Dieses Kapitel stammt aus dem Buch

›Dateiformate – Eine Referenz‹

von Günter Born.

www.borncity.de

ISBN 3-934358-83-7
119,90 DM

Informationen zum Buch
mit Bestellmöglichkeit

www.galileocomputing.de

Galileo Computing

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.

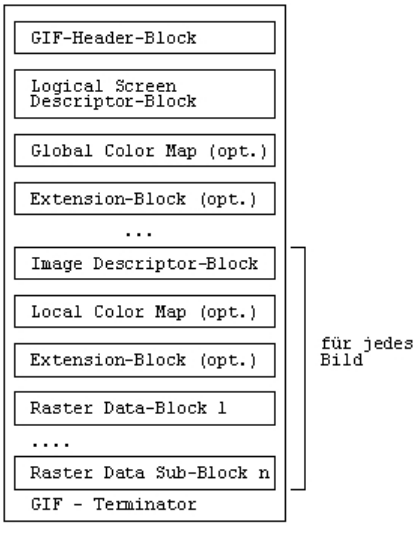

Abbildung 22.1 Struktur einer GIF-Datei

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.

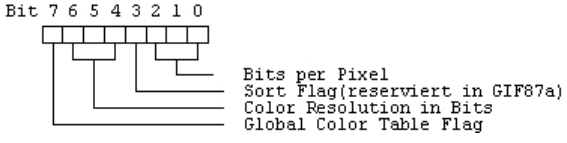

Abbildung 22.2 Kodierung des Resolution-Flag

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.

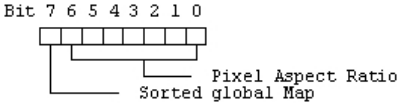

Abbildung 22.3 Kodierung des Pixel Aspect Ratio-Flag (GIF87a)

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).

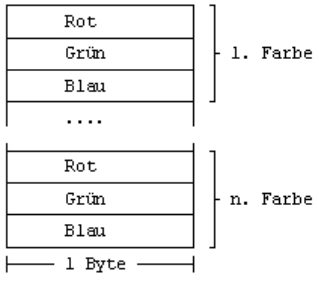

Abbildung 22.4 Struktur der Global Color Map

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.

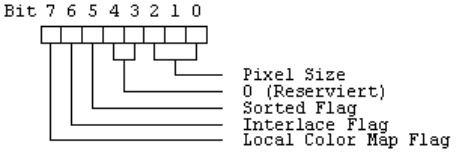

Abbildung 22.5 Kodierung des Flags im Image Descriptor-Block (IDB)

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.

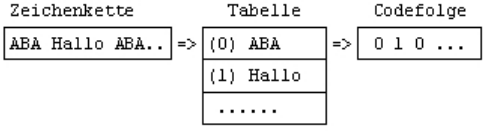

Abbildung 22.6 LZW-Encoding

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:

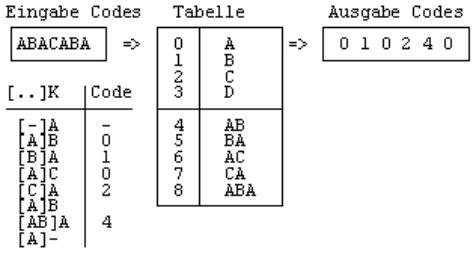

Abbildung 22.7 LZW-Kompression

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.

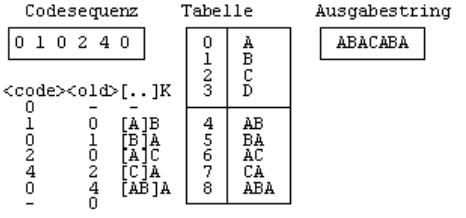

Abbildung 22.8 LZW-Dekodierung

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.

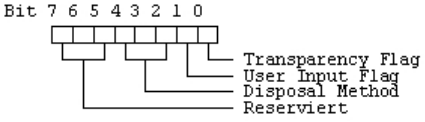

Abbildung 22.9 Kodierung des Flag im Graphics Control Extension-Block

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.