IFF-ANIM-Format

...

COLLABORATORS			
	TITLE : IFF-ANIM-Format		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	...	May 28, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

IFF-ANIM-Format

1.1 main

A N I M
An IFF Format For CEL Animations

Revision date: June 1997

1.2 authors

```
   A N I M - An IFF Format For CEL Animations
   ------------------------------------------
```

prepared by:
    SPARTA Inc.
    23041 de la Carlota
    Laguna Hills, Calif 92653
    (714) 768-8161
    contact: Gary Bonham

also by:
    Aegis Development Co.
    2115 Pico Blvd.
    Santa Monica, Calif 90405
    213) 392-9972

Anim6 Appendix (august 91) by
    Cryogenic Software
    13045 SouthEast Stark St
    Suite 144
    Portland, OR 97233-1557
    Contact: William J. Coldwell

Anim7 Appendix (july 92) by:
    Wolfgang Hofer
    A-2722 Winzendorf
    Wr. Neustaedterstr. 140

Anim8 Appendix (january 92) by:
    Joe Porkka

Anim-with-Sound-Appendix (june 1997) by:
    Virtual Worlds Productions
    Michael Pfeiffer

This document was assembled, edited and converted in AmigaGuide-Format by
Michael Pfeiffer using differend sources.

1.3 introduction

```
Introduction
------------
```

The ANIM IFF format was developed at Sparta originally for the production of animated video sequences on the Amiga computer. The intent was to be able to store, and play back, sequences of frames and to minimize both the storage space on disk (through compression) and playback time (through efficient de-compression algorithms). It was desired to maintain maximum compatibility with existing IFF formats and to be able to display the initial frame as a normal still IFF picture.

Several compression schemes have been introduced in the ANIM format. Most of these are strictly of historical interest as the only one currently being placed in new code is the vertical run length encoded byte encoding developed by Jim Kent.

1.4 overview

```
   ANIM Format Overview
   --------------------
```

The general philosophy of ANIMs is to present the initial frame as a normal, run-length-encoded, IFF picture. Subsequent frames are then described by listing only their differences from a previous frame. Normally, the "previous" frame is two frames back as that is the frame remaining in the hidden screen buffer when double-buffering is used. To better understand this, suppose one has two screens, called A and B, and the ability to instantly switch the display from one to the other. The normal playback mode is to load the initial frame into A and duplicate it into B. Then frame A is displayed on the screen. Then the differences for frame 2 are used to alter screen B and it is displayed. Then the differences for frame 3 are used to alter screen A and it is displayed, and so on. Note that frame 2 is stored as differences from frame 1, but all other frames are stored as differences from two frames back.

ANIM is an IFF FORM and its basic format is as follows (this assumes the reader has a basic understanding of IFF format files):

```
   FORM ANIM
   .  FORM ILBM         first frame
   .  .  BMHD              normal type IFF data
   .  .  ANHD              optional animation header chunk for timing of
                           1st frame.
   .  .  CMAP
   .  .  BODY
   .  FORM ILBM         frame 2
   .  .  ANHD              animation header chunk
   .  .  DLTA              delta mode data
   .  FORM ILBM         frame 3
   .  .  ANHD
   .  .  DLTA
```

The initial FORM ILBM can contain all the normal ILBM chunks, such as CRNG, etc. The BODY will normally be a standard run-length-encoded data chunk (but may be any other legal compression mode as indicated by the BMHD). If desired, an ANHD chunk can appear here to provide timing data for the first frame. If it is here, the operation field should be =0.

The subsequent FORMs ILBM contain an ANHD, instead of a BMHD, which duplicates some of BMHD and has additional parameters pertaining to the animation frame. The DLTA chunk contains the data for the delta compression modes. If the older XOR compression mode is used, then a BODY chunk will be here. In addition, other chunks may be placed in each of these as deemed necessary (and as code is placed in player programs to utilize them). A good example would be CMAP chunks to alter the color palette. A basic assumption in ANIMs is that the size of the bitmap, and the display mode (e.g. HAM) will not change through the animation. Take care when playing an ANIM that if a CMAP occurs with a frame, then the change must be applied to both buffers.

Note that the DLTA chunks are not interleaved bitmap representations, thus the use of the ILBM form is inappropriate for these frames. However, this inconsistency was not noted until there were a number of commercial products either released or close to release which generated/played this format. Therefore, this is probably an inconsistency which will have to stay with us.

1.5 recording

```
Recording ANIMs
---------------
```

To record an ANIM will require three bitmaps - one for creation of the next frame, and two more for a "history" of the previous two frames for performing the compression calculations (e.g. the delta mode calculations).

There are five frame-to-frame compression methods currently defined. The first three are mainly for historical interest. The product Aegis VideoScape 3D utilizes the third method in version 1.0, but switched to method 5 on 2.0. This is the only instance known of a commercial product generating ANIMs of any of the first three methods. The fourth method is a general short or long word compression scheme which has several options including whether the compression is horizontal or vertical, and whether or not it is XOR format. This offers a choice to the user for the optimization of file size and/or playback speed. The fifth method is the byte vertical run length encoding as designed by Jim Kent. Do not confuse this with Jim’s RIFF file format which is different than ANIM. Here we utilized his compression/decompression routines within the ANIM file structure.

The following pages give a general outline of each of the methods of compression currently included in this spec:

XOR mode
Long Delta mode
Short Delta mode
General Delta mode
Byte Vertical Compression

1.6 xor

```
XOR mode
--------
```

This mode is the original and is included here for historical interest. In general, the delta modes are far superior. The creation of XOR mode is quite simple. One simply performs an exclusive-or (XOR) between all corresponding bytes of the new frame and two frames back. This results in a new bitmap with 0 bits wherever the two frames were identical, and 1 bits where they are different. Then this new bitmap is saved using run-length-encoding. A major obstacle of this mode is in the time consumed in performing the XOR upon reconstructing the image.

1.7 longdelta

```
Long Delta mode
---------------
```

This mode stores the actual new frame long-words which are different, along with the offset in the bitmap. The exact format is shown and discussed below.

Each plane is handled separately, with no data being saved if no changes take place in a given plane. Strings of 2 or more long-words in a row which change can be run together so offsets do not have to be saved for each one.

Constructing this data chunk usually consists of having a buffer to hold the data, and calculating the data as one compares the new frame, long-word by long-word, with two frames back.

1.8 shortdelta

```
Short Delta mode
----------------
```

This mode is identical to the Long Delta mode except that short-words are saved instead of long-words. In most instances, this mode results in a smaller DLTA chunk. The Long Delta mode is mainly of interest in improving the playback speed when used on a 32-bit 68020 Turbo Amiga.

1.9 generaldelta

```
   General Delta mode
   ------------------
```

The above two delta compression modes were hastily put together. This mode was an attempt to provide a well-thought-out delta compression scheme. Options provide for both short and long word compression, either vertical or horizontal compression, XOR mode (which permits reverse playback), etc. About the time this was being finalized, the fifth mode, below, was developed by Jim Kent. In practice the short-vertical-run-length-encoded deltas in this mode play back faster than the fifth mode (which is in essence a byte-vertical-run-length-encoded delta mode) but does not compress as well - especially for very noisy data such as digitized images. In most cases, playback speed not being terrifically slower, the better compression (sometimes 2x) is preferable due to limited storage media in most machines.

Details on this method are contained below.

1.10 bytevertical

```
   Byte Vertical Compression
   -------------------------
```

This method does not offer the many options that method 4 offers, but is very successful at producing decent compression even for very noisy data such as digitized images. The method was devised by Jim Kent and is utilized in his RIFF file format which is different than the ANIM format. The description of this method in this document is taken from Jim's writings. Further, he has released both compression and decompression code to public domain.

1.11 playing

```
   Playing ANIMs
   -------------
```

Playback of ANIMs will usually require two buffers, as mentioned above, and double-buffering between them. The frame data from the ANIM file is used to modify the hidden frame to the next frame to be shown. When using the XOR mode, the usual run-length-decoding routine can be easily modified to do the exclusive-or operation required. Note that runs of zero bytes, which will be very common, can be ignored, as an exclusive or of any byte value to a byte of zero will not alter the original byte value.

The general procedure, for all compression techniques, is to first decode the initial ILBM picture into the hidden buffer and doublebuffer it into view. Then this picture is copied to the other (now hidden) buffer. At this point each frame is displayed with the same procedure. The next frame is formed in the hidden buffer by applying the DLTA data (or the XOR data from

the BODY chunk in the case of the first XOR method) and the new frame is double-buffered into view. This process continues to the end of the file.

A master colormap should be kept for the entire ANIM which would be initially set from the CMAP chunk in the initial ILBM. This colormap should be used for each frame. If a CMAP chunk appears in one of the frames, then this master colormap is updated and the new colormap applies to all frames until the occurrance of another CMAP chunk.

Looping ANIMs may be constructed by simply making the last two frames identical to the first two. Since the first two frames are special cases (the first being a normal ILBM and the second being a delta from the first) one can continually loop the anim by repeating from frame three. In this case the delta for creating frame three will modify the next to the last frame which is in the hidden buffer (which is identical to the first frame), and the delta for creating frame four will modify the last frame which is identical to the second frame.

Multi-File ANIMs are also supported so long as the first two frames of a subsequent file are identical to the last two frames of the preceeding file. Upon reading subsequent files, the ILBMs for the first two frames are simply ignored, and the remaining frames are simply appended to the preceeding frames. This permits splitting ANIMs across multiple floppies and also permits playing each section independently and/or editing it independent of the rest of the ANIM.

Timing of ANIM playback is easily achieved using the vertical blank interrupt of the Amiga. There is an example of setting up such a timer in the ROM Kernel Manual. Be sure to remember the timer value when a frame is flipped up, so the next frame can be flipped up relative to that time. This will make the playback independent of how long it takes to decompress a frame (so long as there is enough time between frames to accomplish this decompression).

1.12 chunkformats

```
Chunk Formats
-------------

 ANHD-Chunk
 DLTA-Chunk
    DLTA Format for methods 2 & 3
    DLTA Format for method 4
    DLTA Format for method 5
```

1.13 anhd

```
ANHD Chunk
----------
```

The ANHD chunk consists of the following data structure:

```
UBYTE operation  The compression method:
                    =0 set directly (normal ILBM BODY),
                    =1 XOR ILBM mode,
                    =2 Long Delta mode,
                    =3 Short Delta mode,
                    =4 Generalized short/long Delta mode,
                    =5 Byte Vertical Delta mode  Double-buffered
                    =6 Byte Vertical Delta mode, Quad-buffered
                    =7 short/long Vertical Delta mode
                    =8 short/long Vertical Delta mode
                    =74 (ascii 'J') reserved for Eric Graham's
                        compression technique (details to be
                        released later).

UBYTE mask       (XOR mode only - plane mask where each bit is set =1
                  if there is data and =0 if not.)
UWORD w,h        (XOR mode only - width and height of the area
                  represented by the BODY to eliminate unnecessary un-
                  changed data)
WORD  x,y        (XOR mode only - position of rectangular area
                  represented by the BODY)
ULONG abstime    (currently unused - timing for a frame relative to the
                  time the first frame was displayed - in jiffies
                  (1/60 sec))
ULONG reltime    (timing for frame relative to time previous frame was
                  displayed - in jiffies (1/60 sec))
UBYTE interleave (used in ANIM6 method - indicates how may frames back
                  this data is to modify.  =0 defaults to indicate two
                  frames back (for double buffering). =n indicates n
                  frames back. The main intent here is to allow values
                  of =1 for special applications where frame data would
                  modify the immediately previous frame)
UBYTE pad0        Pad byte, not used at present.
ULONG bits        32 option bits used by options=4 and 5. At present
                  only 6 are identified, but the rest are set =0 so they
                  can be used to implement future ideas. These are defined
                  for option 4 only at this point. It is recommended that
                  all bits be set =0 for option 5 and that any bit
                  settings used in the future (such as for XOR mode) be
                  compatible with the option 4 bit settings. Player code
                  should check undefined bits in options 4 and 5 to assure
                  they are zero.

                  The six bits for current use are:

                     bit #              set =0               set =1
                     ================================================
                     0              short data           long data
                     1                 set                  XOR
                     2             separate info        one info list
                                   for each plane       for all planes
                     3               not RLC        RLC (run length coded)
                     4              horizontal           vertical
                     5           short info offsets   long info offsets

UBYTE pad[16]     This is a pad for future use for future compression
```

modes.

1.14 dlta

```
DLTA Chunk
----------
```

This chunk is the basic data chunk used to hold delta compression data. The format of the data will be dependent upon the exact compression format selected.

```
DLTA Format for methods 2 & 3
DLTA Format for method 4
DLTA Format for method 5
```

1.15 method23

```
Format for methods 2 & 3
------------------------
```

This chunk is a basic data chunk used to hold the delta compression data. The minimum size of this chunk is 32 bytes as the first 8 long-words are byte pointers into the chunk for the data for each of up to 8 bitplanes. The pointer for the plane data starting immediately following these 8 pointers will have a value of 32 as the data starts in the 33-rd byte of the chunk (index value of 32 due to zero-base indexing).

The data for a given plane consists of groups of data words. In Long Delta mode, these groups consist of both short and long words - short words for offsets and numbers, and long words for the actual data. In Short Delta mode, the groups are identical except data words are also shorts so all data is short words. Each group consists of a starting word which is an offset. If the offset is positive then it indicates the increment in long or short words (whichever is appropriate) through the bitplane. In other words, if you were reconstructing the plane, you would start a pointer (to shorts or longs depending on the mode) to point to the first word of the bitplane. Then the offset would be added to it and the following data word would be placed at that position. Then the next offset would be added to the pointer and the following data word would be placed at that position. And so on... The data terminates with an offset equal to 0xFFFF.

A second interpretation is given if the offset is negative. In that case, the absolute value is the offset+2. Then the following short-word indicates the number of data words that follow. Following that is the indicated number of contiguous data words (longs or shorts depending on mode) which are to be placed in contiguous locations of the bitplane.

If there are no changed words in a given plane, then the pointer in the first 32 bytes of the chunk is =0.

1.16 method4

```
   Format for method 4
   -------------------
```

The DLTA chunk is modified slightly to have 16 long pointers at the start. The first 8 are as before - pointers to the start of the data for each of the bitplanes (up to a max of 8 planes). The next 8 are pointers to the start of the offset/numbers data list. If there is only one list of offset/numbers for all planes, then the pointer to that list is repeated in all positions so the playback code need not even be aware of it. In fact, one could get fancy and have some bitplanes share lists while others have different lists, or no lists (the problems in these schemes lie in the generation, not in the playback).

The best way to show the use of this format is in a sample playback routine.

```
SetDLTAshort(bm,deltaword)
struct BitMap *bm;
WORD *deltaword;
{
   int i;
   LONG *deltadata;
   WORD *ptr,*planeptr;
   register int s,size,nw;
   register WORD *data,*dest;

   deltadata = (LONG *)deltaword;
   nw = bm->BytesPerRow >>1;

   for (i=0;i<bm->Depth;i++) {
      planeptr = (WORD *)(bm->Planes[i]);
      data = deltaword + deltadata[i];
      ptr  = deltaword + deltadata[i+8];
      while (*ptr != 0xFFFF) {
         dest = planeptr + *ptr++;
         size = *ptr++;
         if (size < 0) {
            for (s=size;s<0;s++) {
               *dest = *data;
               dest += nw;
            }
            data++;
         }
         else {
            for (s=0;s<size;s++) {
               *dest = *data++;
               dest += nw;
            }
         }
      }
   }
   return(0);
```

```
            }
```

The above routine is for short word vertical compression with run length compression. The most efficient way to support the various options is to replicate this routine and make alterations for, say, long word or XOR. The variable nw indicates the number of words to skip to go down the vertical column. This one routine could easily handle horizontal compression by simply setting nw=1. For ultimate playback speed, the core, at least, of this routine should be coded in assembly language.

1.17 method5

```
Format for method 5
-------------------
```

In this method the same 16 pointers are used as in option 4. The first 8 are pointers to the data for up to 8 planes. The second set of 8 are not used but were retained for several reasons. First to be somewhat compatible with code for option 4 (although this has not proven to be of any benefit) and second, to allow extending the format for more bitplanes (code has been written for up to 12 planes).

Compression/decompression is performed on a plane-by-plane basis. For each plane, compression can be handled by the skip.c code (provided Public Domain by Jim Kent) and decompression can be handled by unvscomp.asm (also provided Public Domain by Jim Kent).

Compression/decompression is performed on a plane-by-plane basis. The following description of the method is taken directly from Jim Kent's code with minor re-wording. Please refer to Jim's code (skip.c and unvscomp.asm) for more details:

Each column of the bitplane is compressed separately. A 320x200 bitplane would have 40 columns of 200 bytes each. Each column starts with an op-count followed by a number of ops. If the op-count is zero, that's ok, it just means there's no change in this column from the last frame. The ops are of three classes, and followed by a varying amount of data depending on which class:

1. Skip ops - this is a byte with the hi bit clear that says how many rows to move the "dest" pointer forward, ie to skip. It is non-zero.
2. Uniq ops - this is a byte with the hi bit set. The hi bit is masked down and the remainder is a count of the number of bytes of data to copy literally. It's of course followed by the data to copy.
3. Same ops - this is a 0 byte followed by a count byte, followed by a byte value to repeat count times.

Do bear in mind that the data is compressed vertically rather than horizontally, so to get to the next byte in the destination we add the number of bytes per row instead of one!

1.18 anim6

```
   Appendix for Anim6 Formats (by William J Coldwell)
   ---------------------------------------------------

 Introduction
 OpCode 6 Additions to OpCode 5
 Playing OpCode 6 ANIMs


Prepared by:
   Cryogenic Software
   13045 SouthEast Stark St
   Suite 144
   Portland, OR 97233-1557
   Contact:  William J. Coldwell

   Voice:    (503) 254-8147 (11a-4p PDT/PST)
   Data:     (503) 257-4823 (EMail to SYSOP)
   Portal:   Cryogenic
   UUCP:     uunet!m2xenix!percy!cryo!billc
   Internet: billc@cryo.rain.com
```

1.19 anim6_introduction

```
   Introduction
   ------------

In 1989, we added support into one of our commercial products to support
the Haitex X-Specs glasses. This documentation will not go into a detailed
description of this product. Contact Haitex for more information concerning
the hardware:
                           Haitex Resources, Inc.
                           Post Office Box 20609
                           Charleston, SC 29413
                             Voice: (803) 881-7518
                               Fax: (803) 881-7522
                           Contact: Shawn Glisson

We found that there was not a supported way to display stereo animations
using the current IFF ANIM OpCode 5 specification.

Cryogenic supported OpCode 6 as an internal format in our commercial
programs (see below) and provided Public Domain players. It is our
intention at this time, to release this format to other developers wishing
to support stereo animations using this OpCode.

When we first started this project, the current Amiga machines had a 512K
of CHIP RAM maximum. This caused some memory problems with some of the
higher resolution stereo animations, since the Quad Buffers were in CHIP RAM
for our players. It was our intention to attempt to do some memory magic to
require only two of the four bitmaps to be in CHIP RAM at one time. It was
our feeling that this would have caused the animations to slow down, due to
data swapping that may or may not have needed to be done.  By the end of 1989,
all development had stopped on OpCode 6. This left all buffers in CHIP, and
```

the format has remained the same since then.

1.20 anim6_additions

```
   OpCode 6 Additions to OpCode 5
   ------------------------------
```

The format is exactly the same as OpCode 5 5 but is QUAD buffered instead of DOUBLE buffered. This allows the player to show 2 screens at one time for the X-Specs Glasses. Each picture MUST be viewed for 1/60th of a second, therefore to see a 3-D Picture the viewer can only play ANIMs at 30 frames per second (2 pictures = 1 frame).

The IFF file is stored exactly the same except that instead of having each DLTA (delta) modify bitmap two frames back, it modifies the bitmap four frames back.

Example:

```
 ------------------------
 |                      |
 |   BMHD               |
 |                      |
 ------------------------
 |   DLTA  (1)          |
  ------------------------
 |   DLTA  (2)          |
 ------------------------
 |   DLTA  (3)          |
 ------------------------
 |   DLTA  (4)          |
 ------------------------
 |   DLTA  (5)          |
 ------------------------
 |   DLTA  (6)          |
 ------------------------
            .
            .
            .
 ------------------------
 |   DLTA  (x)          |
 ------------------------
```

1.21 anim6_playing

```
   Playing OpCode 6 ANIMs
   ----------------------
```

Four bitmaps are allocated. Bitmaps 1 and 3 are the left views, and bitmaps 2 and 4 are the right.

```
The First bitmap is gets its image from the bitmap in the file (BMHD). The
Second bitmap is a copy of the first with DLTA (1) performed on it. The
Third Bitmap is a copy of the first with DLTA (2) performed on it. The
Fourth Bitmap is a copy of the first with DLTA (3) performed on it.

We now have the first two 3-D Pictures: One in bitmaps 1 and 2 and the other
in bitmaps 3 and 4

DLTA (6) is used to create the third left view from bitmap 1.
DLTA (7) is used to create the third right view from bitmap 2.

DLTA (8) is used to create the forth left view from bitmap 3.
DLTA (9) is used to create the forth right view from bitmap 4.

NOTE:  This technique requires 4 Loop frames at the end to perform looping.
```

1.22 anim7

```
   Appendix for Anim7 Formats (by Wolfgang Hofer)
   ----------------------------------------------

Anim method 7 is designed for maximum playback speed and acceptable packing
rates (packing usually not as good as method 5, but more efficient than
methodes 1 -- 4)

 Chunk Sequence
 Chunk Formats
    ANHD-Chunk
    DLTA-Chunk


Method 7 is not in the IFF specification today but supported by the Public
Domain Programs AAP/AAC.
```

1.23 anim7_chunksequence

```
   Chunk Sequence
   --------------

Method 7 Anims should use the same Chunk Sequence as methods 1..5.
Alternativley the first frame may have a DLTA chunk instead of the BODY
chunk. In that case the DLTA is the difference to a 'black frame'. A player
has to clear all bitplanes of the first bitmap to zero, and then call his
DLTA unpack routines for this frame.

   FORM ANIM
   .  FORM ILBM         first frame
   .  .  BMHD              normal type IFF data
   .  .  ANHD              optional animation header chunk for timing of
```

```
                          1st frame.
   .  .  CMAP
   .  .  BODY/DLTA         full picture or difference to 'black frame'
   .  FORM ILBM         frame 2
   .  .  ANHD              animation header chunk
   .  .  DLTA              delta mode data
   .  .  [CMAP]
   .  FORM ILBM         frame 3
   .  .  ANHD
   .  .  DLTA
   .  .  [CMAP]
```

The initial FORM ILBM can contain all the normal ILBM chunks, such as CRNG, etc. The BODY will normally be a standard run-length-encoded data chunk (but may be any other legal compression mode as indicated by the BMHD). If desired, an ANHD chunk can appear here to provide timing data for the first frame. If it is here, the operation field should be =0.

If the initial FORM ILBM uses a DLTA chunk, the ANHD chunk must appear, and the operation field must be set to the according anim method.

1.24 anim7_chunkformats

```
   Chunk Formats
   -------------

 ANHD-Chunk
 DLTA-Chunk
```

1.25 anim7_anhd

```
   ANHD Chunk for method 7
   -----------------------
```

The ANHD chunk consists of the following data structure:

```
   UBYTE operation  The compression method: =7 short/long Vertical Delta mode
   UBYTE mask       unused
   UWORD w,h        unused
   WORD  x,y        unused
   ULONG abstime    unused
   ULONG reltime    (timing for frame relative to time previous frame was
                    displayed - in jiffies (1/60 sec))
   UBYTE interleave =0 (see ANHD description)
   UBYTE pad0       unused
   ULONG bits       32 option bits used by methode=4 and 5, methode 7 uses
                    only bit #0

                       bit #              set =0               set =1
```

```
                    ================================================
                     0               short data          long data
```

```
   UBYTE pad[16]    unused
```

1.26 anim7_dlta

```
   DLTA Chunk Format for method 7
   ------------------------------
```

The DLTA Chunks of method7 consists of

- 8 pointers to opcode lists
- 8 pointers to data lists
- data lists (long/short)
- opcode lists (bytes)

In this method the DLTA Chunk begins with 16 pointers. The first 8 longwords are pointers to the opcode lists for up to 8 planes. The second set of 8 longwords are pointers to the correspondig data lists. If there are less than 8 Planes all unused pointers are set to zero.

Compression/decompression is performed on a plane-by-plane basis. The following description of the method is similar to Jim Kent's methode 5, except that data is stored in a seperated datalist (long or short, depending on bit#0 of the ANHD bits) and doesn't follow immediate after the opcode.

In methode 7 the bitplane is splitted into vertical columns. Each column of the bitplane is compressed separately. A 320x200 bitplane would have 20 columns of 200 short datas each (or 10 columns of 200 long datas). Each column starts with an op-count followed by a number of ops. If the op-count is zero, that's ok, it just means there's no change in this column from the last frame. The ops are of three classes. The ops refer to a varying amount of data (to fetch from the corresponding datalist) depending on which class:

1. Skip ops - this is a byte with the hi bit clear that says how many rows to move the "dest" pointer forward, ie to skip. It is non-zero. Skip ops have no corresponding data-items in the datalist.
2. Uniq ops - this is a byte with the hi bit set. The hi bit is masked down and the remainder is a count of the number of data to copy literally from the datalist to the "dest" pointer column. (Each data item to the next destination row) Data items may be long or short organized.
3. Same ops - this is a 0 byte followed by a count byte. The count byte says how many rows of the current column are to be set to the same data-item. the data-item (long or short) is fetched from the datalist.

Do bear in mind that the data is compressed vertically rather than horizontally, so to get to the next address in the destination we have to add the number of bytes per row instead of 2 (or 4)!

1.27 anim8

```
   Appendix for Anim8 Formats (by Joe Porkka)
   ------------------------------------------
```

Anim method 8 is designed for maximum playback speed and acceptable packing rates (packing usually not as good as method 5, but more efficient than methodes 1 -- 4). In addition, it is easier to convert existing Anim5 code to support Anim8 than Anim7.

1.28 anim8_chunksequence

```
   Chunk Sequence:
   ---------------
```

Method 8 Anims should use the same Chunk Sequence as methods 1..5. Alternativley the first frame may have a DLTA chunk instead of the BODY chunk. In that case the DLTA is the difference to a 'black frame'. A player has to clear all bitplanes of the first bitmap to zero, and then call his DLTA unpack routines for this frame. The same rules about copying the first frame into both frame buffers still applies in this case.

```
   FORM ANIM
   .  FORM ILBM         first frame
   .  .  BMHD              normal type IFF data
   .  .  ANHD              optional animation header chunk for timing of
                           1st frame.
   .  .  CMAP
   .  .  BODY/DLTA         full picture or difference to 'black frame'
   .  FORM ILBM         frame 2
   .  .  ANHD              animation header chunk
   .  .  DLTA              delta mode data
   .  .  [CMAP]
   .  FORM ILBM         frame 3
   .  .  ANHD
   .  .  DLTA
   .  .  [CMAP]
```

The initial FORM ILBM can contain all the normal ILBM chunks, such as CRNG, etc. The BODY will normally be a standard run-length-encoded data chunk (but may be any other legal compression mode as indicated by the BMHD). If desired, an ANHD chunk can appear here to provide timing data for the first frame. If it is here, the operation field should be =0.

If the initial FORM ILBM uses a DLTA chunk, the ANHD chunk must appear, and the operation field must be set to the according anim method.

```
Each of the frames from frame 2 on up may use an anhd->operation of 0, 5 or
8. Note that only for the first frame in the file do you copy the image data
into two buffers, not every time you get an ANHD->operation==0.
```

1.29 anim8_chunkformats

```
   Chunk Formats
   -------------

    ANHD-Chunk
    DLTA-Chunk
```

1.30 anim8_anhd

```
   ANHD Chunk for method 8
   -----------------------

The ANHD chunk consists of the following data structure:

   UBYTE operation  The compression method: =8 short/long Vertical Delta mode
   UBYTE mask       unused
   UWORD w,h        unused
   WORD  x,y        unused
   ULONG abstime    unused
   ULONG reltime    (timing for frame relative to time previous frame was
                     displayed - in jiffies (1/60 sec))
   UBYTE interleave =0 (see ANHD description)
   UBYTE pad0       unused
   ULONG bits       32 option bits used by methode=4 and 5. methode 8 uses
                    only bit #0

                       bit #              set =0               set =1
                       ================================================
                        0               short data           long data

   UBYTE pad[16]    unused
```

1.31 anim8_dlta

```
   DLTA Chunk Format for method 8
   ------------------------------

The DLTA Chunks of method8 consists of

- 16 pointers, same as in method 5

In this method the DLTA Chunk begins with 16 pointers. The first 8 longwords
```

are pointers to the opcode lists for up to 8 planes. The second set of 8 longwords are unused. If there are less than 8 Planes all unused pointers are set to zero.

Compression/decompression is performed on a plane-by-plane basis. The following description of the method is similar to Jim Kent's methode 5, except that data is either in WORDs or LONGS, depending on bit 0 of the ANHD bits:

In methode 8 the bitplane is split into vertical columns. Each column of the bitplane is compressed separately. A 320x200 bitplane would have 20 columns of 200 short datas each (or 10 columns of 200 long datas). Each column of the bitplane is compressed separately. A 320x200 bitplane would have 20 (WORD) or 10 (LONG)columns of 200 bytes each. Each column starts with an op-count followed by a number of ops. If the op-count is zero, that's ok, it just means there's no change in this column from the last frame. The ops are of three classes, and followed by a varying amount of data depending on which class:

1. Skip ops - this is a word or long with the hi bit clear that says how many rows to move the "dest" pointer forward, ie to skip. It is non-zero. Note that the range of values is much larger for word and long data, 0x7fff and 0x7fffffff.
2. Uniq ops - this is a word or long with the hi bit set. The hi bit is masked down and the remainder is a count of the number of bytes of data to copy literally. It's of course followed by the data to copy. Note that the range of values is much larger for word and long data, 0x7fff and 0x7fffffff.
3. Same ops - this is a 0 word or long followed by a count word or long, followed by a word or long value to repeat count times. Note that the range of values is much larger for word and long data, 0xffff and 0xffffffff.

Do bear in mind that the data is compressed vertically rather than horizontally, so to get to the next word or long in the destination we add the number of bytes per row instead of one!

There is a slight complication in the case of long data. Normally an Amiga BitMap is and even number of 16bit WORDs wide, so it is possible to have an image which is not an even number or LONGs wide. For example, an image which is 336 pixels wide is 42 bytes wide, 21 words wide, and 10.5 longs wide. In the case that the data is not an even number of longs wide, and the data is to be long compressed, then the last column of data is to be word compressed instead. So, that 336 pixel wide image would be compress as 10 long columns and 1 word column.

1.32 animsound

Appendix for Anims with Sound (by Michael Pfeiffer)

(I have to excuse my terrible english, but I hope you understand what I mean - Michael)

Chunk Sequence

```
    Chunk Formats
       SXHD-Chunk
       SBDY-Chunk


prepared by:
    Virtual Worlds Productions
    Michael Pfeiffer


The ANIM-Format is a wellknown fileformat for animations but it doesn´t
exists a standard for animation with sound.

The softwarepackage WaveTracer® (by  VIRTUAL WORLDS Productions®) gives
users much possibilities to create good sounds, also in Dolby-Surround®.
The WaveTracer® together with the included CineTracer® makes it possible
to create sounds out of the datafiles of a raytracer. These sounds can
be injected into a ANIM-file. A player wich knows how (e.g. the
AnimFX®-Player, also by Virtual Worlds Productions) can play these anims
with sound.
```

1.33 animsound_chunksequence

```
   Chunk Sequence
   --------------

These method to include a sound into a existing anim is compatible to all
known IFF-ANIM-formats. Ignoring the differences between the formats, a
animfile with additional soundchunks could look like this:

   FORM ANIM
   .  FORM ILBM         first frame
   .  .  BMHD
   .  .  SXHD              chunk with all necessairy definitions
   .  .  SBDY              soundbody with sampledata for the first frame
   .  .  ANHD

   .  .  CMAP
   .  .  BODY/DLTA
   .  FORM ILBM         frame 2
   .  .  ANHD
   .  .  SBDY              sampledata for the second frame
   .  .  DLTA
   .  FORM ILBM         frame 3
   .  .  ANHD
   .  .  SBDY              sampledata for the third frame
   .  .  DLTA

      * * *

   .  FORM ILBM         last frame
   .  .  ANHD
   .  .  SBDY              sampledata for the last frame
   .  .  SBDY              additional sampledata wich have to be played after
```

```
                          the last frame or while fading to an other screen
   .  .  DLTA
```

1.34 animsound_chunkformats

```
   Chunk Formats
   -------------

    SXHD-Chunk
    SBDY-Chunk
```

1.35 animsound_sxhd

```
   SXHD Chunk
   ----------
```

This chunk includes all informations about the soundsample. It isn´t compatible to the VHDR-chunk in the 8SVX format but its the same like in HISX-soundformats of our product WaveTracer®. For more detailed informations about these format, its usage and its soundmodes, see the original WaveTracer®-Manual and the WaveTracer® Developers-Information.

```
   BYTE SampleDepth   depth of the Sample in unit Bits, most common is 8
                      but all others from 1 Bit up to 32 Bits are
                      (theoretical) possible
   BYTE FixedVolume   Amiga-typical unit for playback-volume, 0 is off and
                      64 is maximum
   BYTE Length        length of a SBDY-chunk of ONE frame. This length is
                      calculated for a constant playbackspeed. If speed
                      varies from frame to frame the length of the
                      SBDY-samplechunk also varies and this value is
                      invalid and/or set to 0
   LONG PlayRate      Amiga-typical unit for playback-speed; if you like to
                      use the duration of sampleplayback for the synchroni-
                      sation of the anim-playrate or if you use a soundcard,
                      you better should use "PlayFreq"
   LONG CompressionMethod   has to be =0 in anims everytime, because the
                      AnimPlayer became slower and Dolby-Surround®-
                      informations will be destroyed by all known
                      compressions
   BYTE UsedChannels  Flag for number and type of channels saved in the
                      SBDY-chunk:
                          1 - Channel left
                          2 - Channel right
                          4 - Channel center
                      Left and right together or center are supported this
                      time, but following channels are theoretical also
                      possible:
                          8 - Surroundchannel or surround left
                         16 - Surround right
```

```
                        32 - Subwoofer effectchannel
   BYTE UsedMode       says, wich soundmode has to be used:
                           1: Mode Mono (center-channel only)
                           2: Mode Stereo (channels left and right)
                       Following modes are also possible but not supported
                       yet:
                           3 - 3.0 3Channel
                           4 - 4.0 DTS®-Quadro
                           5 - 4.0 DolbySurround®
                           6 - 5.0 DTS® / AC-3®
                           7 - 5.1 DTS® / AC-3®
   LONG PlayFreq       play-/samplefrequency, spezifies the correct playspeed
   WORD Private        reserved
```

1.36 animsound_sbdy

```
   SBDY Chunk
   ----------
```

The samplechunk SBDY includes all sampledata wich have to be played back for the actual frame. Every SBDY chunk includes only a small part of the whole sample and every part fits to its frame. If you use the correct playback-speed, you can use the sample-playbacktime for synchronising your anim-playbackspeed.

The organisation of the SBDY chunk depends on the definitions in SXHD chunk. The most important examples have following structure:

Sampledepth: 8 Bit
Mode: Mono
   Sampledata are stored like in normal 8SVX-format, UBYTE by UBYTE and only one channel.

Sampledepth: 8 Bit
Mode Stereo
   Here Sampledata are also stored like in normal Stereo-8SVX-Format, first all UBYTEs for the left channel and then all for the right channel.

NOTE! The last frame of a ANIM could have two SBDY chunks. The first one has to be played like all others when the last frame is displayed. The second Frame includes sampledata wich often have an other length. These sampledata have to be played after the end of the animation (e.g. when the screen fades to an other).

NOTE ALSO! In future there can be programs wich support ANIMs with sound. If such a program merges two ANIMs with sound, it can happen, that a frame in the middle of the complete animation owns two SBDY-Chunks. Then you have to do following: Replay the second SBDY with your normal two channels (could be Amiga®-channels A and B) and start immediately replaying the next frame with its SBDY-Sounddata onto the other two channels (the Amiga®-channels D and C). So you will get the perfect fading from one part of a animation to the other (it should look and sound nearly like in good movies!).