Alleskönner

Das IFF-Dateiformat

Schluß mit der Formatvielfalt, etwas Universelles muß her. Und das gibt es schon. Wie würde es Ihnen gefallen, Grafiken, Texte oder beliebiges anderes mit ein und demselben Format speichern zu können? Dieselben Routinen können beliebige Daten lesen. Mit IFF ist das kein Problem.

Das ist das Schöne an Standards: Man hat die Auswahl. Die amerikanische Softwarefirma „Electronic Arts", ihres Zeichens Spielehersteller und Produzent des Grafikprogramms „Deluxe Paint", hatte angesichts der großen Zahl an Standards für Grafik-, Text- und Sonstwas-Dateien die Faxen dicke und dachte sich etwas Neues aus. Der IFF-Standard, „Interchange File Format", wurde geboren. Der Hauscomputer dieses Standards war zunächst der Amiga, denn dieser konnte wegen seiner damals herausragenden Grafik- und Soundeigenschaften am ehesten einen universellen Aufzeichnungsstandard vertragen. Wie der Name schon sagt, soll IFF keineswegs nur auf den Amiga beschränkt sein.

Grafiken werden zwar am häufigsten in den IFFs gespeichert, genausogut können sich darin aber auch Texte, Musikstücke, Daten einer Datenbank, Arbeitsblätter einer Tabellenkalkulation oder, und das ist wieder Amiga-typisch, digitalisierte Töne und Geräusche verbergen. Von Electronic Arts wurden vier dieser Anwendungen bereits vorgesehen. Und das ist das Interessante an IFF: Man kann die Files beliebig mit eigenen Daten-Kreationen erweitern, ohne daß die Daten zu bestehenden Anwendungen inkompatibel werden.

Das Geheimnis steckt in der internen Datenaufteilung eines IFFs. Jedes IFF besteht aus zwei Datenbereichen: Aus Forms und aus Chunks. Dabei faßt ein Form mehrere Chunks zusammen. Das Form ist sozusagen der Oberbegriff eines Files. Nehmen wir an, wir wollen die Daten eines Bildes speichern. Dazu müssen mindestens die Daten des Bildes ins File. Zudem sollten die verwendeten Farben gespeichert werden, und sinnvoll wäre außerdem noch die Auflösung, damit PCs mit VGA-Karte zum Beispiel automatisch den richtigen Modus wählen. Dann hätten wir schon drei Datenbereiche, sprich drei Chunks. Die drei fassen wir zu einem Bilder-Form zusammen.

Anhand des Forms erkennt ein Grafik-Programm jetzt, daß die Daten auch tatsächlich ein Bild enthalten, und keine Texte oder Datensätze einer Datenbank. Innerhalb des Forms sucht sich das Programm dann die Farbtabelle, die Auflösung und die Bilddaten selbst zusammen. Ein Grafikprogramm könnte innerhalb des Forms vielleicht noch Daten speichern, mit denen das Bild auf dem Bildschirm verschoben wird, um so

vielleicht für Präsentationen einen Effekt zu erzeugen. Da unser Programm diese Daten weder kennt noch benötigt, werden sie einfach übergangen, und wir haben trotzdem unsere Grafik auf dem Bildschirm.

Der Aufbau eines Files

Der theoretische Aufbau eines Forms ist in Bild 1 gezeigt. Jedes Form, und damit auch jedes IFF, beginnt mit vier Byte, die im ASCII die Buchstaben „FORM" enthalten. Anhand dieser Kennung kann man ein IFF erkennen. Nach diesen vier Byte folgen wiederum vier Byte, die die Länge des Forms enthalten. Die Zählung beginnt dabei mit dem ersten Byte hinter der Längenangabe. Demnach ist ein komplettes File immer acht Byte länger als die Angabe im Form.
Da das IFF für den Amiga entwickelt wurde, sind die Bytes der Längenangabe nach Motorola-Notation gespeichert. Bei diesen wird das höchstwertige Byte im Gegensatz zu der Intel-Konvention immer als erstes geschrieben. Man muß diese Bytes im Programm also in einer eigenen kleinen Routine umrechnen. Außerdem fängt die Zählung bei eins an, und nicht wie bei einigen anderen Formaten bei Null.
Noch eines sollte man beachten, wenn man die Länge des IFF prüfen will. Hinter dem ersten Form können im IFF weitere Forms folgen. Diese haben dann wieder ihren eigenen Vier-Byte-Kopf und eine eigene Längenangabe. Sinnvoll ist das zum Beispiel, wenn man zu einem Bild eine Musik speichern möchte, die dann bei einer Präsentation gespielt wird. In diesem Fall muß man sich bei der Überprüfung der Länge mit der Angabe im Form bis ans tatsächliche Dateiende des IFF hangeln, und dann die Länge mit der im Directory-Eintrag vergleichen.
Hinter den vier Byte mit der Längenangabe stehen wieder vier Byte, die den Typ des Forms bezeichnen. Hier wird festgelegt, ob es sich bei dem Form um ein Bild, eine Musik, einen Text oder sonst irgend etwas handelt. In Tabelle 1 sehen Sie die Bezeichnungen der häufigsten Form-Typen. Diese zwölf Byte bilden den Form-Header.

Dateien klötzchenweise

Ein Form ist normalerweise aus mehreren Chunks aufgebaut. Es ist natürlich auch möglich, nur einen Chunk im Form zu speichern, oder einen Form aufzubauen, der keine Chunks enthält. In letzterem Fall besteht der Form nur aus 8 Byte, seine Längenangabe ist Null. In der Praxis könnte man so etwas als reine Markierung verwenden, bisher nutzt das noch kein Programm.
Auch ein Chunk hat wieder einen Kopf, an dem man seinen Zweck erkennt. Dieser besteht wieder aus vier Byte, die eine Buchstabenkombination enthalten und vier Byte, die die Länge des Chunks angeben. Wie beim Form-Header zählt die Länge erst hinter dem ersten Byte nach der Längenangabe, und auch hier muß man die Motorola-Notation beachten. Bild 2 zeigt den generellen Aufbau eines Chunks.
Auch mit den Längenangaben in den Chunks kann man prüfen, ob ein IFF richtig aufgebaut ist. Dazu addiert man die Längenangaben aller Chunks und rechnet nochmals acht Byte für jeden Chunk hinzu. Diese Längenangabe muß exakt der Längenangabe im Form entsprechen. Mit Sicherheit läuft im IFF auch etwas schief, wenn der ASCII-Wert eines der vier Byte in der Form- oder Chunk-Bezeichnung nicht zwischen 32 und 95 liegt. Das sind die Zahlen mit den Großbuchstaben und einigen Sonderzeichen. Man sollte sich bei eigenen Form- oder Chunk-Kreationen auf diese Zeichen beschränken. Wenn die vier Buchstaben partout nicht ausreichen sollten, kann man innerhalb des Chunk-Datenbereiches noch eine weitere Kennung postieren. Bild 3 zeigt den kompletten Aufbau eines IFF mit zwei Forms und drei Chunks.
Damit wäre das Interchange File Format schon beschrieben. Listing 1 enthält eine C-Routine, die einen Chunk innerhalb des ersten Form sucht und lädt. Listing 2 kopiert in ein bestehendes IFF einen Chunk ein. Der Routine muß der Chunk-Name, die Länge des Chunks und der Name des Chunks übergeben werden, nach dem der neue Chunk eingefügt werden soll. Listing 3 löscht einen Chunk aus einem File. Beide Routinen legen ein neues File mit der Bezeichnung „NEW.IFF" an. Mit Listing 4 erkennt man ein IFF. Dazu übergibt man der Routine die Kennung des gesuchten Forms. Alle vier Routinen erwarten ein bereits geöffnetes File, dessen File-Handle als Pointer übergeben wird.
Das erreicht man mit

```
FILE *fp;
fp = fopen("Filename.Ext", "rb");
```

aus STDIO der C-Standard-Bibliothek. Die Variable fp übergibt man anschließend den Funktionen. Nach Aufruf der Funktion muß man selbstverständlich das File wieder schließen.
Wenn die Routine die Arbeit ordnungsgemäß beendet, gibt sie als Return-Wert eine „1". Lief irgend etwas schief, daß zum Beispiel ein Chunk nicht gefunden wurde, wird dem aufrufenden Programm eine Null zurückgegeben. Die Routinen sind getestet und funktionstüchtig, sollen aber nur als Anregungen für eigene Programmarbeiten gesehen werden. Sinnvoll wären zum Beispiel noch Erweiterungen für spezifische Fehlercodes, die die Routinen zurückliefern.

Ins Bilder-Eingemachte

Das schönste Standard-Format mit dem kompatibelsten Aufbau nützt natürlich gar nichts, wenn es dafür keine Anwendungen gibt. Die häufigste Anwendung für IFFs ist das Speichern von Bildern. Das ist einer der Gründe, warum IFF gerne als Grafik-Format bezeichnet wird, was es eigentlich nicht ist. Für die IFFs gibt es aber einen Standard im Grafik-Aufbau, die sogenannten „Interleave Bitmaps". Deren Abkürzung im Form-Header beträgt „ILBM". Meistens erkennt man ein ILBM-IFF schon an der Erweiterung im Filenamen. Wenn ein File beim PC

und beim Atari ST mit „.LBM" endet, kann man mit Sicherheit auf ein IFF tippen. Beim Amiga, dessen File-Namensgebung wesentlich freier ist, heißen die IFFs „.ILBM" oder „.ilbm". Für eigene Programme sollte man sich an diese Erweiterungen halten.
Für die ILBM-Forms gibt es eine ganz Reihe von Chunks, mit denen man die Daten eines Bildes speichert. In Tabelle 2 sind die wichtigsten aufgeführt. Um ein Bild sicher speichern zu können, benötigt man davon allerdings nur drei Chunks: Den „BMHD", der die Auflösung und die Anzahl der Farben enthält, den „CMAP"-Chunk, in dem die benutzte Farbpalette gespeichert wird, und den „BODY"-Chunk, der die eigentlichen Bilddaten enthält.
Neben der Auflösung enthält der BMHD-Chunk, der „Bitmap Header", noch einige weitere Kleinigkeiten (siehe Tabelle 3). Die Zahlenangaben sind im Byte- und im Wortformat gespeichert. Bei letzteren muß man wie gehabt beachten, daß das High-Byte vor dem Low-Byte kommt. Interessant ist dabei die Möglichkeit, die Daten des Bildes komprimiert speichern zu können. Sind die Daten komprimiert, macht man dieses mit einer 1 im „Compression"-Byte des BMHD deutlich. Abhängig davon werden die Daten im BODY-Chunk unterschiedlich ausgewertet.
Die Anzahl der Daten im CMAP-Chunk hängt von der „Planes" im BMHD ab. Planes gibt die Anzahl der Bits an, die für ein Pixel zur Verfügung stehen, und damit auch die Anzahl der Farben des Bildes. Ein EGA-Bild mit 16 Farben benötigt vier Bit pro Pixel, CGA würde mit 2 Bit auskommen, MCGA mit 256 Farben benötigt 8 Bit pro Pixel. Für jede benutzte Farbe wird im CMAP-Chunk eine Drei-Byte-Farbkombination gespeichert. Das erste Byte gibt den Rotanteil der Farbe an, das zweite den grünen und das dritte den blauen Anteil der Farbe. Für ein Bild mit 16 Farben ist der CMAP-Chunk demnach 48 Byte lang. In drei Bytes könnte man dadurch theoretisch 2 hoch 24, also über 16 Millionen Farben, speichern. Nicht benutzte Bits werden im Chunk einfach auf Null gesetzt. Wichtig ist bei den Farbkombinationen noch, daß die benutzten Bits immer linksbündig im Byte stehen, also die höherwertigen Bits als erstes belegt werden. Im BODY-Chunk ist schließlich das Wichtigste gespeichert: die Bilddaten. Amiga-typisch werden diese nicht einfach als Speicherabbild ins File übertragen, sondern jedes Bit der Pixel einzeln. Nehmen wir an, wir hätten ein Bild mit vier Bit pro Pixel in der Auflösung 320 mal 200. Im Chunk werden als erstes die niedrigstwertigen Bits der ersten Zeile gespeichert. Dann kommen die zweitwertigen Bits, die drittwertigen und zum Schluß die höchstwertigen. Erst dann geht es mit der zweiten Zeile mit den niedrigstwertigen Bits weiter (siehe Bild 4). Das Bild wird also zeilenweise auseinandergenommen. Beim Amiga läßt sich das recht einfach machen, da die Daten der einzelnen Bits sowieso in getrennten Speicherbereichen stehen. Ähnlich verhält es sich bei EGA- und VGA-Karten des PCs. Schwieriger wird es schon beim Atari ST, der die Bits wortweise zusammenfaßt, oder bei CGA-Karten.
Die Anzahl der Bits für eine Zeile entspricht der horizontalen Auflösung, die im BMHD angegeben ist. Für 320 Pixel werden pro Zeile 40 Byte gespeichert. Wenn die Auflösung so gewählt wurde, daß ein Byte nicht voll ausgefüllt wird, bleiben die restlichen Bits des Byte leer. Auch hier werden die Bits beim höchstwertigen beginnend benutzt. Wer ein Bild mit 20 Pixel horizontaler Auflösung speichert, braucht drei Byte pro Zeile, das letzte Byte hat vier leere Bits.
Wenn die Daten komprimiert wurden, bleibt die Zeilenaufteilung bestehen. Hier werden die Daten aber in Steuerbyte und Datenbyte aufgeteilt. Zu Beginn steht immer ein Steuerbyte. Dieses gibt an, wieviele Bytes folgen, und wie diese interpretiert werden (siehe Tabelle 4). Diese recht einfache Kompressionsmethode gilt aber nur dann, wenn im Compression-Byte des BMHD-Chunks eine 1 steht. Wird dort eine andere Zahl benutzt, weicht die Methode der Komprimierung ab. Listing 5 ist eine C-Funktion, die komprimierte Daten entpackt. Der Funktion müssen die Speicherbereiche für die zu entpackenden und die fertig entpackten Daten übergeben werden. Außerdem braucht sie noch die Auflösungen in Pixel und die Anzahl der Bits pro Pixel. Mit kleinen Modifikationen kann man das Programm so umbauen, daß die Daten nicht in den Speicher, sondern gleich auf den Bildschirm geschrieben werden. *hf*

Literatur

Jennrich, B.; Schulz, P.; Bleek, W.: Amiga Intern, Band 2, Data Becker.

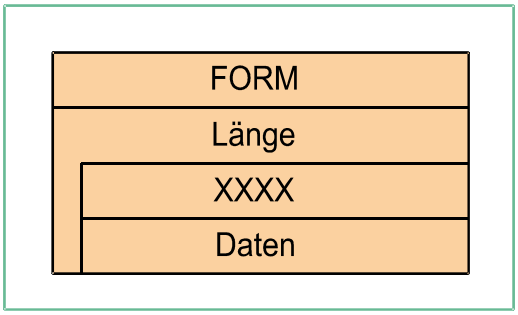

Bild 1. Jedes IFF beginnt mit den vier Buchstaben „FORM". Danach folgt die Länge des Forms und die Bezeichnung des Formtyps.

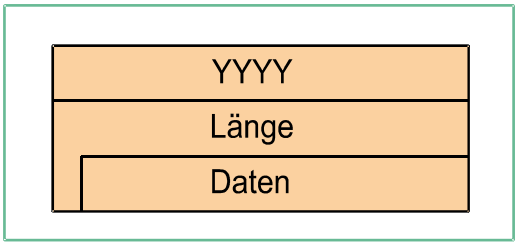

Bild 2. Die Chunks in den Forms sind ähnlich aufgebaut wie Forms. Zu Beginn steht die Chunk-Bezeichnung, dann folgt die Länge des Chunks.

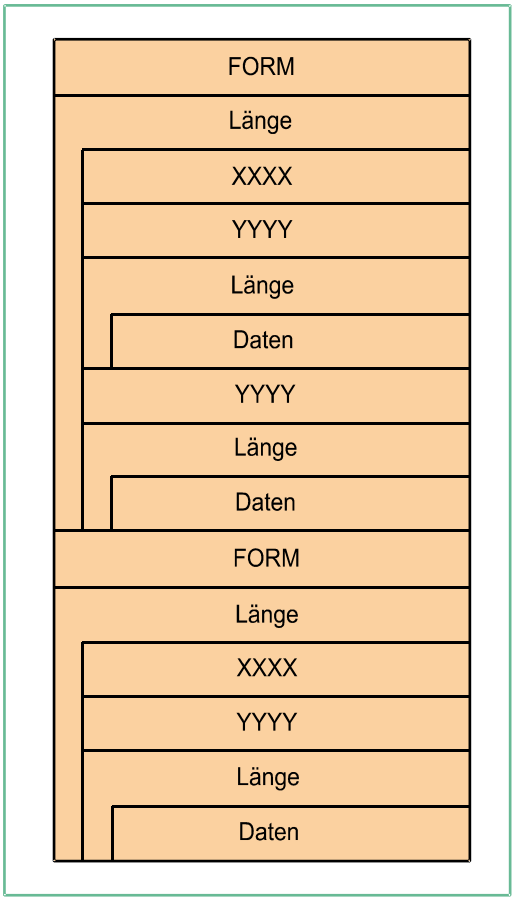

Bild 3. So könnte ein IFF mit zwei Forms und drei Chunks aussehen. In dem oberen könnte ein Bild gespeichert sein, das untere enthält vielleicht stimmungsvolle Musik.

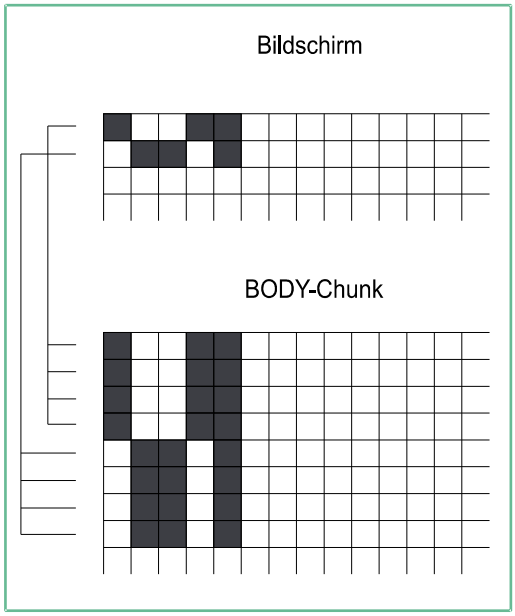

Bild 4. Die Bits der Pixel werden nach Wertigkeit sortiert im BODY-Chunk gespeichert. Erst kommen die niederwertigen Bits.

DPaint II Enhanced

Ausgerechnet „Electronic Arts" ist die Firma, die ihren IFF-Standard anscheinend am wenigsten ernst nimmt. Denn die PC-Version von DPaint, das „Deluxe Paint II Enhanced", weicht in einigen wichtigen Fällen von der IFF-Konvention ab. Das Programm verlangt zum Beispiel eine zwingende Reihenfolge von BMHD, CMAP und BODY. Sind diese Chunks anders sortiert, erkennt das Programm das File nicht mehr als IFF an. Mit den Auflösungen der VGA-Karte, bei denen 256 Farben gleichzeitig dargestellt werden, hat es sich Electronic Arts ebenfalls etwas zu einfach gemacht. Für diese Darstellungsarten, die 8 Bit pro Pixel benötigen, wurde bei DPaint eine neue Form eingeführt. Der heißt hier nicht mehr ILBM sondern „PBM". Das drückt aus, daß die Daten des BODY-Chunks anders als normal interpretiert werden. Im BODY-Chunk werden jetzt die Bits eines Pixel zusammengefaßt und hintereinanderweg gespeichert.

Der schwerwiegendste Fehler verbirgt sich hinter einer sonst recht praktischen Eigenschaft von DPaint. Das Programm speichert zu jedem Bild eine verkleinerte Version, die es dann vor dem Laden anzeigt. Diese Bilder werden in einem eigenen Chunk, genannt „TINY", gespeichert. Und genau dieser Chunk hat manchmal eine falsche Längenangabe. Meistens sind die Daten um ein Byte zu wenig berechnet, als sie sein sollten. Programme, die die Chunks anhand der Längenangaben nach verwertbaren Daten abklappern, kommen nach dem TINY-Chunk dann ins Stolpern. Da DPaint den BODY-Chunk als letztes speichert, können andere Programme diesen leicht übersehen.

Tabelle 1. Die wichtigsten Forms des IFF	
Bezeichnung	**Beschreibung**
8SVX	In diesem Form werden digitalisierte Geräusche und Töne gespeichert.
FTXT	Das Textformat des IFF. Es hat nur den CHRS-Chunk, in dem die Textdaten gespeichert werden.
ILBM	Das Bildformat, das in diesem Artikel beschrieben wird. Siehe auch Tabelle 2.
PBM	Das PBM-Form ist ebenfalls ein Grafik-Format, die Chunks haben die gleiche Bedeutung wie beim ILBM-Form.
SMUS	Der SMUS-Form ist für Musik vorgesehen. Hier werden neben den Noten auch der Autor und die Instrumente gespeichert.

Tabelle 2. Die wichtigsten Chunks des ILBM	
Bezeichnung	**Beschreibung**
BMHD	Der Kopf eines jeden ILBMs. Hier werden die Größe des Bildes, die Anzahl der Bits pro Pixel gespeichert und ob das Bild komprimiert wurde. Siehe auch Tabelle 3.
BODY	In diesem Chunk stehen die Bilddaten.
CCRT	In diesem Chunk wird ein spezieller Effekt gespeichert, mit dem man Farbanimationen auf dem Bildschirm bringt. Dabei wird eine bestimmte Anzahl Farben zyklisch vertauscht. Die Geschwindigkeit und die verwendeten Farben sind hier gespeichert.
CAMG	Ein spezieller Amiga-Chunk. Der Amiga kennt noch einige spezielle Bildschirmmodi, wie den 4096-Farbmodus HAM oder den 64-Farben-EHB-Modus. Bei diesen werden die Daten von der Video-Hardware des Amiga speziell interpretiert. Die Art der Verarbeitung wird in diesem Chunk gespeichert.
CMAP	Die beschriebene Farbpalette.
CRNG	Der CRNG-Chunk hat die gleiche Aufgabe wie der CCRT-Chunk. Die Daten werden allerdings anders interpretiert.
DEST	Mit dem DEST-Chunk werden Bitplanes des BODY-Chunks in andere Bitplanes des Grafikspeichers verlegt. Dadurch lassen sich Farben verfälschen.
GRAB	Wenn man mit einem Zeichenprogramm nur einen kleinen Ausschnitt des Bildes speichert, wird in diesem Chunk die Position des Cursors gespeichert, der den Ausschnitt festhält.
SPRT	Mit diesem Chunk wird mitgeteilt, daß es sich bei der Grafik um ein Sprite handelt. Der Wert in dem Chunk gibt an, wie weit das Sprite vor anderen Sprites auf dem Bildschirm dargestellt wird.
TINY	In diesem Chunk speichert die PC-Version von DPaint einen verkleinerten Bildausschnitt. Der Chunk gehorcht jedoch nicht immer den IFF-Konventionen (siehe Kasten).

Tabelle 3. Die Daten des BMHD-Chunks		
Name	**Länge**	**Beschreibung**
w	2 Byte	Bestimmt die Breite des Bildes in Pixel
h	2 Byte	Bestimmt die Höhe des Bildes in Pixel
Planes	1 Byte	Bestimmt die Anzahl der Bit pro Pixel
Masking	1 Byte	Wenn hier eine 1 steht, wird im BODY-Chunk eine zusätzliche Bitplane gespeichert, die als Maske mit Hintergrundfarbe über das Bild gelegt wird. Sonst steht hier eine Null. Mit einer 2 wird eine Farbe transparent geschaltet. Die Farbe wird in TransColor angegeben.
Compression	1 Byte	Wenn hier eine 1 steht, wird die beschriebene Kompressionsmethode angewandt (siehe Tabelle 4).
Leer	1 Byte	Immer Null
TransColor	2 Byte	Nummer der transparenten Farbe.
X-Verhältnis	1 Byte	Gibt an, wie breit ein Pixel im Verhältnis zur Höhe ist.
Y-Verhältnis	1 Byte	Gibt an, wie hoch ein Pixel im Verhältnis zur Breite ist.
PageH	2 Byte	Gibt an, wieviele Pixel breit die Auflösung war, aus der das Bild gespeichert wurde.
PageW	2 Byte	Gibt an, wieviele Pixel hoch die Auflösung war, aus der das Bild gespeichert wurde.

Tabelle 4. Die Steuerbytes bei komprimiertem BODY	
Wert	**Beschreibung**
0–127	Das Steuerbyte gibt die Anzahl der folgenden Bytes an, die in die Bitplane geschrieben werden. Die Anzahl ist eins höher als der Wert des Steuerbytes.
128	Keine Funktion. Es folgt das nächste Steuerbyte.
129–255	Es folgt ein Byte. Das Steuerbyte gibt in diesem Fall an, wie oft das Folgebyte in den Speicher kopiert wird. Um die Anzahl zu erhalten, zieht man von 257 den Wert des Steuerbytes ab. Bei einem Steuerbyte von 255 werden also zwei Byte kopiert.

Listing 1. Hiermit lädt man einen Chunk

```
int loadchunk(FILE *fp, const char *chunkname, char *daten)
{
  int r = TRUE;
  unsigned short load, count = 0;
  unsigned long len;

  if (searchchunk(fp, chunkname))
  {
    len = buffer[4] * 16777216L +
          buffer[5] * 65536L +
          buffer[6] * 256 +
          buffer[7];
    while (len > 0)
    {
      if (len > 32767)
        load = 32767;
      else
        load = len;
      if (fread(daten + count, 1, load, fp) != len)
      {
        r = FALSE;
        break;
      }
      len -= load;
      count += load;
    }
  }
  else
    r = FALSE;
  return(r);
}
```

Listing 2. Ein Chunk wird in ein IFF eingefügt

```
int insertchunk(FILE *oldfile, const char *chunkafter,
                const char *chunkname, char *chunkdata, long chunklen)
{
  FILE *newfile;
  int r = 1;
  int found = FALSE, insertdone = FALSE;
  unsigned long oldlen, newlen, formcount;
  int i, j, k;

  if (!(newfile = fopen("NEW.IFF", "wb")))
    r = FALSE;
  else
  {
    fseek(oldfile, 0L, SEEK_SET);
    if (fread(&buffer[0], 1, 12, oldf11e) ! = 12)
      r = FALSE;
    else
    {
      oldlen = buffer[4] * 16777216L +
               buffer[5] * 65536L +
               buffer[6] * 256 +
               buffer[7];
      formcount = 4;
      newlen = oldlen + chunklen + 8;
      buffer[4] = newlen >> 24;
      buffer[5] = (newlen >> 16) & 0xff;
      buffer[6] = (newlen >> 8) & 0xff;
      buffer[7] = newlen & 0xff;
      if (fwrite(&buffer[0], 1, 12, newfile) != 12)
        r = FALSE;
      else
         {
        while (formcount < oldlen)
        {
          if (fread(&buffer[0], 1, 8, oldfile) != 8)
          {
            r = FALSE;
            break;
          }
          else
```

```
        {
          if (fwrite(&buffer[0], 1, 8, newfile) != 8)
          {
            r = FALSE;
            break;
          }
          else
          {
            if (strcomp(&buffer[0], chunkafter))
              found = TRUE;
            newlen = buffer[4] * 16777216L +
                     buffer[5] * 65536L +
                     buffer[6] * 256 +
                     buffer[7];
            formcount += newlen + 8;
            while (newlen != 0)
            {
              if (newlen <= 1024)
                i = newlen;
              else
                i = 1024;
              if (fread(&buffer[0], 1, i, oldfile) != i)
              {
                r = FALSE;
                break;
              }
              else
              {
                if (fwrite(&buffer[0], 1, i, newfile) != i)
                {
                  r = FALSE;
                  break;
                }
                else
                  newlen -= i;
              }
            }
            if (r != 1)
              break;
            if ((found == TRUE) && (insertdone == FALSE))
            {
              newlen = chunklen;
              buffer[4] = newlen >> 24;
              buffer[5] = (newlen >> 16) & 0xff;
              buffer[6] = (newlen >> 8) & 0xff;
              buffer[7] = newlen & 0xff;
              for (i = 0; i < 4; i++)
                buffer[i] = *(chunkname + i);
              if (fwrite(&buffer[0], 1, 8, newfile) != 8)
                r = FALSE;
              else
                          {
                while (newlen != 0)
                {
                  if (newlen <= 1024)
                    i = newlen;
                  else
                    i = 1024;
                  j = 0;
                  for (k = 0; k < i; k++)
                    buffer[k] = *(chunkdata + j + k);
                  j += i;
                  if (fwrite(&buffer[0], 1, i, newfile) != i)
                  {
                    r = FALSE;
                    break;
                  }
                  else
                    newlen -= i;
                }
              }
                          insertdone = TRUE;
            }
          }
        }
      }
    }
```

```
        }
    }
    fclose(newfile);
    return(r);
}
```

Listing 3. Entfernt wird ein Chunk mit dieser Funktion

```
int deletechunk(FILE *oldfile, const char *chunkname)
{
  unsigned long chunklen, formlen, newlen;
  unsigned short i;
  int r = TRUE;
  FILE *newfile;

  if (searchchunk(oldfile, chunkname))
  {
    fseek(oldfile, 0L, SEEK_SET);
    chunklen = buffer[4] * 16777216L +
               buffer[5] * 65536L +
               buffer[6] * 256 +
               buffer[7];
    if (newfile = fopen("NEW.IFF", "wb"))
       {
      if (fread(&buffer[0], 1, 12, oldfile) == 12)
      {
        formlen = buffer[4] * 16777216L +
                  buffer[5] * 65536L +
                  buffer[6] * 256 +
                  buffer[7];
        newlen = formlen - chunklen - 8;
        buffer[4] = newlen >> 24;
        buffer[5] = (newlen >> 16) & 0xff;
        buffer[6] = (newlen >> 8) & 0xff;
        buffer[7] = newlen & 0xff;
        if (fwrite(&buffer[0], 1, 12, newfile) == 12)
        {
          while (fread(&buffer[0], 1, 8, oldfile) == 8)
          {
            chunklen = buffer[4] * 16777216L +
                       buffer[5] * 65536L +
                       buffer[6] * 256 +
                       buffer[7];
            if (strcomp(&buffer[0], chunkname))
              fseek(oldfile, chunklen, SEEK_CUR);
            else
            {
              if (fwrite(&buffer[0], 1, 8, newfile) == 8)
                     {
                while (chunklen > 0)
                {
                  if (chunklen > 1024)
                    i = 1024;
                  else
                    i = chunklen;
                  if (fread(&buffer[0], 1, i, oldfile) == i)
                  {
                    if (fwrite(&buffer[0], 1, i, newfile) != i)
                    {
                      r = FALSE;
                      break;
                    }
                  }
                  else
                  {
                    r = FALSE;
                    break;
                  }
                  chunklen -= i;
                }
                else
                {
                  r = FALSE;
                  break;
                }
                       }
            }
          }
          }
        }
        else
          r = FALSE;
      }
      else
        r = FALSE;
```

```
    }
       else
      r = FALSE;
  }
  else
    r = FALSE;
  return(r);
}
```

Listing 4. Prüft ein File auf IFF

```
int checkiff(FILE *fp, const char *formname)
{
  int r = TRUE;
  if (fread(&buffer[0], 1, 12, fp) != 12)
    r = FALSE;
  else
    if (strcomp(&buffer[0], "FORM") == FALSE)
      r = FALSE;
    else
      if (strcomp(&buffer[8], formname) == FALSE)
        r = FALSE;
  return(r);
}
```

Listing 5. Entpackt den BODY-Chunk einer ILBM

```
int unpackbody(FILE *fp, unsigned char *dest, unsigned short x,
               unsigned short y, unsigned short planes)
{
  int r = TRUE;
  unsigned short column, iffplanes, byte;
  unsigned h, i, j, k, l;

  byte = x / 8;
  if (!(x % 8))
    byte++;
  if (searchchunk(fp, "BODY"))
    /* Wir gehen in diesem Programmabschnitt
       davon aus, daß die Daten grundsätzlich
       gepackt sind. Wir zählen hier nacheinander
       alle Zeilen einer Plane durch. */

    for (column = 0; column < y; column++)
    {
      for (iffplanes = 0; iffplanes < planes; iffplanes++)
      {
        h = iffplanes * byte + column * iffplanes * byte;
        i = 0;
        do
        {
          if (fread(dest + h + i, 1, 1, fp) != 1)
          {
            r = FALSE;
            break;
          }
          j = *(dest + h + i);

          /* Es folgen ungepackte Byte. */

          if (J < 128)
          {
            k = j + 1;
            if (fread(dest + h + i, 1, k, fp) != k)
            {
              r = FALSE;
              break;
            }
            i += k;
          }

          /* Es folgt ein gepacktes Byte. */

          if (j > 128)
          {
            k = 257 - j;
            if (fread(dest + h + i, 1, 1, fp) != 1)
            {
              r = FALSE;
              break;
            }
            for (l = i; l < k + i; l++)
              *(dest + h + l) = *(dest + h + i);
            i += k;
          }
        } while (i < byte);
        if (!r)
```

```
          break;
      }
      if (!r)
        break;
    }
  return(r);
}
```

Listing 6. Die beiden Funktionen werden von den anderen Funktionen benötigt

```
int searchchunk(FILE *fp, const char *string)
{
  int r = TRUE;
  unsigned long count;

  fseek(fp, 12, SEEK_SET);
  do
  {
    if (fread(&buffer[0], 1, 8, fp) ! = 8)
      r = FALSE;
    else
      if (strcomp(&buffer[0], string) == FALSE)
      {
        count = buffer[4] * 16777216L +
                buffer[5] * 65536L +
                buffer[6] * 256 +
                buffer[7];
        if (fseek(fp, count, SEEK_CUR) != 0)
          r = FALSE;
      }
      else
        break;
  } while (r == TRUE);
  return(r);
}

int strcomp(const unsigned char *string1, const char *string2)
{
  int i, r = TRUE;
  for (i = 0; *(string2 + i) != NULL; i++)
    if (*(string1 + i) != *(string2 + i))
    {
      r = FALSE;
      break;
    }
  return(r);
}
```

Listing 7. Ist die Header-Datei für die anderen Funktionen

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

unsigned char buffer[12];

int loadchunk(FILE *fp, const char *chunkname, char *daten);
int insertchunk(FILE *oldfile, const char *chunkafter, const char *chunkname, char
*chunkdata, long chunklen);
int deletechunk(FILE *oldfile, const char *chunkname);
int checkiff(FILE *fp, const char *formname);
int unpackbody(FILE *fp, unsigned char *dest, unsigned short x, unsigned short y, unsigned
short planes);
int searchchunk(FILE *fp, const char *string);
int strcomp(const unsigned char *string1, const char *string2);
```