ruft die Prozedur *proc* für jede Operation im Font-Mapping-Algorithmus einmal auf. Beim Aufruf von *proc* enthält der Stack den selektierten Zeichencode und die x-, y-Koordinaten des Vektors für die Ausgabe. *cshow* gibt keine Zeichen aus und verändert auch nicht die aktuelle Zeichenposition. Die Ausgabe des Textes erfolgt über *proc*, d.h., der Befehl läßt sich zur Ausgabe und Positionierung spezieller Zeichen verwenden.

currentblackgeneration

Format: `currentblackgeneration proc`

Gibt die aktuelle *blackgeneration*-Funktion für den Grafikstatus zurück.

currentcacheparams

Format: `currentcacheparams mark size lower upper`

Legt ein *mark*-Objekt, gefolgt von den aktuellen Cacheparametern, auf dem Operandenstack ab. Die Parameterzahl ist dabei variabel.

currentcmykcolor

Format: `currentcmykcolor cyan magenta yellow black`

Gibt die aktuelle Farbe des Grafikstatus im CMYK-Farbsystem auf dem Stack zurück.

Ist das Farbsystem auf *DeviceCMYK* gesetzt, gibt der Befehl die Farben zurück, die mit *setcmykcolor* oder *setcolor* gesetzt wurden. Bei *DeviceRGB* oder *DeviceGray* werden die Farbwerte erst in den CMYK-Farbraum konvertiert.

currentcolor

Format: `currentcolor comp1 comp2 .... compn`

Der Befehl ermittelt die aktuellen Farbkomponenten und gibt diese auf dem Stack zurück.

currentcolorrendering

Format: `currentcolorrendering dict`

Der Befehl gibt den Wert des CIE-Color-Rendering-Dictionary zurück.

currentcolorscreen

Format:
```
currentcolorscreen      redfreq redang redproc
                        greenfreq greenang greenproc
                        bluefreq blueang blueproc
                        grayfreq grayang grayproc
```

Der Befehl gibt alle 12 Halbton-Bildparameter zurück, falls der aktuelle Halbton-Screen mit *setcolorscreen* oder *currentcolorscreen* definiert wurde. Wurde der Halbton-Screen mit *sethalftone* oder *currentcolorscreen* definiert, gibt der Befehl die Werte 60,0 und das Halftone Dictionary viermal zurück.

currentcolorspace

Format: `currentcolorspace array`

Der Befehl gibt ein Feld mit den Werten des Farbraums zurück. Dieses Feld kann zum Beispiel den Namen des Farbraums enthalten.

currentcolortransfer

Format: `currentcolortransfer redproc greenproc blueproc grayproc`

Mit dem Befehl werden die aktuellen Transferprozeduren für die vier Farben zurückgegeben. Wurden die Funktionen mit *setcolortransfer* oder *currentcolortransfer* definiert, enthält der Stack 4 Operanden. Mit *settransfer* gibt *currentcolortransfer* nur eine Funktion zurück.

currentdevparams

Format: `string currentdev currentdevparams dict`

Der Befehl gibt das Dictionary mit allen Keys und aktuellen Werten der Einheit (Device) zurück, die durch *string* angegeben wurde.

currentglobal

Format: `currentglobal bool`

Der Befehl gibt den aktuellen VM Allocation Mode zurück.

currentgstate

Format: `gstate currentgstate gstate`

Der Befehl ersetzt den aktuellen Wert von *gstate* durch eine Kopie des aktuellen Grafikstatus. Weiterhin wird *gstate* auf dem Operandenstack hinterlassen.

currenthalftone

Format: `currenthalftone halftone`

Der Befehl gibt das aktuelle Halftone Dictionary des Grafikstatus zurück.

currentobjectformat

Format: `currentobjectformat int`

Der Befehl gibt den aktuellen *Object Format*-Parameter zurück.

currentoverprint

Format: `currentoverprint bool`

Gibt den Wert des *Overprint*-Parameters zurück.

currentpacking

Format: `currentpacking bool`

Vom Befehl wird das Feld *Packing Mode* zurückgegeben.

currentpagedevice

Format: `currentpagedevice dict`

Der Befehl gibt ein *Dictionary* (read only) zurück, welches ein seitenorientiertes Ausgabegerät beschreibt. Der Befehl erzeugt bei Bedarf ein neues Dictionary. Ist das aktuelle Ausgabegerät nicht seitenorientiert, ist das zurückgegebene Dictionary leer.

currentshared

Format: `currentshared bool`

Der Befehl besitzt die gleiche Semantik wie *currentglobal* und wird aus Kompatibilitätsgründen beibehalten.

currentstrokeadjust

Format: `currentstrokeadjust bool`

Der Befehl gibt den aktuellen Grafikstatus-Parameter *currentstrokeadjust* zurück.

currentsystemparams

Format: `currentsystemparams dict`

Gibt ein Dictionary mit den aktuellen Systemparametern der Implementierung zurück.

currentundercolorremoval

Format: `currentundercolorremoval proc`

Vom Befehl wird die aktuelle *colorremoval*-Funktion zurückgegeben.

currentuserparams

Format: `currentuserparams dict`

Gibt das Dictionary mit den aktuellen Benutzerparametern zurück.

defineuserobject

Format: `index any defineuserobject`

Erzeugt eine Verknüpfung zwischen der positiven Integerzahl *index* und dem Objekt *any* im *UserObject.* Der Befehl speichert ein Objekt *any* an die in *index* angegebene Position im *array* des *UserObject.*

deletefile

Format: `filename deletefile`

Löscht die angegebene Datei aus dem Speicher. Gibt es die Datei nicht, löst der Interpreter einen *undefinedfilename*-Fehler aus.

execuserobject

Format: `index execuserobject`

Der Befehl führt das User-Object aus, welches in *index* angegeben wird. Das Objekt muß mit *defineuserobject* definiert worden sein, sonst erfolgt eine Fehlermeldung.

filenameforall

Format: `template proc scratch filenameforall`

Der Befehl führt alle Dateien auf, deren Namen mit dem in *template* angegebenen Muster übereinstimmen. Die Dateinamen werden in den angegebenen *srcatch*-String kopiert, und dann wird die angegebene Prozedur *proc* aufgerufen. Die Zeichen im Parameter *template* dürfen auch Wildcards * ? enthalten. Wobei * mehrere Zeichen ersetzt und ? für ein Wildcard-Zeichen steht. Beginnt ein Zeichen mit \, wird das folgende Zeichen als Literal verwendet. Beginnt *template* mit %, erfolgt der Vergleich auf komplette Dateinamen in der Form %device%file.

Für den Inhalt der Prozedur *proc* gibt es keine Einschränkungen. Bei jedem gefundenen Dateinamen wird diese Prozedur aufgerufen.

filenameposition

Format: `file filenameposition position`

Der Befehl gibt die aktuelle Position innerhalb der angegebenen Datei in *position* (positive Integerzahl) zurück.

filter

Format: `scr|tgt para1 ... paran filter file`

Vom Befehl wird eine gefilterte Datei erzeugt und in *file* zurückgegeben. Der *scr | tgt*-Operand definiert die unterlagerte Datenquelle, aus der der Filter die Daten liest. Die Quelle kann eine Datei, ein String oder eine Prozedur sein. Die Parameter $para_x$ definieren die Anpassung des speziellen Filters. Meist benötigen die Filter jedoch keine Parameter. Tabelle 48.1 gibt die Namen der Standardfilternamen an:

Name	Name
ASCIIHexEncode	ASCIIHexDecode
ASCII85Encode	ASCII85Decode
LZWEncode	LZWDecode
RunLengthEncode	RunLengtDecode
CCITTFaxEncode	CCITTFaxDecode
NullEncode	NullDecode

Tabelle 48.1 Standardfilter

findencoding

Format: `key findencoding array`

Der Befehl übernimmt den in *key* angegebenen Encoding-Vektor und legt diesen auf dem Stack ab.

findresource

Format: `key category findresource instance`

Der Befehl versucht, eine benannte Ressourceninstanz der spezifizierten Kategorie zu übernehmen. *category* ist ein Name oder ein Stringobjekt, welches die Ressourceninstanz angibt (z. B. ein Font). Der Befehl gibt eine Instanz der Ressource auf dem Stack zurück.

gcheck

Format: `any gcheck bool`

Mit dem Befehl läßt sich prüfen, ob der angegebene Operand *any* zu den grundlegenden PostScript-Befehlen gehört oder in der globalen VM als zusammengesetzter Operand existiert. In diesen Fällen wird der Wert *true* zurückgegeben. Der Wert *false* wird zurückgegeben, falls ein zusammengesetzter Operand in der lokalen VM vorhanden ist.

globaldict

Format: `globaldict dict`

Der Befehl legt das Objekt *globaldict* auf dem Operandenstack ab.

GlobalFontDirectory

Format: `GlobalFontDirectory dict`

Mit dem Befehl wird ein Dicitonary oder ein definierter Font auf dem Operandenstack hinterlegt.

glyphshow

Format: `name glyphshow`

Der Befehl zeigt ein einzelnes Zeichen aus dem aktuellen Font. Das Zeichen ist in *name* anzugeben. *glyphshow* umgeht dabei die aktuelle *Font-Encoding*-Funktion und kann damit auf alle Zeichen eines Fonts zugreifen. Die Wirkungsweise des Operators ist aber abhängig vom verwendeten Font (Type 1, Type 3).

ineofill

Format:
```
x y ineofill bool
userpath ineofill bool
```

Der Befehl besitzt zwei Varianten und ist vergleichbar mit *infill*. Der Test wird lediglich auf *eofill* und nicht auf *fill* durchgeführt.

infill

Format:
```
x y infill bool
userpath infill bool
```

Der Befehl besitzt zwei Varianten. Der erste Befehl gibt den Wert *true* zurück, falls der Bildpunkt *x,y* im Benutzerkoordinatensystem bei Anwendung des *fill*-Kommandos gezeichnet wird. Im zweiten Fall gibt der Befehl den Wert *true* zurück, falls Punkte im *userpath* bei Anwendung des *fill*-Operators gezeichnet werden.

instroke

Format:
```
x y instroke bool
userpath instroke bool
```

Der Befehl besitzt zwei Varianten. Der erste Befehl arbeitet ähnlich wie *infill*, wobei der Test sich jedoch auf die Funktion *stroke* bezieht.

inueofill

Format:
```
x y userpath inueofill bool
userpath1 userpath2 inueofill bool
```

Der Befehl besitzt zwei Varianten und wirkt ähnlich wie *inufill*, wobei sich der Test auf den Befehl *inueofill* bezieht.

inufill

Format: `x y userpath inufill bool`
`userpath1 userpath2 inufill bool`

Der Befehl besitzt zwei Varianten. Der erste Befehl gibt den Wert *true* zurück, falls der Bildpunkt *x,y* im Benutzerkoordinatensystem bei Anwendung des *ufill*-Kommandos im *Userpath* gezeichnet wird. Im zweiten Fall werden Punkte in *userpath1* als Appertur interpretiert. Dann gibt der Befehl den Wert *true* zurück, falls Punkte der Appertur im *userpath2* bei Anwendung des *ufill*-Operators gezeichnet werden.

inustroke

Format: `x y userpath inustroke bool`
`x y userpath matrix inustroke bool`
`userpath1 userpath2 inustroke bool`
`userpath1 userpath2 matrix inustroke bool`

Der Befehl besitzt mehrere Varianten. Der erste Befehl testet, ob ein Bildpunkt *x,y* auf dem aktuellen Device mit *inustroke* gezeichnet wird, und gibt dann den Wert *true* zurück. Im zweiten Befehl wird nach der Interpretation des User-Pfads die Matrix zur CTM (coordinate transfer matrix) hinzugefügt. Im dritten und vierten Befehl werden die Pixel im *userpath1* als *Appertur* interpretiert. Wird ein solches Pixel von *ustroke* gezeichnet, gibt *inustroke* den Wert *true* zurück.

ISOLatin1Encoding

Format: `ISOLatin1Encoding array`

Der Befehl legt einen ISO-Latin1-Encoding-Vektor auf dem Operandenstack ab. Dieser Vektor ist ein Array mit 256 Einträgen, welches durch die Zeichencodes indiziert wird.

languagelevel

Format: `languagelevel int`

Der Befehl ermittelt die Sprachversion des PostScript-Interpreters und gibt diese auf dem Stack zurück. Der Wert 2 signalisiert, daß alle Level 2-Befehle implementiert sind.

makepattern

Format: `dict matrix makepattern pattern`

Der Befehl erzeugt ein Muster. Zuerst wird geprüft, ob *dict* ein Pattern-Dictionary ist, welches alle erforderlichen Werte enthält. Dann wird eine Kopie in der lokalen VM erzeugt. Der Inhalt von *matrix* wird in der CTM gesichert. Als Resultat wird das Muster-Dictionary auf dem Stack zurückgegeben. Das Muster ist dann über *setpattern* aus dem Dictionary abrufbar.

packedarray

Format: `any0 ... anyn n packedarray packedarray`

Der Befehl erzeugt aus den Einträgen any_0 ... any_n ein gepacktes Array der Länge *n* und legt es auf dem Operandenstack ab.

printobject

Format: `obj tag printobject`

Schreibt eine binäre Objektsequenz in die Standard-Ausgabedatei. Die binäre Objektsequenz enthält ein Top-Level Array der Länge 1. Dieses Element stellt die Kodierung für *obj* dar. *tag* ist ein Wert zwischen 0 und 255 und bezieht sich auf das *top level object*.

product

Format: `product string`

Mit dem Befehl äßt sich ein *string* (read only) abrufen, der die Umgebung des PostScript-Interpreters angibt.

realtime

Format: `realtime int`

Der Befehl erlaubt die Abfrage der Uhrzeit, die in Millisekunden in *int* zurückgegeben wird.

rectclip

Format:
```
x y width heigth rectclip
numarray rectclip
numstring rectclip
```

Der Befehl schneidet ein Rechteck aus dem aktuellen Clipping-Pfad. In der ersten Variante wird ein Rechteckpfad anhand des Startpunkts und der Abmessungen erzeugt. In den beiden anderen Varianten wird der Operand als *array* oder als *string* angegeben. Damit lassen sich mehrere Rechtecke angeben. Nach der Berechnung des Clipping-Pfades wird *current path* wie bei *newpath* auf *empty* gesetzt.

rectfill

Format:
```
x y width heigth rectfill
numarray rectfill
numstring rectfill
```

Der Befehl besitzt mehrere Varianten. Mit diesem Befehl läßt sich ein Pfad, bestehend aus einem oder mehreren Rechtecken, in der aktuellen Farbe füllen. Die Parameter besitzen die gleiche Bedeutung wie bei *rectclip*.

rectstroke

Format:
```
x y width heigth rectstroke
x y width heigth matrix rectstroke
numarray rectstroke
numarray matrix rectstroke
numstring rectstroke
numstring matrix rectstroke
```

Der Befehl besitzt mehrere Varianten. Er führt einen *stroke*-Befehl auf einen Pfad mit einem oder mehreren Rechtecken aus. Ist der Parameter *matrix* vorhanden, fügt der Interpreter den Inhalt in der CTM ein. Die restlichen Parameter besitzen die Bedeutung wie bei *rectclip*.

rectviewclip

Format:
```
x y width heigth rectviewclip
numarray rectviewclip
numstring rectviewclip
```

Der Befehl ersetzt den aktuellen Bildausschnitt (*view clip*) durch einen Rechteckpfad. Für die Parameter gelten die gleichen Bedingungen wie bei *rectclip*.

renamefile

Format: `old new renamefile`

Der Befehl benennt eine Datei mit dem Namen *old* in den neuen Namen *new* um. Die Parameter *old* und *new* sind Strings. Existiert die Datei nicht, wird ein *undefinedfilename*-Fehler ausgelöst. Ist eine Umbenennung nicht erlaubt, wird ein *invalidfileaccess*-Error ausgelöst.

resourceforall

Format: `template proc scratch category resourceforall`

Der Befehl zählt alle Namen der Instanzen einer angegebenen Ressourcenkategorie auf. *category* ist der Objektname einer Ressourcenkategorie (z. B. Fonts). In *template* wird ein String übergeben, der als Suchmuster für die Namen interpretiert wird. Dabei sind auch die Wildcards * ? erlaubt. Mit dem Zeichen \ wird das folgende Zeichen als Literal interpretiert (z. B. bedeutet *, daß ein * nicht als Wildcard zu interpretieren ist). Die gefundenen Namen werden im String *scratch* gespeichert. Für jeden gefundenen Namen wird die Prozedur *proc* aufgerufen.

resourcestatus

Format: `key category resourcestatus status size bool`

Der Befehl gibt den Status der in *key* angegebenen Ressourceninstanz zurück. In *category* läßt sich eine Kategorie der Ressource (z. B. Font) festlegen. Existiert die Ressource, gibt

der Befehl in *bool* den Wert *true* zurück. Dann enthält der Stack zwei zusätzliche Parameter. *status* kann folgende Werte aufweisen:

0 = in VM per *defineresource* definiert
1 = in VM durch den vorherigen Befehl *findresource* definiert
2 = nicht in VM definiert, aber über externen Speicher verfügbar

Der Integerwert *size* gibt an, wie viele Bytes die Ressource ungefähr in der VM verbraucht. Existiert die Ressource nicht, gibt der Befehl nur den Wert *false* auf dem Stack zurück.

rootfont

Format: `rootfont font`

Der Befehl gibt den zuletzt mit *setfont* oder *selectfont* ausgewählten Font zurück. Normalerweise ist der Wert mit der durch *currentfont* ermittelten Größe identisch. Beim Aufruf aus *cshow* gibt der Befehl den *composite font* zurück.

scheck

Format: `any scheck bool`

Der Befehl besitzt die gleiche Funktion wie *gcheck* und wird aus Kompatibilitätsgründen beibehalten.

selectfont

Format:
```
key scale selectfont
key matrix selectfont
```

Der Befehl übernimmt einen Font, dessen Namen in *key* angegeben wird. Der Font wird bei der Übernahme gemäß *scale* oder *matrix* transformiert. In Level 1 konnte der gleiche Effekt mit der Sequenz:

```
/Times Roman findfont
10 scalefont
setfont
```

erreicht werden. In Level 2 ist nur noch die Anweisung:

```
/Times Roman 10 selectfont
```

erforderlich.

serialnumber

Format: `serialnumber int`

Der Befehl ermittelt die Seriennummer des Interpreters und gibt diese als Integerwert auf dem Stack zurück.

setbbox

Format: `llx lly urx ury setbbox`

Der Befehl legt die Bounding-Box des aktuellen Pfades fest, wobei die Bounding-Box durch zwei Koordinatenpaare definiert wird. *llx lly* definiert die linke untere Ecke (lower left corner), *urx* und *ury* definieren die obere rechte Ecke (upper right corner). Die Bounding-Box bleibt so lange erhalten, wie der Pfad gültig ist.

setblackgeneration

Format: `proc setblackgeneration`

Der Befehl definiert die Parameter *proc* der Prozedur *blackgeneration* im Grafik-State. Als Operand wird in *proc* eine Prozedur übergeben. Die Prozedur muß sich mit einer Zahl zwischen 0.0 und 1.0 aufrufen lassen. Aus diesem Wert (Graustufe) wird dann der neue Wert durch die Prozedur bestimmt und durch diese auf dem Stack zurückgegeben.

setcachedevice2

Format: `w0x w0y llx lly urx ury w1x w1y vx vy setcachedevice2`

Der Befehl übergibt zwei Sets der Zeichenmetrik an die Fontmaschine. *w0x* und *w0y* definieren den Abstand zum aktuellen Punkt zum neuen *current point* für die Textausgabe im Modus 0. *llx*, *lly* und *urx*, *ury* definieren die Bounding-Box bezogen auf den aktuellen Punkt für die Zeichen. *w1x* und *w1y* definieren den Abstand vom aktuellen Punkt zum neuen *current point*, wenn der Text im Mode 1 ausgegeben wird. *vx* und *vy* geben dann den Abstand vom Punkt 0 zum Punkt 1 an. Nach *setcachedevice2* bis zum Ende von *BuildGlyph* oder *BuildChar* sind Befehle zur Farbauswahl oder *imagemask* nicht zulässig.

setcacheparams

Format: `mark size lower upper setcacheparams`

Mit dem Befehl werden die Cacheparameter gesetzt. Die Zahl der Parameter wird in *size* angegeben.

setcmykcolor

Format: `cyan magenta yellow black setcmykcolor`

Dieser Befehl erlaubt es, die aktuelle Farbe im CMYK-Farbraum zu definieren, wobei die Farbdefinition über die Farbanteile *cyan, magenta, yellow* und *black* erfolgt. Die Werte dieser Parameter müssen zwischen 0.0 und 1.0 liegen.

setcolor

Format: `comp1 ... compn setcolor`

Mit dem Befehl werden die aktuellen Farbparameter im *graphic state* gesetzt. Die Zahl der Komponenten *comp* und deren Werte variieren in Abhängigkeit vom verwendeten Farbsystem.

setcolorrendering

Format: `dict setcolorrendering`

Der Befehl installiert *dict* als das aktuelle CIE-basierte *color rendering*-Dictionary im Graphic-State. Damit läßt sich die Farbausgabe an bestimmte Geräte anpassen.

setcolorscreen

Format:
```
redfreq redang redproc
greenfreq greenang greenproc
bluefreq blueang blueproc
grayfreq grayang grayproc setcolorscreen
```

Mit dem Befehl lassen sich die Halbton-Parameter im Graphic-State setzen. Die Parameter geben die Farbanteile Rot, Grün, Blau und Grau mit den jeweiligen Komponenten an.

setcolorspace

Format:
```
array setcolorspace
name setcolorspace
```

Der Befehl definiert den zu verwendenden Farbraum. Das *array* muß dabei im Format *[key* $param_1$ *...* $param_n$*]* angegeben werden. In der zweiten Variante reicht es, den Namen des Farbsystems (DeviceGray, DeviceRGB, DeviceCMYK, Pattern) anzugeben.

setcolortransfer

Format: `redproc greenproc blueproc grayproc setcolortransfer`

Der Befehl definiert eine Transferfunktion für die Grafikmaschine. Jeder der Operanden für Rot, Grün, Blau und Grau muß eine PostScript-Prozedur sein, die Parameter im Bereich zwischen 0.0 und 1.0 erwartet. Dieser Parameter wird durch die Transferfunktion in einen Wert im Bereich 0.0 bis 1.0 umgerechnet.

setdevparams

Format: `string dict setdevparams`

Mit dem Befehl lassen sich ein oder mehrere Parameter für ein in *string* angegebenes Gerät übergeben. Die Schlüssel und Werte befinden sich in *dict*. Jeder Wert wird dabei durch einen Schlüssel *key* im Dictionary definiert.

setfileposition

Format: `file position setfileposition`

Der Befehl repositioniert den Zeiger einer geöffneten Datei *file* auf die in *position* angegebene Position. *position* muß eine positive Integerzahl sein. Bei einer Positionierung über das Dateiende hinaus hängt die Reaktion von der Implementierung des Dateisystems ab.

setglobal

Format: `bool setglobal`

Der Befehl setzt den VM Allocation Mode. Mit *true* wird der Modus auf *global* festgelegt, während *false* den VM-Modus auf *lokal* einstellt. Damit wirken sich *save*- und *restore*-Befehle auf die lokale oder globale VM aus.

setgstate

Format: `gstate setgstate`

Der Befehl ersetzt den aktuellen Grafikstatus (graphics state) durch den Inhalt des Objekts *gstate*.

sethalftone

Format: `halftone sethalftone`

Mit dem Befehl wird der Halbtonparameter im aktuellen *graphic state* gesetzt.

setobjectformat

Format: `int setobjectformat`

Der Befehl legt das Objektformat für binäre Sequenzen von *printobject* und *writeobject* gemäß Tabelle 48.2 fest.

Code	Objekt
0	Disable binäre Kodierung
1	High-order byte first, IEEE Standard Real Format
2	Low-order byte first, IEEE Standard Real Format
3	High-order byte first, native Real Format
4	Low-order byte first, native Real Format

Tabelle 48.2 Kodierung der binären Objektformate

Der Wert 0 sperrt alle binären Kodierungen für die Ausgabewerte. Damit werden die Ausgaben als ASCII-Strings kodiert.

setoverprint

Format: `bool setoverprint`

Mit diesem Befehl wird der *setoverprint*-Parameter im *graphic state* gesetzt. Der Befehl wird verwendet, wenn die Einheit Farbseparationen unterstützt. Mit *false* werden die anderen Separationen gelöscht, und mit *true* bleiben diese erhalten.

setpacking

Format: `bool setpacking`

Der Befehl setzt den Packmodus für Arrays gemäß dem spezifizierten Wert. Mit *true* werden gepackte Arrays selektiert.

setpagedevice

Format: `dict setpagedevice`

Mit dem Befehl kann ein Systemadministrator ein neues Rasterausgabegerät im Interpreter installieren.

setpattern

Format: `pattern setpattern`
`comp1 .... compn pattern setpattern`

Der Befehl ermöglicht die Definition eines Musters in der aktuellen Farbe im *graphic state*. In *pattern* wird das Muster als *Dictionary* definiert (wird durch *setpattern* erzeugt). Definiert *pattern* ein farbloses Muster, enthalten die Parameter $comp_x$ die betreffenden Farben, die dem Muster unterlegt werden.

setrgbcolor

Format: `red green blue setrgbcolor`

Eine Farbe im RGB-Farbraum läßt sich mit *setrgbcolor* definieren. Die Werte für *red*, *green* und *blue* müssen dabei zwischen 0.0 und 1.0 liegen.

setshared

Format: `bool setshared`

Der Befehl besitzt die gleiche Semantik wie die Anweisung *setglobal* und wird aus Kompatibilitätsgründen beibehalten.

setstrokeadjust

Format: `bool setstrokeadjust`

Mit dem Befehl wird der Parameter *stroke adjust* im aktuellen Grafikstatus (graphics state) beeinflußt. Mit dem Wert *true* wird die Justierung für *stroke* aktiviert.

setsystemparms

Format: `dict setsystemparms`

Der Befehl erlaubt es, einen oder mehrere Systemparameter zu setzen. Die Schlüssel *keys* und die Werte dieser Systemparameter sind in *dict* zu übergeben. Die Änderung der Systemparameter wird in der Regel durch ein Kennwort geschützt. Dieses Kennwort kann durch den Eintrag *Password* im Dictionary gesetzt werden.

setucacheparms

Format: `mark blimit setucacheparms`

Mit *setucacheparms* wird der Parameter des *user path cache* beeinflußt. Die Integerparameter liegen oberhalb des mit *mark* markierten Elements auf dem Stack. In *blimit* wird die Zahl der belegbaren Bytes für einen Pfad im *user path cache* festgelegt.

setundercolorremoval

Format: `proc setundercolorremoval`

Der Befehl übergibt eine Prozedur, mit der eine Farbsubtraktion für RGB, GRAY oder CMYK realisiert wird. Die der Prozedur übergebenen Werte liegen im Bereich zwischen 0.0 und 1.0. Der Interpreter erwartet die Ergebnisse im gleichen Wertebereich.

setuserparams

Format: `dict setuserparams`

Der Befehl besitzt die obige Syntax und ermöglicht es, User-Pparameter über den Dictionary-Eintrag *dict* zu setzen.

setvmthreshold

Format: `int setvmthreshold`

Mit dem Befehl wird die Schwelle für den Beginn der *Garbage Collection* gesetzt. Mit -1 wird die in der Implementierung definierte Schwelle verwendet.

shareddict

Format: `shareddict dict`

Der Befehl wirkt wie *globaldict* und wird aus Kompatibilitätsgründen zu Display-PostScript beibehalten.

sharedfontdirectory

Format: `sharedfontdirectory dict`

Der Befehl wirkt wie *globalfontdirectory* und wird aus Kompatibilitätsgründen zu Display-PostScript beibehalten.

startjob

Format: `bool password startjob bool`

Mit *startjob* wird ein neuer Job im PostScript-Interpreter gestartet.

uappend

Format: `userpath uappend`

Mit dem Befehl wird die *userpath*-Definition interpretiert und das Ergebnis an den aktuellen Pfad angehängt.

ucache

Format: `ucache`

Der Befehl veranlaßt, daß der aktuelle *user path* im Cache zu halten ist.

ucachestatus

Format: `ucachestatus mark bsize bmax rsize rmax blimit`

Der Befehl ermittelt den Status des User-Cache.

ueofill

Format: `userpath ueofill`

Der Befehl wirkt wie *ufill*, verwendet aber die *eofill*-Funktion anstelle von *fill*.

ufill

Format: `userpath fill`

Mit dem Befehl wird die Definition des *userpaths* interpretiert und der resultierende Pfad mit *fill* gefüllt. Die Operation wird dabei mit *gsave* und *grestore* eingeschlossen.

undef

Format: `dict key undef`

Der Befehl entfernt den Schlüssel *key* und den zugehörigen Wert aus dem angegebenen Dictionary *dict*.

undefinedresource

Dies ist ein Fehler, der auftreten kann, falls eine nicht definierte Ressource aufgerufen wird.

undefinefont

Format: `key undefinefont`

Der Befehl entfernt den Eintrag *key* und den zugehörigen Wert aus dem Font-Dictionary.

undefineresource

Format: `key category undefineresource`

Mit dem Befehl läßt sich die in *key* benannte Ressource der angegebenen Kategorie (z. B. Font) entfernen.

undefineuserobject

Format: `index undefineuserobject`

Der Befehl hebt die Verknüpfung zwischen dem User-Objekt und dem angegebenen Index auf.

upath

Format: `bool upath userpath`

Mit dem Befehl wird ein neues *user path*-Objekt erzeugt, welches dem aktuellen Pfad entspricht. Der Parameter *bool* definiert, ob *ucache* als erstes Element in den Pfad aufzunehmen ist.

UserObjects

Format: `UserObjects array`

Der Befehl gibt ein Array der User Objects des aktuellen *userdicts* zurück.

ustroke

Format: `userpath ustroke`
`userpath matrix ustroke`

Mit dem Befehl wird der Userpfad interpretiert und ein *stroke*-Befehl ausgeführt. *ustroke* wird mit *gsave* und *grestore* eingeschlossen.

ustrokepath

Format: `userpath ustrokepath`
`userpath matrix ustrokepath`

Der Befehl ersetzt den aktuellen Pfad mit einem Pfad, der bei Anwendung von *ustroke* entsteht.

vmreclaim

Format: `int vmreclaim`

Der Befehl kontrolliert die *Garbage Collection* des Speichers über den Parameter *int*. Mit *-2* wird die automatische Speicherbereinigung in der lokalen und globalen VM abgeschaltet. Mit *-1* bezieht sich die Sperre nur auf die lokale VM. Mit *0* wird die automatische Speicherbereinigung zugelassen. *1* bewirkt eine sofortige Speicherbereinigung in der lokalen VM, während *2* sich auf beide VMs bezieht.

writeobject

Format: `file obj tag writeobject`

Der Befehl schreibt eine Sequenz binärer Objekte in die angegebene Datei. Der Dateiname wird in *file* angegeben. Der Befehl wirkt wie *printobject*.

xyshow

Format: `string numarray xyshow`
`string numstring xyshow`

Mit diesem Befehl wird ein *string* ähnlich wie bei *show* ausgegeben. Nach der Ausgabe eines Zeichens werden aus *numarray* oder *numstring* zwei aufeinanderfolgende Zahlen gelesen, die die Koordinaten der nächsten Ausgabeposition im User-Koordinatensystem angeben. Die erste Zahl gibt die relative x-Position zum zuletzt ausgegebenen Zeichen an. Der andere Parameter bezieht sich auf die y-Position.

yshow

Format: `string numarray yshow`
`string numstring yshow`

Der Befehl ist vergleichbar *xyshow*, wobei jedoch nur ein Wert für y aus *numarray* oder *numstring* bezogen wird.

Die obige Aufstellung enthält nur die wichtigsten Informationen zu den PostScript Level-2-Befehlen. Die Beschreibung aller Feinheiten würde ein eigenes Buch umfassen. Hier möchte ich auf das bei Addison-Wesley publizierte Werk »PostScript Language Reference Manual« verweisen.

Index

D

E

F

G

H

I

Q

R

S

T

V

W

X

Z