Devices

COLLABORATORS			
	TITLE : Devices		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Devices

1.1 A / Third Party Public FORM and Chunk Specifications

```
0000.CSET.doc         ANIM.brush.doc       ILBM.DPI.doc    RGBN-RGB8.doc
0000.FVER.doc         ANIM.doc             ILBM.DPPV.doc   SAMP.doc
8SVX.CHAN.PAN.doc     DR2D.doc             ILBM.DRNG.doc   TDDD.doc
8SVX.SEQN.FADE.doc    FANT.doc             ILBM.EPSF.doc   WORD.doc
ACBM.doc              HEAD.doc             MTRX.doc
AIFF.doc              ILBM.CLUT.doc        PGTB.doc
ANBM.doc              ILBM.CTBL.DYCP.doc   PRSP.doc
```

1.2 A / IFF Third Party Public Form and Chunk Specification / 0000.CSET.doc

```
Chunk for specifying character set

Registered by Martin Taillefer.

A chunk for use in any FORM, to specify character set used for text in
FORM.

   struct CSet {
           LONG    CodeSet;       /* 0=ECMA Latin 1 (std Amiga charset) */
           LONG    Reserved[7];   /* CBM will define additional values  */
           }
```

1.3 A / IFF Third Party Public Form and Chunk Specification / 0000.FVER.doc

```
Chunk for 2.0 VERSION string of an IFF file

Registered by Martin Taillefer.

A chunk for use in any FORM, to contain standard 2.0 version string.

$VER: name ver.rev

where "name" is the name or identifier of the file and ver.rev is a
```

version/revision such as 37.1

Example:

$VER: workbench.catalog 37.42

1.4 A / IFF Third Party Public Form and Chunk Specification / 8SVX.CHAN.PAN.doc

Stereo chunks for 8SVX form

SMUS.CHAN and SMUS.PAN Chunks
Stereo imaging in the "8SVX" IFF 8-bit Sample Voice

Registered by David Jones, Gold Disk Inc.

There are two ways to create stereo imaging when playing back a digitized sound. The first relies on the original sound being created with a stereo sampler: two different samples are digitized simultaneously, using right and left inputs. To play back this type of sample while maintaining the stereo imaging, both channels must be set to the same volume. The second type of stereo sound plays the identical information on two different channels at different volumes. This gives the sample an absolute position in the stereo field. Unfortunately, there are currently a number of methods for doing this currently implemented on the Amiga, none truly adhering to any type of standard. What I have tried to to is provide a way of doing this consistently, while retaining compatibility with existing (non-standard) systems. Introduced below are two optional data chunks, CHAN and PAN. CHAN deals with sounds sampled in stereo, and PAN with samples given stereo characteristics after the fact.

Optional Data Chunk CHAN
Optional Data Chunk PAN

1.5 8SVX.CHAN.PAN.doc / Optional Data Chunk CHAN

This chunk is already written by the software for a popular stereo sampler. To maintain the ability read these samples, its implementation here is therefore limited to maintain compatability.

The optional data chunk CHAN gives the information neccessary to play a sample on a specified channel, or combination of channels. This chunk would be useful for programs employing stereo recording or playback of sampled sounds.

```
    #define RIGHT           4L
    #define LEFT            2L
    #define STEREO          6L

    #define ID_CHAN MakeID('C','H','A','N')

    typedef sampletype LONG;
```

If "sampletype" is RIGHT, the program reading the sample knows that it was originally intended to play on a channel routed to the right speaker, (channels 1 and 2 on the Amiga). If "sampletype" is LEFT, the left speaker was intended (Amiga channels 0 and 3). It is left to the discretion of the programmer to decide whether or not to play a sample when a channel on the side designated by "sampletype" cannot be allocated.

If "sampletype" is STEREO, then the sample requires a pair of channels routed to both speakers (Amiga pairs [0,1] and [2,3]). The BODY chunk for stereo pairs contains both left and right information. To adhere to existing conventions, sampling software should write first the LEFT information, followed by the RIGHT. The LEFT and RIGHT information should be equal in length.

Again, it is left to the programmer to decide what to do if a channel for a stereo pair can't be allocated; wether to play the available channel only, or to allocate another channels routed to the wrong speaker.

1.6 8SVX.CHAN.PAN.doc / Optional Data Chunk PAN

The optional data chunk PAN provides the neccessary information to create a stereo sound using a single array of data. It is neccessary to replay the sample simultaneously on two channels, at different volumes.

```
    #define ID_PAN MakeID('P','A','N',' ')

    typedef sposition Fixed; /* 0 <= sposition <= Unity */

    /* Unity is elsewhere #defined as 10000L, and
     * refers to the maximum possible volume.
     * /

    /* Please note that "Fixed" (elsewhere #defined as LONG) is used to
     * allow for compatabilty between audio hardware of different
     * resolutions.
     * /
```

The "sposition" variable describes a position in the stereo field. The numbers of discrete stereo positions available is equal to 1/2 the number of discrete volumes for a single channel.

The sample must be played on both the right and left channels. The overall volume of the sample is determined by the "volume" field in the Voice8Header structure in the VHDR chunk.

```
The left channel volume = overall volume / (Unity / sposition).
 "  right   "       "   = overall volume - left channel volume.
```

```
For example:
  If sposition = Unity, the sample is panned all the way to the left.
  If sposition = 0, the sample is panned all the way to the right.
  If sposition = Unity/2, the sample is centered in the stereo field.
```

1.7 A / IFF Third Party Public Form & Chunk Specification / 8SVXSEQN.FADE.doc

Looping chunks for 8SVX form

SEQN and FADE Chunks

Multiple Loop Sequencing in the '8SVX' IFF 8-bit Sample Voice

Registered by Peter Norman, RamScan Software Pty Ltd.

Sound samples are notorious for demanding huge amounts of memory.

While earlier uses of digital sound on the Amiga were mainly in the form of short looping waveforms for use as musical instruments, many people today wish to record several seconds (even minutes) of sound. This of course eats memory.

Assuming that quite often the content of these recordings is music, and that quite often music contains several passages which repeat at given times, "verse1 .. chorus .. verse2 .. chorus .." etc, a useful extention has been added to the 8SVX list of optional data chunks. It's purpose is to conserve memory by having the computer repeat sections rather than having several instances of a similar sound or musical passage taking up valuable sample space.

The 'SEQN' chunk has been created to define "Multiple" loops or sections within a single octave 8SVX MONO or STEREO waveform.

It is intended that a sampled sound player program which supports this chunk will play sections of the waveform sequentially in an order that the SEQN chunk specifies. This means for example, if an identical chorus repeats throughout a recording, rather than have this chorus stored several times along the waveform, it is only necessary to have one copy of the chorus stored in the waveform.

A "SEQeNce" of definitions can then be set up to have the computer loop back and repeat the chorus at the required time. The remaining choruses stored in the waveform will no longer be necessary and can be removed.

E.g., if we had a recording of the following example, we would find that there are several parts which simply repeat. Substantial savings can be made by having the computer repeat sections rather than have them stored in memory.

EXAMPLE
Chunk Definitions

1.8 8SVXSEQN.FADE.doc / EXAMPLE

"Haaaallelujah....Haaaallelujah...Hallelujah..Hallelujah..
Halleeeelujaaaah."

Applying a sequence to the above recording would look as follows.

```
Haaaallelujah....Haaaallelujah...Hallelujah..Hallelujah..Halleeeelujaaaah.
[     Loop1     ]
[     Loop2     ]
                                 [  Loop3   ]
                                 [  Loop4   ]
                                                        [     Loop5     ]
               [   Dead Space   ]          [ Dead Space ]
```

The DEAD SPACE can be removed. With careful editing of the multiple loop positions, the passage can be made to sound exactly the same as the original with far less memory required.

1.9 8SVXSEQN.FADE.doc / Chunk Definitions

Optional Data Chunk SEQN
Optional Data Chunk FADE

1.10 8SVXSEQN.FADE.doc / Optional Data Chunk SEQN

The optional data chunk SEQN gives the information necessary to play a sample in a sequence of defined blocks. To have a segment repeat twice, the definition occurs twice in the list.

This list consists of pairs of ULONG "loop start" and "end" definitions which are offsets from the start of the waveform. The locations or values must be LONGWORD aligned (divisable by 4).

To determine how many loop definitions in a given file, simply divide the SEQN chunk size by 8.

E.g., if chunk size == 40 ... number of loops = (40 / 8) .. equals 5 loops.

The raw data in a file might look like this...

```
'S-E-Q-N' [ size ] [     Loop 1    ] [     Loop 2    ] [     Loop 3    ]

 5345514E 00000028 00000000 00000C00 00000000 00000C00 00000C08 00002000
             ^
             ^     'Haaaallelujah..' 'Haaaallelujah..'   'Hallelujah..'
             ^
             ^
             40 bytes decimal / 8 = 5 loop or segments


      [     Loop 4    ] [    Loop 5     ]'B-O-D-Y'   Size     Data

      00000C08 00002000 00002008 00003000 424F4459 000BE974 010101010101010

        'Hallelujah..'  'Halleeeelujah..'
```

In a waveform containing SEQN chunks, the oneShotHiSamples should be set

to 0 and the repeatHiSamples should equal the BODY length (divided by 2 if STEREO).

Remember the locations of the start and end of each segment or loop should be LONGWORD aligned.

If the waveform is Stereo, treat the values and locations in exactly the same way. In other words, if a loop starts at location 400 within a Stereo waveform, you start the sound at the 400th byte position in the left data and the 400th byte position in the right data simultaneously.

```
  #define ID_SEQN MakeID('S','E','Q','N')
```

1.11 8SVXSEQN.FADE.doc / Optional Data Chunk FADE

The FADE chunk defines at what loop number the sound should begin to fade away to silence. It is possible to finish a sample of music in much the same way as commercial music does today. A FADE chunk consists of one ULONG value which has a number in it. This number corresponds to the loop number at which the fade should begin.

eg. You may have a waveform containing 50 loops. A FADE definition of 45 will specify that once loop 45 is reached, fading to zero volume should begin. The rate at which this fade takes place is determined by the length of time left to play. The playing software should do a calculation based on the following...

Length of all remaining sequences including current sequence (in bytes)

divided by

the current playback rate in samples per second

= time remaining.

Begin stepping the volume down at a rate which will hit zero volume just as the waveform finishes.

The raw data in a file may look like this.

```
 'F-A-D-E'  [ Size ]   Loop No.  'B-O-D-Y'   Size   Data..

  46414445  00000004   0000002D   424F4459 000BE974 01010101 01010101 etc
                           ^
                           Start fading when loop number 45 is reached.
```

```
  #define ID_FADE MakeID('F','A','D','E')
```

Although order shouldn't make much difference, it is a general rule of thumb that SEQN should come before FADE and FADE should be last before the BODY.

```
Stereo waveforms would have CHAN,SEQN,FADE,BODY in that order.
```

1.12 A / IFF Third Party Public Form and Chunk Specification / ACBM.doc

```
Amiga Contiguous Bitmap form

                IFF FORM / CHUNK DESCRIPTION

Form/Chunk ID:   FORM  ACBM  (Amiga Contiguous BitMap)
                 Chunk ABIT  (Amiga BITplanes)

Date Submitted:  05/29/86
Submitted by:    Carolyn Scheppner   CBM

 FORM
 Chunk
 Supporting Software
```

1.13 ACBM.doc / FORM

```
FORM ID:  ACBM  (Amiga Contiguous BitMap)

FORM Description:

FORM ACBM has the same format as FORM ILBM except the normal BODY chunk
(InterLeaved BitMap) is replaced by an ABIT chunk (Amiga BITplanes).

FORM Purpose:

To enable faster loading/saving of screens, especially from Basic, while
retaining the flexibility and portability of IFF format files.
```

1.14 ACBM.doc / Chunk

```
Chunk ID:   ABIT  (Amiga BITplanes)

Chunk Description:

The ABIT chunk contains contiguous bitplane data.  The chunk contains
sequential data for bitplane 0 through bitplane n.

Chunk Purpose:

To enable loading/storing of bitmaps with one DOS Read/Write per bitplane.
Significant speed increases are realized when loading/saving screens from
Basic.
```

1.15 ACBM.doc / Supporting Software

```
(Public Domain, available soon via Fish PD disk, various networks)

LoadILBM-SaveACBM (AmigaBasic)
   Loads and displays an IFF ILBM pic file (Graphicraft, DPaint, Images).
   Optionally saves the screen in ACBM format.

LoadACBM (AmigaBasic)
   Loads and display an ACBM format pic file.

SaveILBM (AmigaBasic)
   Saves a demo screen as an ILBM pic file which can be loaded into
   Graphicraft, DPaint, Images.
```

1.16 A / IFF Third Party Public Form and Chunk Specification / AIFF.doc

```
Audio 1-32 bit samples (Mac,AppleII,Synthia Pro)

provided by Steve Milne and Matt Deatherage, Apple Computer, Inc.


            AIFF: Audio Interchange File Format File

The Audio Interchange File Format (Audio IFF) provides a standard for
storing sampled sounds.  The format is quite flexible, allowing the
storage of monaural or multichannel sampled sounds at a variety of sample
rates and sample widths.

Audio IFF conforms to the "EA IFF 85: Standard for Interchange Format
Files" developed by Electronic Arts.

Audio IFF is primarily an interchange format, although application
designers should find it flexible enough to use as a data storage format
as well.  If an application does choose to use a different storage format,
it should be able to convert to and from the format defined in this
document.  This ability to convert will facilitate the sharing of sound
data between applications.

Audio IFF is the result of several meetings held with music developers
over a period of ten months during 1987 and 1988.  Apple Computer greatly
appreciates the comments and cooperation provided by all developers who
helped define this standard.

Another "EA IFF 85" sound storage format is "8SVX IFF 8-bit Sampled
Voice", by Electronic Arts.  '8SVX', which handles eight-bit monaural
samples, is intended mainly for storing sound for playback on personal
computers.  Audio IFF is intended for use with a larger variety of
computers, sampled sound instruments, sound software applications, and
high fidelity recording devices.

 Data Types                              The Marker Chunk
 Constants                               The Instrument Chunk
 Data Organization                       The MIDI Data Chunk
```

1.17 AIFF.doc / AIFF / Data Types

A C-like language will be used to describe the data structures in this document The data types used are listed below.

char: 8 bits signed. A char can contain more than just ASCII characters. It can contain any number from -128 to 127 (inclusive).
unsigned char: 8 bits signed. Contains numbers from 0 to 255 (inclusive).
short: 16 bits signed. Contains any number from -32,768 to 32,767 (inclusive).
unsigned short:16 bits unsigned. Contains any number from 0 to 65,535 (inclusive).
long: 32 bits signed. Contains any number from -2,147,483,648 to 2,147,483,647 (inclusive).
unsigned long: 32 bits unsigned. Contains any number from 0 to 4,294,967,295 (inclusive).
extended: 80 bit IEEE Standard 754 floating point number (Standard Apple Numeric Environment [SANE] data type Extended)
pstring: Pascal-style string, a one-byte count followed by text bytes. The total number of bytes in this data type should be even. A pad byte can be added to the end of the text to accomplish this. This pad byte is not reflected in the count.
ID: 32 bits,the concatenation of four printable ASCII characters in the range ' ' (space, 0x20) through '~' (tilde, 0x7E). Leading spaces are not allowed in the ID but trailing spaces are OK. Control characters are forbidden.

1.18 AIFF.doc / AIFF / Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

1.19 AIFF.doc / AIFF / Data Organization

All data is stored in Motorola 68000 format. The bytes of multiple-byte values are stored with the high-order bytes first. Data is organized as follows:

```
                7  6  5  4  3  2  1  0
        +-----------------------+
```

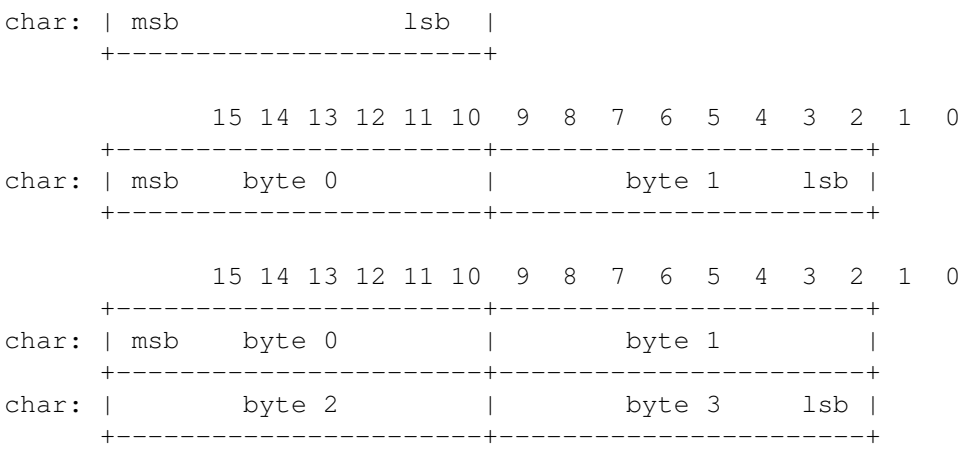


Figure 1: IFF data storage formats

1.20 AIFF.doc / AIFF / Referring to Audio IFF

The official name for this standard is Audio Interchange File Format. If an application program needs to present the name of this format to a user, such as in a "Save As..." dialog box, the name can be abbreviated to Audio IFF. Referring to Audio IFF files by a four-letter abbreviation (i.e., 'AIFF') in user-level documentation or program-generated messages should be avoided.

1.21 AIFF.doc / AIFF / File Structure

The "EA IFF 85 Standard for Interchange Format Files" defines an overall structure for storing data in files. Audio IFF conforms to those portions of "EA IFF 85" that are germane to Audio IFF. For a more complete discussion of "EA IFF 85", please refer to the document "EAIFF 85, Standard for Interchange Format Files."

An "EA IFF 85" file is made up of a number of chunks of data. Chunks are the building blocks of "EA IFF 85" files. A chunk consists of some header information followed by data:

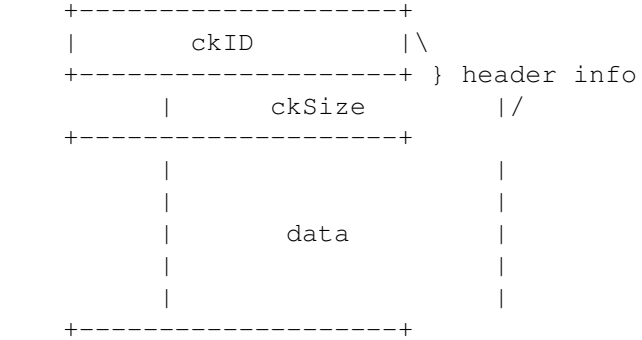


Figure 2: IFF Chunk structure

A chunk can be represented using our C-like language in the following manner:

```
    typedef struct {
  ID    ckID;   /* chunk ID   */
  long    ckSize;   /* chunk Size   */

  char    ckData[]; /* data     */
  } Chunk;
```

The ckID describes the format of the data portion of a chunk. A program can determine how to interpret the chunk data by examining ckID.

The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The ckData contains the data stored in the chunk. The format of this data is determined by ckID. If the data is an odd number of bytes in length, a zero pad byte must be added at the end. The pad byte is not included in ckSize.

Note that an array with no size specification (e.g., char ckData[];) indicates a variable-sized array in our C-like language. This differs from standard C.

An Audio IFF file is a collection of a number of different types of chunks. There is a Common Chunk which contains important parameters describing the sampled sound, such as its length and sample rate. There is a Sound Data Chunk which contains the actual audio samples. There are several other optional chunks which define markers, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in later sections of this document.

The chunks in an Audio IFF file are grouped together in a container chunk. "EA IFF 85" Standard for Interchange Format Files defines a number of container chunks, but the one used by Audio IFF is called a FORM. A FORM has the following format:

```
    typedef struct {
  ID  ckID;
  long  ckSize;
  ID  formType;
  char  chunks[];
    }
```

The ckID is always 'FORM'. This indicates that this is a FORM chunk.

The ckSize contains the size of data portion of the 'FORM' chunk. Note that the data portion has been broken into two parts, formType and chunks[].

The formType field describes what's in the 'FORM' chunk. For Audio IFF files, formType is always 'AIFF'. This indicates that the chunks within the FORM pertain to sampled sound. A FORM chunk of formType 'AIFF' is called a FORM AIFF.

The chunks field are the chunks contained within the FORM. These chunks

are called local chunks. A FORM AIFF along with its local chunks make up an Audio IFF file.

Here is an example of a simple Audio IFF file. It consists of a file containing single FORM AIFF which contains two local chunks, a Common Chunk and a Sound Data Chunk.

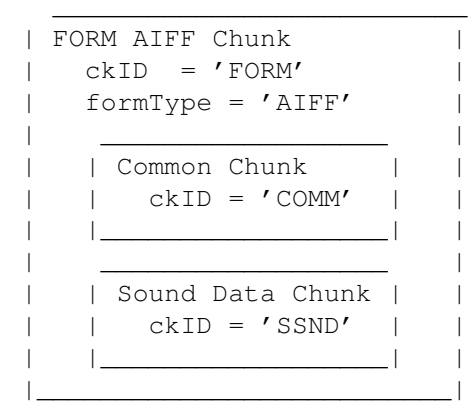


Figure 3: Simple Audio IFF File

There are no restrictions on the ordering of local chunks within a FORM AIFF.

A more detailed example of an Audio IFF file can be found in Appendix A. Please refer to this example as often as necessary while reading the remainder of this document.

1.22 AIFF.doc / AIFF / Storage of AIFF on Apple and Other Platforms

On a Macintosh, the FORM AIFF, is stored in the data fork of an Audio IFF file. The Macintosh file type of an Audio IFF file is 'AIFF'. This is the same as the formType of the FORM AIFF. Macintosh applications should not store any information in Audio IFF file's resource fork, as this information may not be preserved by all applications. Applications can use the Application Specific Chunk, defined later in this document, to store extra information specific to their application.

Audio IFF files may be identified in other Apple file systems as well. On a Macintosh under MFS or HFS, the FORM AIFF is stored in the data fork of a file with file type 'AIFF'. This is the same as the formType of the FORM AIFF.

On an operating system such as MS-DOS or UNIX, where it is customary to use a file name extension, it is recommended that Audio IFF file names use '.AIF' for the extension.

On an Apple II, FORM AIFF is stored in a file with file type $D8 and auxiliary type $0000. Versions 1.2 and earlier of the Audio IFF standard used file type $CB and auxiliary type $0000. This is incorrect; the assignment listed in this document is the correct assignment.

On the Apple IIGS stereo data is stored with right data on even channels

and left data on odd channels. Some portions of AIFF do not follow this convention. Even where it does follow the convention, AIFF usually uses channel two for right data instead of channel zero as most Apple IIGS standards do. Be prepared to interpret data accordingly.

1.23 AIFF.doc / AIFF / Local Chunk Types

The formats of the different local chunk types found within a FORM AIFF are described in the following sections, as are their ckIDs.

There are two types of chunks: required and optional. The Common Chunk is required. The Sound Data chunk is required if the sampled sound has a length greater than zero. All other chunks are optional. All applications that use FORM AIFF must be able to read the required chunks and can choose to selectively ignore the optional chunks. A program that copies a FORM AIFF should copy all the chunks in the FORM AIFF, even those it chooses not to interpret.

1.24 AIFF / Local Chunk Types / The Common Chunk

The Common Chunk describes fundamental parameters of the sampled sound.

```
    #define CommonID  'COMM'  /* ckID for Common Chunk */

    typedef struct {
  ID    ckID;
  long    ckSize;

  short   numChannels;
  unsigned long numSampleFrames;
  short   sampleSize;
  extended  sampleRate;
    } CommonChunk;
```

The ckID is always 'COMM'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize. For the Common Chunk, ckSize is always 18.

The numChannels field contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, and 4 means four channel sound, etc. Any number of audio channels may be represented. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame.

The actual sound samples are stored in another chunk, the Sound Data Chunk, which will be described shortly.

Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each

channel available.

The numSampleFrames field contains the number of sample frames. This is not necessarily the same as the number of bytes nor the number of samplepoints in the Sound Data Chunk. The total number of sample points in the file is numSampleFrames times numChannels.

The sampleSize is the number of bits in each sample point. It can be any number from 1 to 32. The format of a sample point will be described in the next section.

The sampleRate field is the sample rate at which the sound is to be played back in sample frames per second.

One, and only one, Common Chunk is required in every FORM AIFF.

1.25 AIFF / Local Chunk Types / Sound Data Chunk

The Sound Data Chunk contains the actual sample frames.

```
    #define   SoundDataID 'SSND'  /* ckID for Sound Data Chunk  */

    typedef struct {
  ID    ckID;
  long    ckSize;

  unsigned long offset;
  unsigned long blockSize;
  unsigned char SoundData [];
    }  SoundDataChunk;
```

The ckID is always 'SSND'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The offset field determines where the first sample frame in the soundData starts. The offset is in bytes. Most applications won't use offset and should set it to zero. Use for a non-zero offset is explained in the Block-Aligning Sound Data section below.

The blockSize is used in conjunction with offset for block-aligning sound data. It contains the size in bytes of the blocks that sound data is aligned to. As with offset, most applications won't use blockSize and should set it to zero. More information on blockSize is in the Block-Aligning Sound Data section below.

The soundData field contains the sample frames that make up the sound. The number of sample frames in the soundData is determined by the numSampleFrames field in the Common Chunk. Sample points and sample frames are explained in detail in the next section.

The Sound Data Chunk is required unless the numSampleFrames field in the Common Chunk is zero. A maximum of one Sound Data Chunk may appear in a FORM AIFF.

1.26 AIFF.doc / AIFF / Sample Points and Sample Frames

A large part of interpreting Audio IFF files revolves around the two concepts of sample points and sample frames.

A sample point is a value representing a sample of a sound at a given point in time. Each sample point is stored as a linear, 2's-complement value which may be from 1 to 32 bits wide, as determined by sampleSize in the Common Chunk.

Sample points are stored in an integral number of contiguous bytes. One- to eight-bit wide sample points are stored in one byte, 9- to 16-bit wide sample points are stored in two bytes, 17- to 24-bit wide sample points are stored in three bytes, and 25- to 32-bit wide sample points are stored in four bytes (most significant byte first). When the width of a sample point is not a multiple of eight bits, the sample point data is left justified, with the remaining bits zeroed. An example case is illustrated in Figure 4. A 12-bit sample point, binary 101000010111, is stored left justified in two bytes. The remaining bits are set to zero.

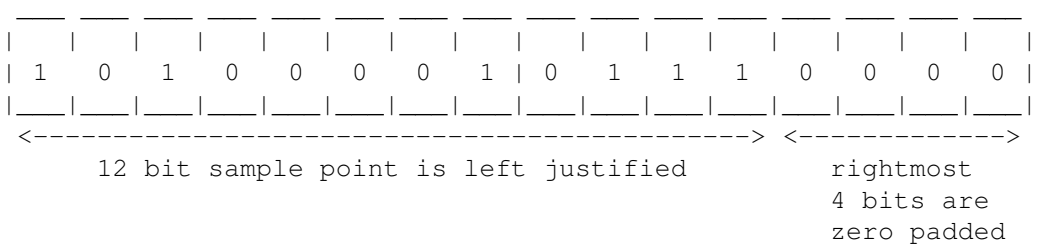


Figure 4: 12-Bit Sample Point

For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame. Single sample points from each channel are interleaved such that each sample frame is a sample point from the same moment in time for each channel available. This is illustrated in Figure 5 for the stereo (two channel) case.

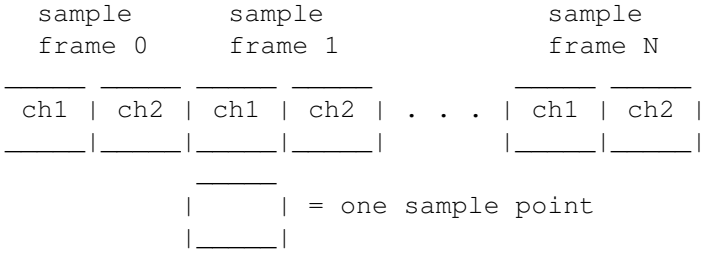
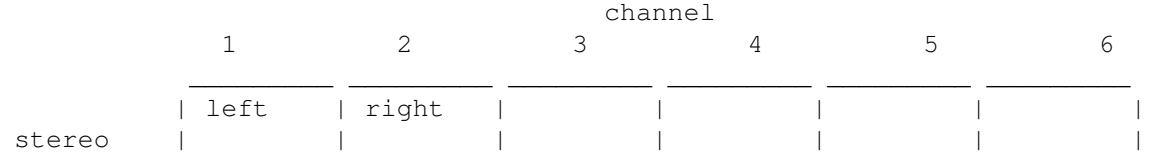


Figure 5: Sample Frames for Multichannel Sound

For monophonic sound, a sample frame is a single sample point. For multichannel sounds, you should follow the conventions in Figure 6.



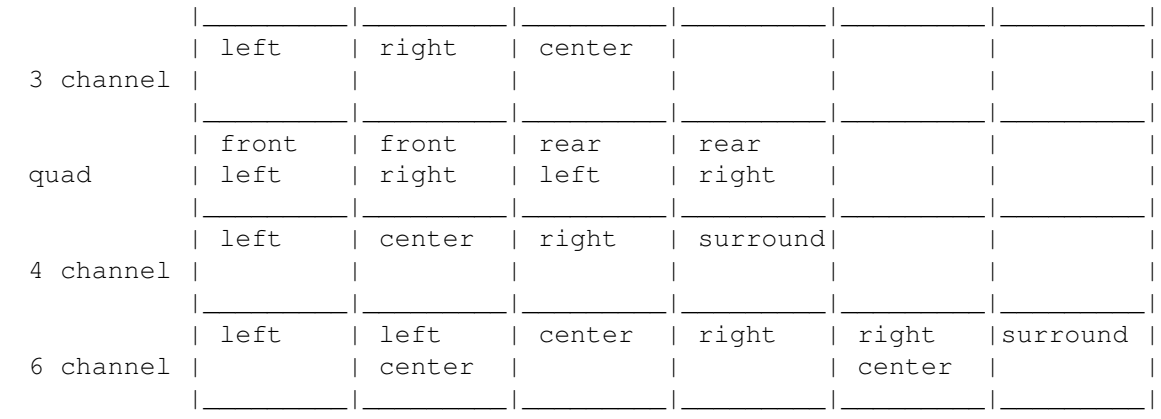


Figure 6: Sample Frame Conventions for Multichannel Sound

Sample frames are stored contiguously in order of increasing time. The sample points within a sample frame are packed together; there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

1.27 AIFF.doc / AIFF / Block-Aligning Sound Data

There may be some applications that, to ensure real time recording and playback of audio, wish to align sampled sound data with fixed-size blocks. This alignment can be accomplished with the offset and blockSize parameters of the Sound Data Chunk, as shown in Figure 7.

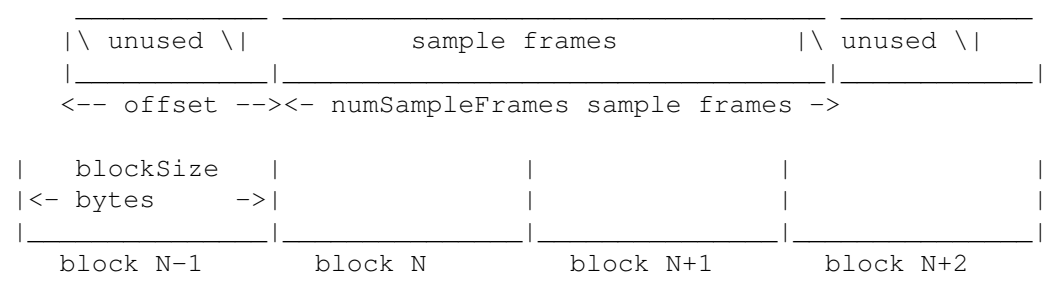


Figure 7: Block-Aligned Sound Data

In Figure 7, the first sample frame starts at the beginning of block N. This is accomplished by skipping the first offset bytes of the soundData. Note too, that the soundData bytes can extend beyond valid sample frames, allowing the soundData bytes to end on a block boundary as well.

The blockSize specifies the size in bytes of the block to which you would align the sound data. A blockSize of zero indicates that the sound data does not need to be block-aligned. Applications that don't care about block alignment should set the blockSize and offset to zero when creating Audio IFF files. Applications that write block-aligned sound data should set blockSize to the appropriate block size. Applications that modify an existing Audio IFF file should try to preserve alignment of the sound data, although this is not required. If an application does not preserve alignment, it should set the blockSize and offset to zero. If an application needs to realign sound data to a different sized block, it

should update blockSize and offset accordingly.

1.28 AIFF.doc / AIFF / The Marker Chunk

The Marker Chunk contains markers that point to positions in the sound data. Markers can be used for whatever purposes an application desires. The Instrument Chunk, defined later in this Note, uses markers to mark loop beginning and end points.

Markers
Marker Chunk Format

1.29 AIFF / The Marker Chunk / Markers

A marker has the following format.

```
typedef short   MarkerId;

typedef struct  {
MarkerID  id;
unsigned long position;
pstring   markerName;
} Marker;
```

The id is a number that uniquely identifies that marker within a FORM AIFF. The id can be any positive non-zero integer, as long as no other marker within the same FORM AIFF has the same id.

The marker's position in the sound data is determined by the position field. Markers conceptually fall between two sample frames. A marker that falls before the first sample frame in the sound data is at position zero, while a marker that falls between the first and second sample frame in the sound data is at position 1. Note that the units for position are sample frames, not bytes nor sample points.

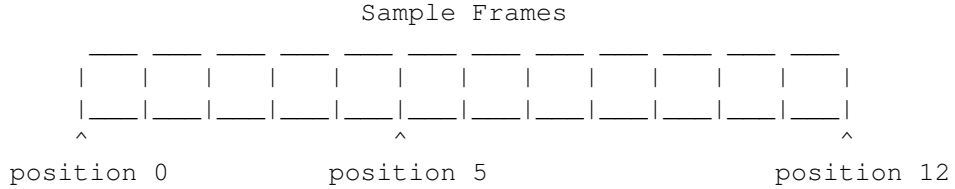


Figure 8: Sample Frame Marker Positions

The markerName field is a Pascal-style text string containing the name of the mark.

Note: Some "EA IFF 85" files store strings a C-strings (text bytes followed by a null terminating character) instead of Pascal-style strings. Audio IFF uses pstrings because they are more efficiently skipped over when scanning through chunks. Using pstrings, a program can skip over a string by adding the string count to the address of the first character.

C strings require that each character in the string be examined for the
null terminator.

1.30 AIFF / The Marker Chunk / Marker Chunk Format

The format for the data within a Marker Chunk is shown below.

```
    #define MarkerID  'MARK'  /* ckID for Marker Chunk */

    typedef  struct {
  ID        ckID;
  long        ckSize;

  unsigned short    numMarkers;
  Marker      Markers [];
    } MarkerChunk;
```

The ckID is always 'MARK'. The ckSize is the size of the data portion of
the chunk in bytes. It does not include the 8 bytes used by ckID and
ckSize.

The numMarkers field is the number of markers in the Marker Chunk. If
numMarkers is non-zero, it is followed by the markers themselves. Because
all fields in a marker are an even number of bytes, the length of any
marker will always be even. Thus, markers are packed together with no
unused bytes between them. The markers need not be ordered in any
particular manner.

The Marker Chunk is optional. No more than one Marker Chunk can appear in
a FORM AIFF.

1.31 AIFF.doc / AIFF / The Instrument Chunk

The Instrument Chunk defines basic parameters that an instrument, such as
a sample, could use to play the sound data.

1.32 AIFF / The Instrument Chunk / Looping

Sound data can be looped, allowing a portion of the sound to be repeated
in order to lengthen the sound. The structure below describes a loop.

```
    typedef struct {
  short PlayMode;
  MarkerId beginLoop;
  MarkerId endLoop;
    } Loop;
```

A loop is marked with two points, a begin position and an end position. There are two ways to play a loop, forward looping and forward/backward looping. In the case of forward looping, playback begins at the beginning of the sound, continues past the begin position and continues to the end position, at which point playback starts again at the begin position. The segment between the begin and end positions, called the loop segment, is played repeatedly until interrupted by a user action, such as the release of a key on a sampling instrument.

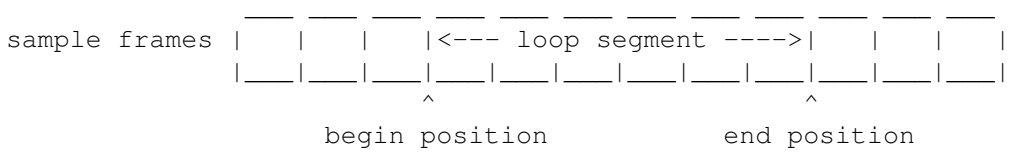


Figure 9: Sample Frame Looping

With forward/backward looping, the loop segment is first played from the begin position to the end position, and then played backwards from the end position to the begin position. This flip-flop pattern is repeated over and over again until interrupted.

The playMode specifies which type of looping is to be performed:

```
#define NoLooping   0
#define ForwardLooping    1
#define ForwardBackwardLooping  2
```

If NoLooping is specified, then the loop points are ignored during playback.

The beginLoop is a marker id that marks the begin position of the loop segment.

The endLoop marks the end position of a loop. The begin position must be less than the end position. If this is not the case, then the loop segment has zero or negative length and no looping takes place.

1.33 AIFF / The Instrument Chunk / The Instrument Chunk Format

The format of the data within an Instrument Chunk is described below.

```
#define             InstrumentID  'INST' /*ckID for Instruments Chunk */

typedef struct {
    ID              ckID;
    long            ckSize;

    char            baseNote;
    char            detune;
    char            lowNote;
    char            highNote;
    char            lowvelocity;
    char            highvelocity;
    short           gain;
```

```
    Loop            sustainLoop;
    Loop            releaseLoop;
} InstrumentChunk;
```

The ckID is always 'INST'. ckSize is the size of the data portion of the chunk, in bytes. For the Instrument Chunk, ckSize is always 20.

The baseNote is the note at which the instrument plays back the sound data without pitch modification. Units are MIDI (MIDI is an acronym for Musical Instrument Digital Interface) note numbers, and are in the range 0 through 127. Middle C is 60.

The detune field determines how much the instrument should alter the pitch of the sound when it is played back. Units are in cents (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised.

The lowNote and highNote fields specify the suggested range on a keyboard for playback of the sound data. The sound data should be played if the instrument is requested to play a note between the low and high, inclusive. The base note does not have to be within this range. Units for lowNote and highNote are MIDI note values.

The lowVelocity and highVelocity fields specify the suggested range of velocities for playback of the sound data. The sound data should be played if the note-on velocity is between low and high velocity, inclusive. Units are MIDI velocity values, 1 (lowest velocity) through 127 (highest velocity).

The gain is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0db means no change, 6db means double the value of each sample point, while -6db means halve the value of each sample point.

The sustainLoop field specifies a loop that is to be played when an instrument is sustaining a sound.

The releaseLoop field specifies a loop that is to be played when an instrument is in the release phase of playing back a sound. The release phase usually occurs after a key on an instrument is released.

The Instrument Chunk is optional. No more than one Instrument Chunk can appear in a FORM AIFF.

ASIF Note: The Apple IIGS Sampled Instrument Format also defines a chunk with ID of 'INST', which is not the same as the Audio IFF Instrument Chunk. A good way to tell the two chunks apart in generic IFF-style readers is by the ckSize fields. The Audio IFF Instrument Chunk's ckSize field is always 20, whereas the Apple IIGS Sampled Instrument Format Instrument Chunk's ckSize field, for structural reasons, can never be 20.

1.34 AIFF.doc / AIFF / The MIDI Data Chunk

The MIDI Data Chunk can be used to store MIDI data. Please refer to Musical Instrument Digital Interface Specification 1.0, available from the International MIDI Association, for more details on MIDI.

The primary purpose of this chunk is to store MIDI System Exclusive messages, although other types of MIDI data can be stored in the block as well. As more instruments come to market, they will likely have parameters that have not been included in the Audio IFF specification. The MIDI System Exclusive messages for these instruments may contain many parameters that are not included in the Instrument Chunk. For example, a new sampling instrument may have more than the two loops defined in the Instrument Chunk. These loops will likely be represented in the MIDI System Exclusive message for the new machine. This MIDI System Exclusive message can be stored in the MIDI Data Chunk.

```
    #define   MIDIDataID   'MIDI' /* ckID for MIDI Data Chunk */

    typedef struct {
  ID    ckID;
  long    ckSize;

  unsigned char MIDIdata[];
    } MIDIDataChunk;
```

The ckID is always 'MIDI'. ckSize of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The MIDIData field contains a stream of MIDI data.

The MIDI Data Chunk is optional. Any number of MIDI Data Chunks may exist in a FORM AIFF. If MIDI System Exclusive messages for several instruments are to be stored in a FORM AIFF, it is better to use one MIDI Data Chunk per instrument than one big MIDI Data Chunk for all of the instruments.

1.35 AIFF.doc / AIFF / The Audio Recording Chunk

The Audio Recording Chunk contains information pertinent to audio recording devices.

```
    #define     AudioRecordingID 'AESD'  /* ckID for Audio Recording */
                                         /*  Chunk.                  */
    typedef struct {
        ID                      ckID
        long                    ckSize;

        unsigned char   AESChannelStatusData[24];
    } AudioRecordingChunk;
```

The ckID is always 'AESD'. The ckSize is the size of the data portion of the chunk, in bytes For the Audio Recording Chunk, ckSize is always 24.

The 24 bytes of AESCChannelStatusData are specified in the "AES

Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data", transmission of digital audio between audio devices. This information is duplicated in the Audio Recording Chunk for convenience. Of general interest would be bits 2, 3, and 4 of byte 0, which describe recording emphasis.

The Audio Recording Chunk is optional. No more than one Audio Recording Chunk may appear in a FORM AIFF.

1.36 AIFF.doc / AIFF / The Application Specific Chunk

The Application Specific Chunk can be used for any purposes whatsoever by developers and application authors. For example, an application that edits sounds might want to use this chunk to store editor state parameters such as magnification levels, last cursor position, etc.

```
    #define ApplicationSpecificID 'APPL' /* ckID for Application */
               /*  Specific Chunk.     */
    typedef struct {
  ID    ckID;
  long    ckSize;
  OSType    applicationSignature;
  char    data[];
    } ApplicationSpecificChunk;
```

The ckID is always 'APPL'. The ckSize is the size of the data portion of the chunk, in bytes. It does not include the 8 bytes used by ckID and ckSize.

The applicationSignature identifies a particular application. For Macintosh applications, this will be the application's four character signature.

The OSType field is used by applications which run on platforms from Apple Computer, Inc. For the Apple II, the OStype field should be set to 'pdos'. For the Macintosh, this field should be set to the four character signature as registered with Apple Technical Support.

The data field is the data specific to the application.

The Application Specific Chunk is optional. Any number of Application Specific Chunks may exist in a single FORM AIFF.

1.37 AIFF.doc / AIFF / The Comments Chunk

The Comments Chunk is used to store comments in the FORM AIFF. "EA IFF 85" has an Annotation Chunk (used in ASIF) that can be used for comments, but the Comments Chunk has two features not found in the "EA IFF 85" chunk. They are a time-stamp for the comment and a link to a marker.

Comment
Comments Chunk Format

1.38 AIFF / The Comments Chunk / Comment

```
A comment consists of a time stamp, marker id, and a text count followed
by text.

    typedef struct {
  unsigned long timeStamp;
  MarkerID  marker;
  unsigned short  count;
  char    text;
    } Comment;

The timeStamp indicates when the comment was created.  On the Amiga, units
are the number of seconds since January 1, 1978.  On the Macintosh, units
are the number of seconds since January 1, 1904.

A comment can be linked to a marker.  This allows applications to store
long descriptions of markers as a comment.  If the comment is referring to
a marker, then the marker field is the ID of that marker.  Otherwise,
marker is zero, indicating that this comment is not linked to a marker.

The count is the length of the text that makes up the comment.  This is a
16-bit quantity, allowing much longer comments than would be available
with a pstring.

The text field contains the comment itself.

The Comments Chunk is optional.  No more than one Comments Chunk may
appear in a single FORM AIFF.
```

1.39 AIFF / The Comments Chunk / Comments Chunk Format

```
    #define   CommentID 'COMT'  /* ckID for Comments Chunk  */

    typedef struct {
  ID    ckID;
  long    ckSize;

  unsigned short  numComments;
  Comment   comments[];
    }CommentsChunk;

The ckID is always 'COMT'.  The ckSize is the size of the data portion of
the chunk, in bytes.  It does not include the 8 bytes used by ckID and
ckSize.

The numComments field contains the number of comments in the Comments
Chunk. This is followed by the comments themselves.  Comments are always
even numbers of bytes in length, so there is no padding between comments
in the Comments Chunk.

The Comments Chunk is optional.  No more than one Comments Chunk may
appear in a single FORM AIFF.
```

1.40 AIFF.doc / AIFF / The Text Chunks, Name, Author, Copyright, Annotation

```
These four chunks are included in the definition of every "EA IFF 85"
file. All are text chunks; their data portion consists solely of text.
Each of these chunks is optional.

    #define   NameID 'NAME'   /* ckID for Name Chunk */
    #define   NameID 'AUTH'   /* ckID for Author Chunk */
    #define   NameID '(c) '   /* ckID for Copyright Chunk */
    #define   NameID 'ANNO'   /* ckID for Annotation Chunk */

    typedef struct {
  ID  ckID;
  long  ckSize;
  char  text[];
    }TextChunk;

The ckID is either 'NAME', 'AUTH', '(c) ', or 'ANNO' depending on whether
the chunk is a Name Chunk, Author Chunk, Copyright Chunk, or  Annotation
Chunk, respectively.  For the Copyright Chunk, the 'c' is lowercase and
there is a space (0x20) after the close parenthesis.

The ckSize is the size of the data portion of the chunk, in this case the
text.

The text field contains pure ASCII characters.  it is not a pstring or a C
string.  The number of characters in text is determined by ckSize.  The
contents of text depend on the chunk, as described below:

 Name Chunk
 Author Chunk
 Copyright Chunk
 Annotation Chunk
```

1.41 AIFF / The Text Chunks / Name Chunk

```
The text contains the name of the sampled sound.  The Name Chunk is
optional.  No more than one Name Chunk may exist within a FORM AIFF.
```

1.42 AIFF / The Text Chunks / Author Chunk

```
The text contains one or more author names.  An author in this case is the
creator of a sampled sound.  The Author Chunk is optional.  No more than
one Author Chunk may exist within a FORM AIFF.
```

1.43 AIFF / The Text Chunks / Copyright Chunk

```
The Copyright Chunk contains a copyright notice for the sound.  The text
field contains a date followed by the name of the copyright owner.  The
```

chunk ID '(c) ' serves as the copyright character. For example, a Copyright Chunk containing the text "1991 Commodore-Amiga, Inc." means "(c) 1991 Commodore-Amiga, Inc." The Copyright Chunk is optional. No more than one Copyright Chunk may exist within a FORM AIFF.

1.44 AIFF / The Text Chunks / Annotation Chunk

The text contains a comment. Use of this chunk is discouraged within a FORM AIFF. The more powerful Comments Chunk should be used instead. The Annotation Chunk is optional. Many Annotation Chunks may exist within a FORM AIFF.

1.45 AIFF.doc / AIFF / Chunk Precedence

Several of the local chunks for FORM AIFF may contain duplicate information. For example, the Instrument Chunk defines loop points and MIDI System Exclusive data in the MIDI Data Chunk may also define loop points. What happens if these loop points are different? How is an application supposed to loop the sound? Such conflicts are resolved by defining a precedence for chunks. This precedence is illustrated in Figure 10.

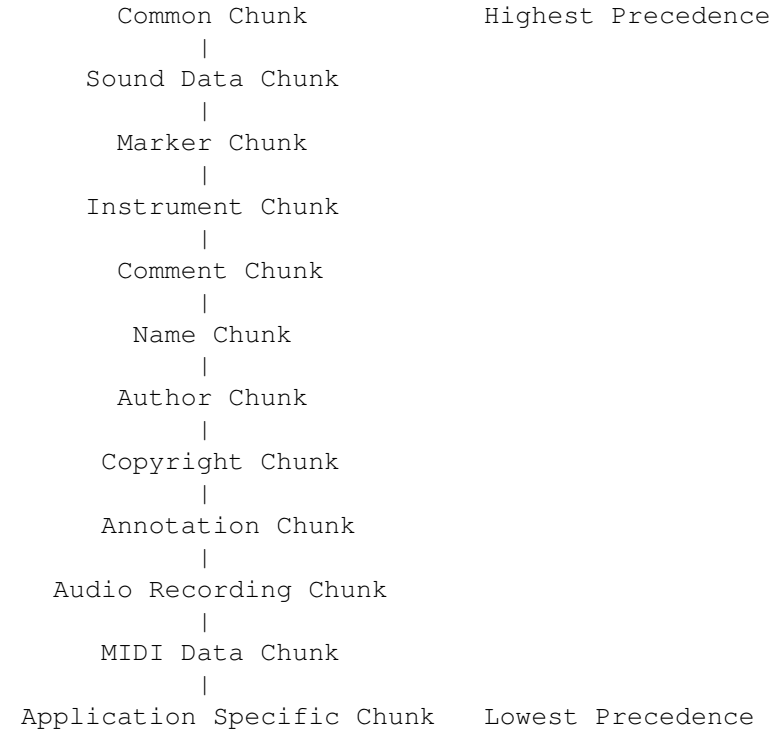


Figure 10: Chunk Precedence

The Common Chunk has the highest precedence, while the Application Specific Chunk has the lowest. Information in the Common Chunk always takes precedence over conflicting information in any other chunk. The Application Specific Chunk always loses in conflicts with other chunks.

By looking at the chunk hierarchy, for example, one sees that the loop points in the Instrument Chunk take precedence over conflicting loop points found in the MIDI Data Chunk.

It is the responsibility of applications that write data into the lower precedence chunks to make sure that the higher precedence chunks are updated accordingly.

Figure 11 illustrates an example of a FORM AIFF. An Audio IFF file is simple a file containing a single FORM AIFF. The FORM AIFF is stored in the data fork of Macintosh file systems that can handle resource forks.

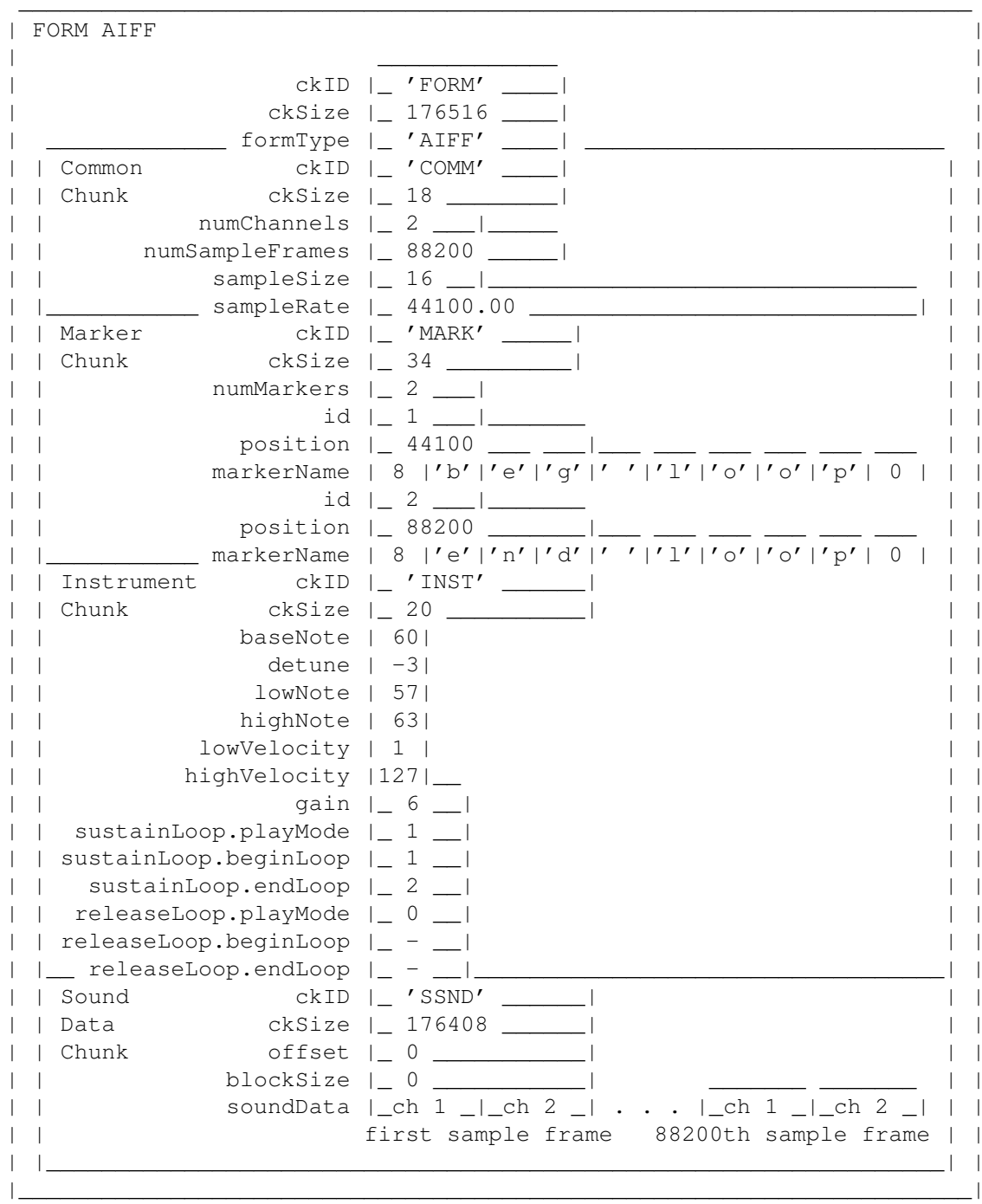


Figure 11: Sample FORM AIFF

1.46 AIFF.doc / AIFF / Further Reference

- "Inside Macintosh", Volume II, Apple Computer, Inc.
- "Apple Numerics Manual", Second Edition, Apple Computer, Inc.
- "File Type Note: File Type $D8, Auxiliary Type $0002, Apple IIGS Sampled Instrument Format", Apple Computer, Inc.
- "Audio Interchange File Format v1.3", APDA
- "AES Recommended Practice for Digital Audio Engineering--Serial Transmission Format for Linearly Represented Digital Audio Data", Audio Engineering Society, 60 East 42nd Street, New York, NY 10165
- "MIDI: Musical Instrument Digital Interface, Specification 1.0", the International MIDI Association.
- "'EA IFF 85' Standard for Interchange Format Files", Electronic Arts
- "'8SVX' IFF 8-bit Sampled Voice", Electronic Arts

1.47 A / IFF Third Party Public Form and Chunk Specification / ANBM.doc

Animated bitmap form (Framer, Deluxe Video)

TITLE: Form ANBM (animated bitmap form used by Framer, Deluxe Video)

(note from the author)

The format was designed for simplicity at a time when the IFF standard was very new and strange to us all. It was not designed to be a general purpose animation format. It was intended to be a private format for use by DVideo, with the hope that a more powerful format would emerge as the Amiga became more popular.

I hope you will publish this format (and we did!) so that other formats will not inadvertantly conflict with it.

PURPOSE: To define simple animated bitmaps for use in DeluxeVideo.

In Deluxe Video objects appear and move in the foreground with a picture in the background. Objects are "small" bitmaps usually saved as brushes from DeluxePaint and pictures are large full screen bitmaps saved as files from DeluxePaint.

Two new chunk headers are defined: ANBM and FSQN.

An animated bitmap (ANBM) is a series of bitmaps of the same size and depth. Each bitmap in the series is called a frame and is labeled by a character, 'a b c ...' in the order they appear in the file.

The frame sequence chunk (FSQN) specifies the playback sequence of the individual bitmaps to achieve animation. FSQN_CYCLE and FSQN_TOFRO specify two algorithmic sequences. If neither of these bits is set, an arbitrary sequence can be used instead.

```
    ANBM          - identifies this file as an animated bitmap
    .FSQN         - playback sequence information
    .LIST ILBM    - LIST allows following ILBMs to share properties
    ..PROP ILBM   - properties follow
    ...BMHD       - bitmap header defines common size and depth
    ...CMAP       - colormap defines common colors
    ..FORM ILBM   - first frame follows
    ..BODY        - the first frame
       .          - FORM ILBM and BODY for each remaining frame
       .
       .
```

ANBM
FSQN
Supporting Software

1.48 ANBM.doc / ANBM

Chunk Description:

The ANBM chunk identifes this file as an animated bitmap

Chunk Spec:

```
   #define ANBM    MakeID('A','N','B','M')
```

Disk record:

   none

1.49 ANBM.doc / FSQN

Chunk Description:

The FSQN chunk specifies the frame playback sequence

Chunk Spec:

```
#define FSQN    MakeID('F','S','Q','N')

/* Flags */
#define FSQN_CYCLE  0x0001 /* Ignore sequence, cycle a,b,..y,z,a,b,.. */
#define FSQN_TOFRO  0x0002 /* Ignore sequence, cycle a,b,..y,z,y,..a,b, */
/* Disk record */
typedef struct {
    WORD numframes;      /* Number of frames in the sequence */
    LONG dt;             /* Nominal time between frames in jiffies */
    WORDBITS flags;      /* Bits modify behavior of the animation */
    UBYTE sequence[80];  /* string of 'a'..'z' specifying sequence */
    } FrameSeqn;
```

1.50 ANBM.doc / Supporting Software

DeluxeVideo by Mike Posehn and Tom Case for Electronic Arts

1.51 A / IFF Third Party Public Form and Chunk Specification / ANIM.brush.doc

ANIM brush format

Dpaint Anim Brush IFF Format

From a description by the author of DPaint,
Dan Silva, Electronic Arts

The "Anim Brushes" of DPaint III are saved on disk in the IFF "ANIM" format. Basically, an ANIM Form consists of an initial ILBM which is the first frame of the animation, and any number of subsequent "ILBM"S (which aren't really ILBM's) each of which contains an ANHD animation header chunk and a DLTA chunk comprised of the encoded difference between a frame and a previous one.

To use ANIM terminology (for a description of the ANIM format, see the IFF Anim Spec, by Gary Bonham). Anim Brushes use a "type 5" encoding, which is a vertical, byte-oriented delta encoding (based on Jim Kent's RIFF). The deltas have an interleave of 1, meaning deltas are computed between adjacent frames, rather than between frames 2 apart, which is the usual ANIM custom for the purpose of fast hardware page-flipping. Also, the deltas use Exclusive Or to allow reversable play.

However, to my knowledge, all the existing Anim players in the Amiga world will only play type 5 "Anim"s which have an interleave of 0 (i.e. 2) and which use a Store operation rather than Exclusive Or, so no existing programs will read Anim Brushes anyway. The job of modifying existing Anim readers to read Anim Brushes should be simplified, however.

Here is an outline of the structure of the IFF Form output by DPaint III as an "Anim Brush". The IFF Reader should of course be flexible enough to tolerate variation in what chunks actually appear in the initial ILBM.

```
FORM ANIM
    . FORM ILBM         first frame
    . . BMHD
    . . CMAP
    . . DPPS
    . . GRAB
    . . CRNG
    . . CRNG
    . . CRNG
    . . CRNG
    . . CRNG
    . . CRNG
    . . DPAN     my own little chunk.
    . . CAMG
    . . BODY
```

```
                        . FORM ILBM         frame 2
                        . . ANHD                animation header chunk
                        . . DLTA                delta mode data

                        . FORM ILBM         frame 3
                        . . ANHD                animation header chunk
                        . . DLTA                delta mode data

                        . FORM ILBM         frame 4
                        . . ANHD                animation header chunk
                        . . DLTA                delta mode data
       ...
                     . FORM ILBM         frame N
                        . . ANHD                animation header chunk
                        . . DLTA                delta mode data

 DPAN Chunk Format
 ANHD Chunk Format
 RIFF
```

1.52 ANIM.brush.doc / DPAN chunk Format

```
   typedef struct {
    UWORD version;   /* current version=4 */
    UWORD nframes;   /* number of frames in the animation.*/
    ULONG flags;   /* Not used */
    } DPAnimChunk;
```

The version number was necessary during development. At present all I look at is "nframes".

1.53 ANIM.brush.doc / ANHD chunk format

```
typedef struct {
 UBYTE operation;  /* =0  set directly
       =1  XOR ILBM mode,
       =2 Long Delta mode,
       =3 Short Delta mode
       =4 Generalize short/long Delta mode,
       =5 Byte Vertical Delta (riff)
       =74 (Eric Grahams compression mode)
   */
 UBYTE mask;      /* XOR ILBM only: plane mask where data is*/
 UWORD w,h;
 WORD x,y;
 ULONG abstime;
 ULONG reltime;
 UBYTE interleave; /* 0 defaults to 2 */
 UBYTE pad0;   /* not used */
 ULONG bits;   /* meaning of bits:
    bit# =0                 =1
    0    short data         long data
```

```
    1    store              XOR
    2    separate info      one info for
         for each plane     for all planes
    3    not RLC            RLC (run length encoded)
    4    horizontal         vertical
    5    short info offsets long info offsets
   */
 UBYTE pad[16];
 } AnimHdr;


for Anim Brushes, set:

 animHdr.operation = 5;  /* RIFF encoding */
 animHdr.interleave = 1;
 animHdr.w = curAnimBr.bmob.pict.box.w;
 animHdr.h = curAnimBr.bmob.pict.box.h;
 animHdr.reltime = 1;
 animHdr.abstime = 0;
 animHdr.bits = 4; /* indicating XOR */

- everything else is set to 0.

NOTE: the "bits" field was actually intended ( by the original creator of
the ANIM format, Gary Bonham of SPARTA, Inc.) for use with only with
compression method 4. I am using bit 2 of the bits field to indicate the
Exclusive OR operation in the context of method 5, which seems like a
reasonable generalization.

For an Anim Brush with 10 frames, there will be an initial frame followed
by 10 Delta's (i.e ILBMS containing ANHD and DLTA chunks).  Applying the
first Delta to the initial frame generates the second frame, applying the
second Delta to the second frame generates the third frame, etc.  Applying
the last Delta thus brings back the first frame.

The DLTA chunk begins with 16 LONG plane offets, of which DPaint only uses
the first 6 (at most).  These plane offsets are either the offset (in
bytes) from the beginning of the DLTA chunk to the data for the
corresponding plane, or Zero, if there was no change in that plane.  Thus
the first plane offset is either 0 or 64.
```

1.54 ANIM.brush.doc / RIFF

```
The following description of the method is based on Gary Bonham's
rewording of Jim Kent's RIFF documentation.

Compression/decompression is performed on a plane-by-plane
basis.

Each byte-column of the bitplane is compressed separately.  A
320x200 bitplane would have 40 columns of 200 bytes each.  In
general, the bitplanes are always an even number of bytes wide,
so for instance a 17x20 bitplane would have 4 columns of 20
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 kinds, and followed by a varying amount of data depending
on which kind:

1. SKIP - 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. DUMP - this is a byte with the hi bit set. The hi bit is
   masked off and the remainder is a count of the number of
   bytes of data to XOR directly. It is followed by the
   bytes to copy.

3. RUN - this is a 0 byte followed by a count byte, followed
   by a byte value to repeat "count" times, XOR'ing it into
   the destination.

Bear in mind that the data is compressed vertically rather than
horizontally, so to get to the next byte in the destination you
add the number of bytes per row instead of one.

The Format of DLTA chunks is as described in section 2.2.2 of the Anim
Spec. The encoding for type 5 is described in section 2.2.3 of the Anim
Spec.

1.55 A / IFF Third Party Public Form and Chunk Specification / ANIM.doc

Cel animation form

A N I M
An IFF Format For CEL Animations

Revision date: 4 May 1988
Prepared by:
SPARTA Inc.
23041 de la Carlota
Laguna Hills, Calif 92653
(714) 768-8161
contact: Gary Bonham

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

Introduction
Chunk Formats

1.56 ANIM.doc / 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.

1.57 Introduction / 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.

1.58 Introduction / 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 paragraphs 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

1.59 Introduction Recording ANIMs / 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.

1.60 Introduction / Recording ANIMs / 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 in section 2 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.

1.61 Introduction / Recording ANIMs / 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.

1.62 Introduction / Recording ANIMs / 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 in the Format for Method 4 sub-section within the Delta Chunk section.

1.63 Introduction / Recording ANIMs / 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.

Details on this method are contained in the Format for Method 5 sub-section within the Delta Chunk section.

1.64 Introduction / 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 double- buffer 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).
```

1.65 ANIM.doc / Chunk Formats

```
 ANHD Chunk
 DLTA ChunK
```

1.66 Chunk Formats / 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
                    =6 Stereo op 5 (third party)
                    =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 representd 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 (unused so far - 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.
```

1.67 Chunk Formats / 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. At present there are two basic formats for the overall structure of this chunk.

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

1.68 Chunk Formats / DLTA Chunk / 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.

1.69 Chunk Formats / DLTA Chunk / 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 theoretical 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.

1.70 Chunk Formats / DLTA Chunk / 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!
```

1.71 A / IFF Third Party Public Form and Chunk Specification / DR2D.doc

```
2-D Object standard format

FORM DR2D

Description by Ross Cunniff and John Orr

A standard IFF FORM to describe 2D drawings has been sorely needed for a
long time.  Several commercial drawing packages have been available for
some time but none has established its file format as the Amiga standard.
The absence of a 2D drawing standard hinders the development of
applications that use 2D drawings as it forces each application to
understand several private standards instead of a single one.  Without a
standard, data exchange for both the developer and the user is difficult,
if not impossible.

The DR2D FORM fills this void.  This FORM was developed by Taliesin, Inc.
for use as the native file format for their two-dimensional structured
drawing package, ProVector.  Saxon Industries and Soft Logik Publishing
Corporation are planning to support this new FORM in the near future.

Many of the values stored in the DR2D FORM are stored as IEEE single
precision floating point numbers.  These numbers consist of 32 bits,
arranged as follows:


 ________________________________________________________________________
| s e e e e e e e | e m m m m m m m | m m m m m m m m | m m m m m m m m |
 ------------------------------------------------------------------------
  31            24  23            16  15             8   7             0


where:

    s   is the sign of the number where 1 is negative and 0 is
          positive.
    e   is the 8 bit exponent in excess 127 form.  This number
          is the power of two to which the mantissa is raised
          (Excess 127 form means that 127 is added to the
          exponent before packing it into the IEEE number.)
    m   is the 23 bit mantissa.  It ranges from 1.0000000 to
          1.999999..., where the leading base-ten one is
```

```
          assumed.

An IEEE single precision with the value of 0.0000000 has all its bits
cleared.

 The DR2D Chunks
 Simple DR2D Example
 OFNT Form
```

1.72 DR2D.doc / The DR2D Chunks

```
FORM (0x464F524D)   /* All drawings are a FORM */

    struct FORMstruct {
        ULONG       ID;             /* DR2D */
        ULONG       Size;
    };

DR2D (0x44523244)  /* ID of 2D drawing */

The DR2D chunks are broken up into three groups: the global drawing
attribute chunks, the object attribute chunks, and the object chunks. The
global drawing attribute chunks describe elements of a 2D drawing that are
common to many objects in the drawing.  Document preferences, palette
information, and custom fill patterns are typical document-wide settings
defined in global drawing attribute chunks. The object attribute chunks
are used to set certain properties of the object chunk(s) that follows the
object attribute chunk.  The current fill pattern, dash pattern, and line
color are all set using an object attribute chunk.  Object chunks describe
the actual DR2D drawing. Polygons, text, and bitmaps are found in these
chunks.

 Global Drawing Attribute Chunk
 Object Attribute Chunks
 The Object Chunks
```

1.73 The DR2D Chunks / The Global Drawing Attribute Chunks

```
The following chunks describe global attributes of a DR2D document.

 DRHD       CMAP       DASH       FILL
 PPRF       FONS       AROW       LAYR
```

1.74 The Global Drawing Attribute Chunks / DRHD

```
DRHD (0x44524844)   /* Drawing header */

The DRHD chunk contains the upper left and lower right extremes of the
document in (X, Y) coordinates.  This chunk is required and should only
appear once in a document in the outermost layer of the DR2D file (DR2Ds
```

```
can be nested).

  struct DRHDstruct {
      ULONG ID;
      ULONG Size;       /* Always 16 */
      IEEE  XLeft, YTop,
      XRight, YBot;
  };
```

The point (XLeft,YTop) is the upper left corner of the project and the point (XRight,YBot) is its lower right corner. These coordinates not only supply the size and position of the document in a coordinate system, they also supply the project's orientation. If XLeft < XRight, the X-axis increases toward the right. If YTop < YBot, the Y-axis increases toward the bottom. Other combinations are possible; for example in Cartesian coordinates, XLeft would be less than XRight but YTop would be greater than YBot.

1.75 The Global Drawing Attribute Chunks / PPRF

```
PPRF (0x50505249)     /* Page preferences */
```

The PPRF chunk contains preference settings for ProVector. Although this chunk is not required, its use is encouraged because it contains some important environment information.

```
  struct PPRFstruct {
      ULONG ID;
      ULONG Size;
      char  Prefs[Size];
  };
```

DR2D stores preferences as a concatenation of several null-terminated strings, in the Prefs[] array. The strings can appear in any order. The currently supported strings are:

```
    Units=<unit-type>
    Portrait=<boolean>
    PageType=<page-type>
    GridSize=<number>

where:
    <unit-type>   is either Inch, Cm, or Pica
    <boolean>     is either True or False
    <page-type>   is either Standard, Legal, B4, B5, A3,
          A4, A5, or Custom
    <number>      is a floating-point number
```

The DR2D FORM does not require this chunk to explicitly state all the possible preferences. In the absence of any particular preference string, a DR2D reader should fall back on the default value. The defaults are:

```
    Units=Inch
    Portrait=True
    PageType=Standard
```

```
    GridSize=1.0
```

1.76 The Global Drawing Attribute Chunks / CMAP

```
CMAP (0x434D4150)     /* Color map (Same as ILBM CMAP) */
```

This chunk is identical to the ILBM CMAP chunk as described in the IFF ILBM documentation.

```
    struct CMAPstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       ColorMap[Size];
    };
```

ColorMap is an array of 24-bit RGB color values. The 24-bit value is spread across three bytes, the first of which contains the red intensity, the next contains the green intensity, and the third contains the blue intensity. Because DR2D stores its colors with 24-bit accuracy, DR2D readers must not make the mistake that some ILBM readers do in assuming the CMAP chunk colors correspond directly to Amiga color registers.

1.77 The Global Drawing Attribute Chunks / FONS

```
FONS (0x464F4E53)     /* Font chunk (Same as FTXT FONS chunk) */
```

The FONS chunk contains information about a font used in the DR2D FORM. ProVector does not include support for Amiga fonts. Instead, ProVector uses fonts defined in the OFNT FORM which is documented later in this article.

```
    struct FONSstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       FontID;         /* ID the font is referenced by */
        UBYTE       Pad1;           /* Always 0 */
        UBYTE       Proportional;   /* Is it proportional? */
        UBYTE       Serif;          /* does it have serifs? */
        CHAR        Name[Size-4];   /* The name of the font */
    };
```

The UBYTE FontID field is the number DR2D assigns to this font. References to this font by other DR2D chunks are made using this number. The Proportional and Serif fields indicate properties of this font. Specifically, Proportional indicates if this font is proportional, and Serif indicates if this font has serifs. These two options were created to allow for font substitution in case the specified font is not available. They are set according to these values:

```
  0 The DR2D writer didn't know if this font is
      proportional/has serifs.
  1 No, this font is not proportional/does not have
```

```
      serifs.
  2 Yes, this font is proportional/does have serifs.

The last field, Name[], is a NULL terminated string containing the
name of the font.
```

1.78 The Global Drawing Attribute Chunks / DASH

```
DASH (0x44415348)     /* Line dash pattern for edges */

This chunk describes the on-off dash pattern associated with a line.

    struct DASHstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      DashID;             /* ID of the dash pattern */
        USHORT      NumDashes;          /* Should always be even */
        IEEE        Dashes[NumDashes];  /* On-off pattern */
    };

DashID is the number assigned to this specific dash pattern. References to
this dash pattern by other DR2D chunks are made using this number.

The Dashes[] array contains the actual dash pattern.  The first number in
the array (element 0) is the length of the ''on'' portion of the pattern.
The second number (element 1) specifies the ''off'' portion of the
pattern.  If there are more entries in the Dashes array, the pattern will
continue.  Even-index elements specify the length of an ''on'' span, while
odd-index elements specify the length of an ''off'' span.  There must be
an even number of entries.  These lengths are not in the same units as
specified in the PPRF chunk, but are multiples of the line width, so a
line of width 2.5 and a dash pattern of 1.0, 2.0 would have an ''on'' span
of length 1.0 x 2.5 = 2.5 followed by an ''off'' span of length 2.0 x 2.5
= 5.  The following figure shows several dash pattern examples.  Notice
that for lines longer than the dash pattern, the pattern repeats.

 Figure 1 - Dash Patterns

By convention, DashID 0 is reserved to mean 'No line pattern at all', i.e.
the edges are invisible.  This DASH pattern should not be defined by a
DR2D DASH chunk.  Again by convention, a NumDashes of 0 means that the
line is solid.
```

1.79 The Global Drawing Attribute Chunks / AROW

```
AROW (0x41524F57)     /* An arrow-head pattern */

The AROW chunk describes an arrowhead pattern.  DR2D open polygons (OPLY)
can have arrowheads attached to their endpoints.  See the description of
the OPLY chunk later in this article for more information on the OPLY
chunk.
```

```
    #define ARROW_FIRST  0x01 /* Draw an arrow on the OPLY's first point */
    #define ARROW_LAST   0x02 /* Draw an arrow on the OPLY's last point */

    struct AROWstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       Flags;          /* Flags, from ARROW_*, above */
        UBYTE       Pad0;           /* Should always 0 */
        USHORT      ArrowID;        /* Name of the arrow head */
        USHORT      NumPoints;
        IEEE        ArrowPoints[NumPoints*2];
    };
```

The Flags field specifies which end(s) of an OPLY to place an arrowhead based on the #defines above. ArrowID is the number by which an OPLY will reference this arrowhead pattern.

The coordinates in the array ArrowPoints[] define the arrowhead's shape. These points form a closed polygon. See the section on the OPLY/CPLY object chunks for a descriptionof how DR2D defines shapes. The arrowhead is drawn in the same coordinate system relative to the endpoint of the OPLY the arrowhead is attached to. The arrowhead's origin (0,0) coincides with the OPLY's endpoint. DR2D assumes that the arrowhead represented in the AROW chunk is pointing to the right so the proper rotation can be applied to the arrowhead. The arrow is filled according to the current fill pattern set in the ATTR object attribute chunk.

1.80 The Global Drawing Attribute Chunks / FILL

```
FILL (0x46494C4C)     /* Object-oriented fill pattern */
```

The FILL chunk defines a fill pattern. This chunk is only valid inside nested DR2D FORMs. The GRUP object chunk section of this article contans an example of the FILL chunk.

```
    struct FILLstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      FillID;                 /* ID of the fill */
    };
```

FillID is the number by which the ATTR object attribute chunk references fill patterns. The FILL chunk must be the first chunk inside a nested DR2D FORM. A FILL is followed by one DR2D object plus any of the object attribute chunks (ATTR, BBOX) associated with the object.

Figure 2 - Fill Patterns

DR2D makes a ''tile'' out of the fill pattern, giving it a virtual bounding box based on the extreme X and Y values of the FILL's object (Fig. A). The bounding box shown in Fig. A surrounding the pattern (the two ellipses) is invisible to the user. In concept, this rectangle is pasted on the page left to right, top to bottom like floor tiles (Fig. B). Again, the bounding boxes are not visible. The only portion of this tiled pattern that is visible is the part that overlaps the object (Fig. C)

being filled. The object's path is called a clipping path, as it ''clips'' its shape from the tiled pattern (Fig. D). Note that the fill is only masked on top of underlying objects, so any ''holes'' in the pattern will act as a window, leaving visible underlying objects.

1.81 The Global Drawing Attribute Chunks / LAYR

```
LAYR (0x4C415952)     /* Define a layer */
```

A DR2D project is broken up into one or more layers. Each DR2D object is in one of these layers. Layers provide several useful features. Any particular layer can be ''turned off'', so that the objects in the layer are not displayed. This eliminates the unnecessary display of objects not currently needed on the screen. Also, the user can lock a layer to protect the layer's objects from accidental changes.

```
  struct LAYRstruct {
      ULONG    ID;
      ULONG    Size;
      USHORT   LayerID;      /* ID of the layer */
      char     LayerName[16];    /* Null terminated and padded */
      UBYTE    Flags;      /* Flags, from LF_*, below */
      UBYTE    Pad0;       /* Always 0 */
  };
```

LayerID is the number assigned to this layer. As the field's name indicates, LayerName[] is the NULL terminated name of the layer. Flags is a bit field who's bits are set according to the #defines below:

```
  #define LF_ACTIVE       0x01    /* Active for editing */
  #define LF_DISPLAYED    0x02    /* Displayed on the screen */
```

If the LF_ACTIVE bit is set, this layer is unlocked. A set LF_DISPLAYED bit indicates that this layer is currently visible on the screen. A cleared LF_DISPLAYED bit implies that LF_ACTIVE is not set. The reason for this is to keep the user from accidentally editing layers that are invisible.

1.82 The DR2D Chunks / The Object Attribute Chunks

```
 ATTR
 BBOX
 XTRN
```

1.83 The Object Attribute Chunks / ATTR

```
ATTR (0x41545452)     /* Object attributes */
```

The ATTR chunk sets various attributes for the objects that follow it. The attributes stay in effect until the next ATTR changes the attributes, or

the enclosing FORM ends, whichever comes first.

```
    /* Various fill types */
    #define FT_NONE         0    /* No fill                  */
    #define FT_COLOR        1    /* Fill with color from palette */
    #define FT_OBJECTS      2    /* Fill with tiled objects */


    struct ATTRstruct {
        ULONG       ID;
        ULONG       Size;
        UBYTE       FillType;    /* One of FT_*, above      */
        UBYTE       JoinType;    /* One of JT_*, below      */
        UBYTE       DashPattern; /* ID of edge dash pattern */
        UBYTE       ArrowHead;   /* ID of arrowhead to use  */
        USHORT      FillValue;   /* Color or object with which to fill */
        USHORT      EdgeValue;   /* Edge color index        */
        USHORT      WhichLayer;  /* ID of layer it's in     */
        IEEE        EdgeThick;   /* Line width              */
    };
```

FillType specifies what kind of fill to use on this ATTR chunk's objects. A value of FT_NONE means that this ATTR chunk's objects are not filled. FT_COLOR indicates that the objects should be filled in with a color. That color's ID (from the CMAP chunk) is stored in the FillValue field. If FillType is equal to FT_OBJECTS, FillValue contains the ID of a fill pattern defined in a FILL chunk.

JoinType determines which style of line join to use when connecting the edges of line segments. The field contains one of these four values:

```
    /* Join types */
    #define JT_NONE         0       /* Don't do line joins */
    #define JT_MITER        1       /* Mitered join */
    #define JT_BEVEL        2       /* Beveled join */
    #define JT_ROUND        3       /* Round join */
```

DashPattern and ArrowHead contain the ID of the dash pattern and arrow head for this ATTR's objects. A DashPattern of zero means that there is no dash pattern so lines will be invisible. If ArrowHead is 0, OPLYs have no arrow head. EdgeValue is the color of the line segments. WhichLayer contains the ID of the layer this ATTR's objects are in. EdgeThick is the width of this ATTR's line segments.

1.84 The Object Attribute Chunks / BBOX

```
BBOX (0x42424F48)     /* Bounding box of next object in FORM */
```

The BBOX chunk supplies the dimensions and position of a bounding box surrounding the DR2D object that follows this chunk in the FORM. A BBOX chunk can apply to a FILL or AROW as well as a DR2D object. The BBOX chunk appears just before its DR2D object, FILL, or AROW chunk.

```
   struct BBOXstruct {
       ULONG       ID;
```

```
       ULONG       Size;
       IEEE                XMin, YMin,     /* Bounding box of obj. */
                           XMax, YMax;     /* including line width */
   };
```

In a Cartesian coordinate system, the point (XMin, YMin) is the coordinate of the lower left hand corner of the bounding box and (XMax, YMax) is the upper right. These coordinates take into consideration the width of the lines making up the bounding box.

1.85 The Object Attribute Chunks / XTRN

```
XTRN (0x5854524E)     /* Externally controlled object */
```

The XTRN chunk was created primarily to allow ProVector to link DR2D objects to ARexx functions.

```
    struct XTRNstruct {
        ULONG   ID;
        ULONG   Size;
        short   ApplCallBacks;            /* From #defines, below */
        short   ApplNameLength;
        char    ApplName[ApplNameLength]; /* Name of ARexx func to call */
    };
```

ApplName[] contains the name of the ARexx script ProVector calls when the user manipulates the object in some way. The ApplCallBacks field specifies the particular action that triggers calling the ARexx script according to the #defines listed below.

```
    /* Flags for ARexx script callbacks */
    #define    X_CLONE     0x0001    /* The object has been cloned */
    #define    X_MOVE      0x0002    /* The object has been moved */
    #define    X_ROTATE    0x0004    /* The object has been rotated */
    #define    X_RESIZE    0x0008    /* The object has been resized */
    #define    X_CHANGE    0x0010    /* An attribute (see ATTR) of the
                                        object has changed */
    #define    X_DELETE    0x0020    /* The object has been deleted */
    #define    X_CUT       0x0040    /* The object has been deleted, but
                                        stored in the clipboard */
    #define    X_COPY      0x0080    /* The object has been copied to the
                                        clipboard */
    #define    X_UNGROUP   0x0100    /* The object has been ungrouped */
```

For example, given the XTRN object:

```
    FORM xxxx DR2D {
            XTRN xxxx { X_RESIZE | X_MOVE, 10, "Dimension" }
            ATTR xxxx { 0, 0, 1, 0, 0, 0, 0.0 }
            FORM xxxx DR2D {
                    GRUP xxxx { 2 }
                    STXT xxxx { 0, 0.5, 1.0, 6.0, 5.0, 0.0, 4, "3.0" }
                    OPLY xxxx { 2, { 5.5, 5.5, 8.5, 5.5 } }
            }
    }
```

```
ProVector would call the ARexx script named Dimension if the user resized
or moved this object.  What exactly ProVector sends depends upon what the
user does to the object.  The following list shows what string(s)
ProVector sends according to which flag(s) are set.  The parameters are
described below.

    X_CLONE     ''appl CLONE objID dx dy''
    X_MOVE      ''appl MOVE objID dx dy''
    X_ROTATE    ''appl ROTATE objID cx cy angle''
    X_RESIZE    ''appl RESIZE objID cx cy sx sy''
    X_CHANGE    ''appl CHANGE objID et ev ft fv ew jt fn''
    X_DELETE    ''appl DELETE objID''
    X_CUT       ''appl CUT objID''
    X_COPY      ''appl COPY objID''
    X_UNGROUP   ''appl UNGROUP objID''

 where:
    appl is the name of the ARexx script
    CLONE, MOVE, ROTATE, RESIZE, etc. are literal strings
    objID is the object ID that ProVector assigns to this object
    (dx, dy) is the position offset of the CLONE or MOVE
    (cx, cy) is the point around which the object is rotated or resized
    angle is the angle (in degrees) the object is rotated
    sx and sy are the scaling factors in the horizontal and
      vertical directions, respectively.
    et is the edge type (the dash pattern index)
    ev is the edge value (the edge color index)
    ft is the fill type
    fv is the fill index
    ew is the edge weight
    jt is the join type
    fn is the font name

The X_CHANGE message reflects changes to the attributes found in the ATTR
chunk.

If the user resized the XTRN object shown above by factor of 2, ProVector
would call the ARexx script Dimension like this:

    Dimension RESIZE 1985427 7.0 4.75 2.0 2.0
```

1.86 The DR2D Chunks / The Object Chunks

```
The following chunks define the objects available in the DR2D FORM.

 VBM
 CPLY, OPLY
 GRUP
 STXT
 TPTH
```

1.87 The Object Chunks / VBM

```
VBM  (0x56424D20)     /* Virtual BitMap */
```

The VBM chunk contains the position, dimensions, and file name of an ILBM image.

```
    struct VBMstruct {
        IEEE        XPos, YPos,     /* Virtual coords */
                    XSize, YSize,   /* Virtual size */
                    Rotation;       /* in degrees */
        USHORT      PathLen;        /* Length of dir path */
        char        Path[PathLen];  /* Null-terminated path of file */
    };
```

The coordinate (XPos, YPos) is the position of the upper left hand corner of the bitmap and the XSize and YSize fields supply the x and y dimensions to which the image should be scaled. Rotation tells how many degrees to rotate the ILBM around its upper left hand corner. ProVector does not currently support rotation of bitmaps and will ignore this value. Path contains the name of the ILBM file and may also contain a partial or full path to the file. DR2D readers should not assume the path is correct. The full path to an ILBM on one system may not match the path to the same ILBM on another system. If a DR2D reader cannot locate an ILBM file based on the full path name or the file name itself (looking in the current directory), it should ask the user where to find the image.

1.88 The Object Chunks / CPLY, OPLY

```
CPLY (0x43504C59)     /* Closed polygon */
OPLY (0x4F504C59)     /* Open polygon */
```

Polygons are the basic components of almost all 2D objects in the DR2D FORM. Lines, squares, circles, and arcs are all examples of DR2D polygons. There are two types of DR2D polygons, the open polygon (OPLY) and the closed polygon (CPLY). The difference between a closed and open polygon is that the computer adds a line segment connecting the endpoints of a closed polygon so that it is a continuous path. An open polygon's endpoints do not have to meet, like the endpoints of a line segment.

```
    struct POLYstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      NumPoints;
        IEEE        PolyPoints[2*NumPoints];
    };
```

The NumPoints field contains the number of points in the polygon and the PolyPoints array contains the (X, Y) coordinates of the points of the non-curved parts of polygons. The even index elements are X coordinates and the odd index elements are Y coordinates.

[Figure 3 - Bezier Curves

DR2D uses Bezier cubic sections, or cubic splines, to describe curves in polygons. A set of four coordinates (P1 through P4) defines the shape of a cubic spline. The first coordinate (P1) is the point where the curve begins. The line from the first to the second coordinate (P1 to P2) is tangent to the curve at the first point. The line from P3 to P4 is tangent to the cubic section, where it ends at P4.

The coordinates describing the cubic section are stored in the PolyPoints[] array with the coordinates of the normal points. DR2D inserts an indicator point before a set of cubic section points to differentiate a normal point from the points that describe a curve. An indicator point has an X value of 0xFFFFFFFF. The indicator point's Y value is a bit field. If this bit field's low-order bit is set, the points that follow the indicator point make up a cubic section.

The second lowest order bit in the indicator point's bit field is the MOVETO flag. If this bit is set, the point (or set of cubic section points) starts a new polygon, or subpolygon. This subpolygon will appear to be completely separate from other polygons but there is an important connection between a polygon and its subpolygon. Subpolygons make it possible to create holes in polygons. An example of a polygon with a hole is the letter ''O''. The ''O'' is a filled circular polygon with a smaller circular polygon within it. The reason the inner polygon isn't covered up when the outer polygon is filled is that DR2D fills are done using the even-odd rule.

The even-odd rule determines if a point is ''inside'' a polygon by drawing a ray outward from that point and counting the number of path segments the ray crosses. If the number is even, the point is outside the object and shouldn't be filled. Conversely, an odd number of crossings means the point is inside and should be filled. DR2D only applies the even-odd rule to a polygon and its subpolygons, so no other objects are considered in the calculations.

Taliesin, Inc. supplied the following algorithm to illustrate the format of DR2D polygons. OPLYs, CPLYs, AROWs, and ProVector's outline fonts all use the same format:

```
typedef union {
    IEEE num;
    LONG bits;
} Coord;

#define INDICATOR       0xFFFFFFFF
#define IND_SPLINE      0x00000001
#define IND_MOVETO      0x00000002

/* A common pitfall in attempts to support DR2D has
        been to fail to recognize the case when an
        INDICATOR point indicates the following
        coordinate to be the first point of BOTH a
        Bezier cubic and a sub-polygon, ie. the
        value of the flag = (IND_CURVE | IND_MOVETO) */

Coord   Temp0, Temp1;
int     FirstPoint, i, Increment;
```

```
/* Initialize the path */
NewPath();
FirstPoint = 1;

/* Draw the path */
i = 0;
while( i < NumPoints ) {
    Temp0.num = PolyPoints[2*i];    Temp1.num = PolyPoints[2*i + 1];
    if( Temp0.bits == INDICATOR ) {
        /* Increment past the indicator */
        Increment = 1;
        if( Temp1.bits & IND_MOVETO ) {
            /* Close and fill, if appropriate */
            if( ID == CPLY ) {
                FillPath();
            }
            else {
                StrokePath();
            }

            /* Set up the new path */
            NewPath();
            FirstPoint = 1;
        }
        if( Temp1.bits & IND_CURVE ) {
            /* The next 4 points are Bezier cubic control points */
            if( FirstPoint )
                MoveTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            else
                LineTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            CurveTo(    PolyPoints[2*i + 4], PolyPoints[2*i + 5],
                        PolyPoints[2*i + 6], PolyPoints[2*i + 7],
                        PolyPoints[2*i + 8], PolyPoints[2*i + 9] );
            FirstPoint = 0;
            /* Increment past the control points */
            Increment += 4;
        }
    }
    else {
        if( FirstPoint )
            MoveTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        else
            LineTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        FirstPoint = 0;

        /* Increment past the last endpoint */
        Increment = 1;
    }

    /* Add the increment */
    i += Increment;
}

/* Close the last path */
if( ID == CPLY ) {
    FillPath();
}
```

```
    else {
        StrokePath();
    }
```

1.89 The Object Chunks / GRUP

```
GRUP (0x47525550)     /* Group */
```

The GRUP chunk combines several DR2D objects into one. This chunk is only valid inside nested DR2D FORMs, and must be the first chunk in the FORM.

```
    struct GROUPstruct {
        ULONG       ID;
        ULONG       Size;
        USHORT      NumObjs;
    };
```

The NumObjs field contains the number of objects contained in this group. Note that the layer of the GRUP FORM overrides the layer of objects within the GRUP. The following example illustrates the layout of the GRUP (and FILL) chunk.

```
    FORM { DR2D              /* Top-level drawing... */
             DRHD { ... }    /* Confirmed by presence of DRHD chunk */
             CMAP { ... }    /* Various other things... */
             FONS { ... }
             FORM { DR2D             /* A nested form... */
                     FILL { 1 }      /* Ah!  The fill-pattern table */
                     CPLY { ... }    /* with only 1 object */
             }
             FORM { DR2D             /* Yet another nested form */
                     GRUP { ..., 3 } /* Ah! A group of 3 objects */
                     TEXT { ... }
                     CPLY { ... }
                     OPLY { ... }
             }
             FORM { DR2D             /* Still another nested form */
                     GRUP { ..., 2 } /* A GRUP with 2 objects */
                     OPLY { ... }
                     TEXT { ... }
             }
     }
```

1.90 The Object Chunks / STXT

```
STXT (0x53545854)       /* Simple text */
```

The STXT chunk contains a text string along with some information on how and where to render the text.

```
    struct STXTstruct {
        ULONG       ID;
```

```
        ULONG       Size;
        UBYTE       Pad0;           /* Always 0 (for future expansion) */
        UBYTE       WhichFont;      /* Which font to use */
        IEEE        CharW, CharH,   /* W/H of an individual char */
                    BaseX, BaseY,   /* Start of baseline */
                    Rotation;       /* Angle of text (in degrees) */
        USHORT      NumChars;
        char        TextChars[NumChars];
    };
```

The text string is in the character array, TextChars[]. The ID of the font used to render the text is WhichFont. The font's ID is set in a FONS chunk. The starting point of the baseline of the text is (BaseX, BaseY). This is the point around which the text is rotated. If the Rotation field is zero (degrees), the text's baseline will originate at (BaseX, BaseY) and move to the right. CharW and CharH are used to scale the text after rotation. CharW is the average character width and CharH is the average character height. The CharW/H fields are comparable to an X and Y font size.

1.91 The Object Chunks / TPTH

```
TPTH (0x54505448)     /* A text string along a path */
```

This chunk defines a path (polygon) and supplies a string to render along the edge of the path.

```
    struct TPTHstruct {
        ULONG   ID;
        ULONG   Size;
        UBYTE   Justification;      /* see defines, below */
        UBYTE   WhichFont;          /* Which font to use */
        IEEE    CharW, CharH;       /* W/H of an individual char    */
        USHORT  NumChars;           /* Number of chars in the string */
        USHORT  NumPoints;          /* Number of points in the path */
        char    TextChars[NumChars];/* PAD TO EVEN #! */
        IEEE    Path[2*NumPoints];  /* The path on which the text lies */
    };
```

WhichFont contains the ID of the font used to render the text. Justification controls how the text is justified on the line. Justification can be one of the following values:

```
    #define J_LEFT          0x00    /* Left justified */
    #define J_RIGHT         0x01    /* Right justified */
    #define J_CENTER        0x02    /* Center text */
    #define J_SPREAD        0x03    /* Spread text across path */
```

CharW and CharH are the average width and height of the font characters and are akin to X and Y font sizes, respectively. A negative FontH implies that the font is upsidedown. Note that CharW must not be negative. NumChars is the number of characters in the TextChars[] string, the string containing the text to be rendered. NumPoints is the number of points in the Path[] array. Path[] is the path along which the text is rendered. The path itself is not rendered. The points of Path[] are in

the same format as the points of a DR2D polygon.

1.92 DR2D.doc / A Simple DR2D Example

Here is a (symbolic) DR2D FORM:

```
 FORM { DR2D
        DRHD 16 { 0.0, 0.0, 10.0, 8.0 }
        CMAP  6 { 0,0,0, 255,255,255 }
        FONS  9 { 1, 0, 1, 0, "Roman" } 0
        DASH 12 { 1, 2, {1.0, 1.0} }
        ATTR 14 { 0, 0, 1, 0, 0, 0, 0, 0.0 }
        BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
        FORM { DR2D
               GRUP  2 { 2 }
               BBOX 16 { 3.0, 4.0, 7.0, 5.0 }
               STXT 36 { 0,1, 0.5, 1.0, 3.0, 5.0, 0.0, 12, "Hello, World" }
               BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
               OPLY 42 { 5, {2.0,2.0, 8.0,2.0, 8.0,6.0, 2.0,6.0, 2.0,2.0 }
        }
 }
```

Figure 4 - Simple DR2D Drawing

1.93 DR2D.doc / The OFNT FORM

```
 OFNT
 OFHD
 KERN
 CHDF
```

1.94 DR2D.doc / OFNT

```
OFNT  (0x4F464E54)    /* ID of outline font file */
```

ProVector's outline fonts are stored in an IFF FORM called OFNT. This IFF is a separate file from a DR2D. DR2D's FONS chunk refers only to fonts defined in the OFNT form.

1.95 DR2D.doc / OFHD

```
OFHD  (0x4F464844)    /* ID of OutlineFontHeaDer */
```

This chunk contains some basic information on the font.

```
    struct OFHDstruct {
        char   FontName[32];   /* Font name, null padded */
```

```
        short  FontAttrs;      /* See FA_*, below */
        IEEE   FontTop,        /* Typical height above baseline */
               FontBot,        /* Typical descent below baseline */
               FontWidth;      /* Typical width, i.e. of the letter O */
    };

    #define FA_BOLD         0x0001
    #define FA_OBLIQUE      0x0002
    #define FA_SERIF        0x0004
```

The FontName field is a NULL terminated string containing the name of this font. FontAttrs is a bit field with flags for several font attributes. The flags, as defined above, are bold, oblique, and serif. The unused higher order bits are reserved for later use. The other fields describe the average dimensions of the characters in this font. FontTop is the average height above the baseline, FontBot is the average descent below the baseline, and FontWidth is the average character width.

1.96 DR2D.doc / KERN

```
KERN  (0x4B45524C)    /* Kerning pair */
```

The KERN chunk describes a kerning pair. A kerning pair sets the distance between a specific pair of characters.

```
    struct KERNstruct {
        short   Ch1, Ch2;   /* The pair to kern (allows for 16 bits...) */
        IEEE    XDisplace,  /* Amount to displace -left +right */
                YDisplace;  /* Amount to displace -down +up */
    };
```

The Ch1 and Ch2 fields contain the pair of characters to kern. These characters are typically stored as ASCII codes. Notice that OFNT stores the characters as a 16-bit value. Normally, characters are stored as 8-bit values. The wary programmer will be sure to cast assigns properly to avoid problems with assigning an 8-bit value to a 16-bit variable. The remaining fields, XDisplace and YDisplace, supply the baseline shift from Ch1 to Ch2.

1.97 DR2D.doc / CHDF

```
CHDF  (0x43484446)    /* Character definition */
```

This chunk defines the shape of ProVector's outline fonts.

```
    struct CHDFstruct {
        short   Ch;         /* The character we're defining (ASCII) */
        short   NumPoints;  /* The number of points in the definition */
        IEEE    XWidth,     /* Position for next char on baseline - X */
                YWidth;     /* Position for next char on baseline - Y */
     /* IEEE    Points[2*NumPoints]*/       /* The actual points */
    };
```

```
    #define INDICATOR   0xFFFFFFFF  /* If X == INDICATOR, Y is an action */
    #define IND_SPLINE  0x00000001  /* Next 4 pts are spline control pts */
    #define IND_MOVETO  0x00000002    /* Start new subpoly */
    #define IND_STROKE  0x00000004  /* Stroke previous path */
    #define IND_FILL    0x00000008  /* Fill previous path */
```

Ch is the value (normally ASCII) of the character outline this chunk defines. Like Ch1 and Ch2 in the KERN chunk, Ch is stored as a 16-bit value. (XWidth,YWidth) is the offset to the baseline for the following character. OFNT outlines are defined using the same method used to define DR2D's polygons (see the description of OPLY/CPLY for details).

Because the OFNT FORM does not have an ATTR chunk, it needed an alternative to make fills and strokes possible. There are two extra bits used in font indicator points not found in polygon indicator points, the IND_STROKE and IND_FILL bits (see defines above). These two defines describe how to render the current path when rendering fonts.

The current path remains invisible until the path is either filled and/or stroked. When the IND_FILL bit is set, the currently defined path is filled in with the current fill pattern (as specified in the current ATTR chunk). A set IND_STROKE bit indicates that the currently defined path itself should be rendered. The current ATTR's chunk dictates the width of the line, as well as several other attributes of the line. These two bits apply only to the OFNT FORM and should not be used in describing DR2D polygons.

1.98 A / IFF Third Party Public Form and Chunk Specification / FANT.doc

```
Fantavision movie format

FORM FANT

/*****************************************************************/
/*                                                               **
** - FantForm.h                                                  **
**                                                               **
**     This is the IFF movie format for Amiga Fantavision.       **
**                                                               **
**     (c) Copyright 1988 Broderbund Software                    **
**                                                               **
**     - FORMAT FROZED May 5, 1988 -                             **
**                                                               **
**     Implemented by Steve Hales                                **
**                                                               **
** Overvue -                                                     **
**     This is a description of the format used for Amiga        **
**     Fantavision.  It assumes you have intimate knowledge of how   **
**     IFF-FORMs are constructed, layed out, and read.  This file    **
**     can be used as a header file.  This is fairly complete, but   **
**     I'm sure there are a few things missing.                  **
**                                                               **
**     I can be reached in the following ways:                   **
**       UseNet:   Steve_A_Hales@cup.portal.com  OR              **
```

```
**                 sun!cup.portal.com!Steve_A_Hales                 **
**                                                                  **
**       US Mail:  882 Hagemann Drive                               **
**                 Livermore, CA, 94550-2420                        **
**                                                                  **
**       Phone:    (415) 449-5297                                   **
**                                                                  **
**     NOTE:  I cannot, by contract, give out any code to load or   **
**            play Fantavision movies.  If that is want you want    **
**            then you will need to contact Broderbund Software     **
**            directly.  Their number is (415) 492-3200.            **
**                                                                  **
** Enjoy!  Aloha.                                                   **
**                                                                  */
/******************************************************************/

 Misc Fantavision Structures        Polygon Types
 Frame Opcodes                      Fantavision Movie Header
 Frame Modes                        Fantavision Frame Info
 Fantavision FORM Defines           Fantavision Polygon Info
 Polygon Modes                      Fantavision High-level IFF Format

 Notes
```

1.99 FANT.doc / Misc Fantavision Structures

```
typedef struct Rect
{
   int left, top, right, bottom;
};

typedef struct Point
{
   int h, v;
};
```

1.100 FANT.doc / Frame opcodes

```
#define opNEXT     0  /* go on to next frame */
#define opREPEAT   1  /* repeat seqnce starting at frame Parm rep1 times */
#define opGOTO     2  /* goto frame Parm */
```

1.101 FANT.doc / Frame modes

```
#define fNORMAL    0x0000  /* redraw every frame */
#define fTRACE     0x0001  /* draw into both paged screens */
#define fLIGHTNING 0x0020  /* don't erase background */
```

1.102 FANT.doc / Fantavision FORM defines

```
#define ID_FANT    'FANT'  /* FORM type */
#define ID_FHDR    'FHDR'  /* Movie Header */
#define ID_FRAM    'FRAM'  /* Format info for a Frame */
#define ID_POLY    'POLY'  /* Format info for a Polygon */
#define ID_CSTR    'CSTR'  /* \0 terminated string */
```

1.103 FANT.doc / Polygon modes

```
#define pTYPEMASK  0x00FF  /* type mask to get just type of poly */
#define pSELECT    0x8000  /* is object selected? */
#define pOUTLINE   0x4000  /* outlined polygon using DotModeSide to
                           ** determine when to not connect a line.
                           ** ex. 0 draws on all sides, 1 will draw on
                           ** everyother side, 2 will leave every second
                           ** side blank, 3 will every third side
                           ** blank, etc. */
#define pBACKDROP  0x2000  /* polygon will be dropped into the background
                           ** during animation. */
#define pMSKBITMAP 0x1000  /* bitmap has a mask */
```

1.104 FANT.doc / Polygon types

```
#define pDELETE    0x7000  /* object is a filler (deleted from display) */
#define pFILLED    0       /* filled polygon */
#define pLINE      1       /* not-connected line polygon */
#define pLINED     2       /* connected line polygon */
#define pTEXTBLOCK 3       /* text block to draw */
#define pCIRCLEDOT 4       /* draw circle dots at vertex's using
                           ** dotSize at size. */
#define pRECTDOT   5       /* draw square dots at vertex's using
                           ** dotSize at size. */
#define pBITMAPDOT 6       /* draw dots using a bitmap at vertex's using
                           ** BitMap. */
#define pBITMAP    7       /* draw just bitmap image */

/* These are used for the pTEXTBLOCK polygon type
*/
/* Text justification
*/
#define tLEFT      0
#define tCENTER    1
#define tRIGHT     2
/* Text style
*/
#define tNORMAL    (int)(FS_NORMAL)
#define tBOLD      (int)(FSF_BOLD)
#define tITALIC    (int)(FSF_ITALIC)
#define tUNDERLINE (int)(FSF_UNDERLINED)
#define tEXTENDED  (int)(FSF_EXTENDED)
```

1.105 FANT.doc / Fantavision movie header

```
/*
** This header defines how much RAM is needed, how many frames, and sounds
** in the movie.
*/

typedef struct FantHeader
{
   int PointsPerObj;     /* number of vertexs per object */
   int ObjsPerFrame;     /* number of objects per frame */
   int ScreenDepth;      /* 0 to 6, for number of bit planes */
   int ScreenWidth;      /* in pixels */
   int ScreenHeight;     /* in pixels */
   int BackColor;        /* background color palette number */
   long SizeOfMovie;     /* RAM Size of movie, expanded */
   int pad[30];          /* padding for expanding */
   int NumberOfFrames;
   int NumberOfSounds;
   int NumberOfBitMaps;
   int Background;       /* non-zero if first bitmap is a background */
   int SpeedOfMovie;     /* 100 is normal speed, 50 is half speed, etc */
   int pad[3];           /* expansion */
};
```

1.106 FANT.doc / Fantavision frame info

```
/*
**     Each frame has this structure defined.
*/
typedef struct FrameFormat
{
   int OpCode;           /* Frame opcode */
   long Parm;            /* contains frame number for opNEXT, opREPEAT */
   char Rep1, Rep2;      /* Rep1 is repeat counter, Rep2 is not used */
   int TweenRate;        /* number of tweens per frame */

   int ChannelIndex[2];  /* -3 stop sound is this channel
                         ** -2 modify current sound
                         ** -1 no sound for this channel
                         ** (all others) is an index into the sound
                         ** list.  Which sound to use.
                         */

   int NumberOfPolys;    /* number of polygons in this frame */
   int ColorPalette[32]; /* xRGB - format 4 bits per register */
   int Pan, Tilt;        /* 0 is centered, (+-) amounts are in pixels */
   int Modes;            /* Frame modes */
   int pad;              /* expansion */
};
```

1.107 FANT.doc / Fantavision polygon info

```
/*
**     Each polygon has this structure defined.
*/
typedef struct PolyFormat
{
   int NumberOfPoints;   /* how many vertexs for this polygon */
   int Type;             /* polygon type */
   int Color;            /* palette color number (see note 1) */
   Rect Bounds;          /* enclosing rectangle of polygon */
   int Depth;            /* polygon view depth (see note 2) */
   char DotModeSize;     /* in pixels, not larger than 40 */
   char DotModeSide;     /* determines outlining features */
   int OutlineColor;     /* palette color number for outline */
   int BitMapIndex;      /* if not -1,then bitmap index into bitmap list */
   int BMRealWidth;      /* in pixels */
   int BMRealHeight;
   int TextLength;       /* length of text for pTEXTBLOCK */
   int TextJust;
   char TextSize;        /* size in pixels */
   char TextStyle;
   long pad;             /* expansion */
   Point p[];            /* array of points defining vertexs */
};
```

1.108 FANT.doc / Fantavision high-level IFF format

```
/*
FORM FANT
    FHDR

    - background -
    FORM ILBM   if Background is non-zero
        BMHD
        BODY

    - bitmap list -
    NOTE:  If a bitmap has a mask, it will be compute during load time.

    FORM ILBM   times NumberOfBitMaps
        BMHD
        BODY

    - sound list -
{   FORM 8SVX   times NumberOfSounds
        VHDR
        BODY
    SEFX    }   Default parameters for sound

    - frame list -
{   FRAM        times NumberOfFrames
    SEFX        if sound for channel 1.
    SEFX        if sound for channel 2.
```

```
    POLY        times NumberOfPolys
  { CSTR        Text of poly if PolyType = pTEXTBLOCK
    CSTR  } }   Name of font
*/
```

1.109 FANT.doc / Notes

```
/*************************************************************************/
/*
** 1 - The color palette is a number from 0 to 1120.  The first 32 numbers
**     are normal RGB colors, but the rest index into a pre-defined
**     set of patterns.
**
** 2 - The view depth of each polygon determines the display order.  The
**     higher the number the closer the polygon is to the viewer.  During
**     editing, each polygon is assigned numbers in multiplies of 100,
**     but to function, any number can work.
*/
```

1.110 A / IFF Third Party Public Form and Chunk Specification / HEAD.doc

```
Flow -  New Horizons Software

TITLE:  HEAD  (FORM used by Flow - New Horizons Software, Inc.)

IFF FORM / CHUNK DESCRIPTION


Form/Chunk ID:  FORM HEAD, Chunks NEST, TEXT, FSCC

Date Submitted: 03/87
Submitted by:   James Bayless - New Horizons Software, Inc.

 FORM
 CHUNKS
```

1.111 HEAD.doc / FORM

```
FORM ID:  HEAD

FORM Description:

FORM HEAD is the file storage format of the Flow idea processor by New
Horizons Software, Inc.  Currently only the TEXT and NEST chunks are used.
There are plans to incorporate FSCC and some additional chunks for headers
and footers.
```

1.112 HEAD.doc / CHUNKS

```
CHUNK ID:  NEST

This chunk consists of only of a word (two byte) value that gives the new
current nesting level of the outline.  The initial nesting level
(outermost level) is zero.  It is necessary to include a NEST chunk only
when the nesting level changes.  Valid changes to the nesting level are
either to decrease the current value by any amount (with a minimum of 0)
or to increase it by one (and not more than one).

CHUNK ID:  TEXT

This chunk is the actual text of a heading.  Each heading has a TEXT chunk
(even if empty).  The text is not NULL terminated - the chunk size gives
the length of the heading text.

CHUNK ID: FSCC

This chunk gives the Font/Style/Color changes in the heading from the most
recent TEXT chunk.  It should occur immediately after the TEXT chunk it
modifies.  The format is identical to the FSCC chunk for the IFF form type
'WORD' (for compatibility), except that only the 'Location' and 'Style'
values are used (i.e., there can be currently only be style changes in an
outline heading).  The structure definition is:

    typedef struct {
       UWORD   Location;   /* Char location of change */
       UBYTE   FontNum;    /* Ignored */
       UBYTE   Style;      /* Amiga style bits */
       UBYTE   MiscStyle;  /* Ignored */
       UBYTE   Color;      /* Ignored */
       UWORD   pad;        /* Ignored */
    } FSCChange;

The actual chunk consists of an array of these structures, one entry for
each Style change in the heading text.
```

1.113 A / IFF Third Party Public Form and Chunk Specification / ILBM.CLUT.doc

```
Color Lookup Table chunk

amiga.dev/iff message 1527
----------
TITLE: CLUT IFF chunk proposal

"CLUT" IFF 8-Bit Color Look Up Table

Date: July 2, 1989
From: Justin V. McCormick
Status: Public Proposal
Supporting Software:  FG 2.0 by Justin V. McCormick for PP&S

 Introduction
```

```
 Purpose
 Specifications
 CLUT Example
 Design Notes
```

1.114 ILBM.CLUT.doc / Introduction

```
This memo describes the IFF supplement for the new chunk "CLUT".

Description:

A CLUT (Color Look Up Table) is a special purpose data module containing
table with 256 8-bit entries.  Entries in this table can be used directly
as a translation for one 8-bit value to another.
```

1.115 ILBM.CLUT.doc / Purpose

```
To store 8-bit data look up tables in a simple format for later retrieval.
These tables are used to translate or bias 8-bit intensity, contrast,
saturation, hue, color registers, or other similar data in a reproducable
manner.
```

1.116 ILBM.CLUT.DOC / Specifications

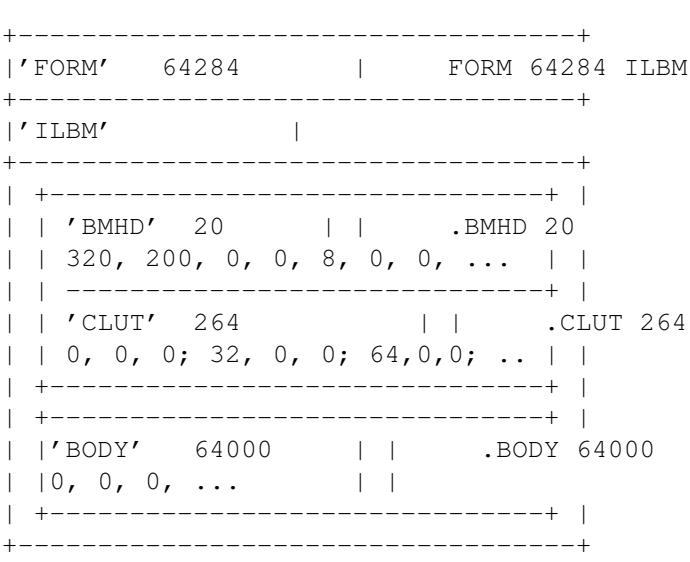
```
/* Here is the IFF chunk ID macro for a CLUT chunk */

#define ID_CLUT MakeID('C','L','U','T')

/*
 * Defines for different flavors of 8-bit CLUTs.
 */
#define CLUT_MONO       0L   /* A Monochrome, contrast or intensity LUT */
#define CLUT_RED        1L   /* A LUT for reds                */
#define CLUT_GREEN      2L   /* A LUT for greens              */
#define CLUT_BLUE       3L   /* A LUT for blues               */
#define CLUT_HUE        4L   /* A LUT for hues                */
#define CLUT_SAT        5L   /* A LUT for saturations         */
#define CLUT_UNUSED6    6L   /* How about a Signed Data flag */
#define CLUT_UNUSED7    7L   /* Or an Assumed Negative flag  */

/* All types > 7 are reserved until formally claimed */
#define CLUT_RESERVED_BITS 0xfffffff8L

/* The struct for Color Look-Up-Tables of all types */
typedef struct
{
  ULONG type;   /* See above type defines */
  ULONG res0;   /* RESERVED FOR FUTURE EXPANSION */
  UBYTE lut[256]; /* The 256 byte look up table */
} ColorLUT;
```

1.117 ILBM.CLUT.doc / CLUT Example

Normally, the CLUT chunk will appear after the BMHD of an FORM ILBM before the BODY chunk, in the same "section" as CMAPs are normally found. However, a FORM may contain only CLUTs with no other supporting information.

As a general guideline, it is desirable to group all CLUTs together in a form without other chunk types between them. If you were using CLUTs to store RGB intensity corrections, you would write three CLUTs in a row, R, G, then B.

Here is a box diagram for a 320x200x8 image stored as an IFF ILBM with a single CLUT chunk for intensity mapping:



1.118 ILBM.CLUT.doc / Design Notes

I have deliberately kept this chunk simple (KISS) to facilitate implementation. In particular, no provision is made for expansion to 16-bit or 32-bit tables. My reasoning is that a 16-bit table can have 64K entries, and thus would benefit from data compression. My suggestion would be to propose another chunk or FORM type better suited for large tables rather than small ones like CLUT.

1.119 A / IFF Third Party Public Form & Chunk Specification / ILBM.CTBL.DYCP.doc

Newtek Dynamic Ham color chunks

Newtek for Digiview IV (dynamic Ham)

ILBM.DYCP - dynamic color palette
3 longwords (file setup stuff)

ILBM.CTBL - array of words, one for each color (0rgb)

1.120 A / IFF Third Party Public Form and Chunk Specification / ILBM.DPI.doc

```
Dots per inch chunk

ILBM DPI chunk
--------------

Registered by:

Spencer Shanson
16 Genesta Rd
Plumstead
London SE18 3ES
ENGLAND

1-16-90

ILBM.DPI   Dots Per Inch   to allow output of an image at the
same resolution it was scanned at

typedef struct {
  UWORD dpi_x;
  UWORD dpi_y;
  } DPIHeader ;

For example, an image scanned at horizontal resolution of
240dpi and vertical resolution of 300dpi would be saved as:

44504920 00000004 00F0 012C
D P I    size     dpi_x dpi_y
```

1.121 A / IFF Third Party Public Form and Chunk Specification / ILBM.DPPV.doc

```
DPaint perspective chunk (EA)

IFF FORM / CHUNK DESCRIPTION

Form/Chunk ID:   Chunk DPPV  (DPaint II ILBM perspective chunk)
Date Submitted:  12/86
Submitted by:    Dan Silva

 Chunk Description
 Chunk Specification
 Supporting Software
```

1.122 ILBM.DPPV.doc / Chunk Description

```
The DPPV chunk describes the perspective state in a DPaintII ILBM.
```

1.123 ILBM.DPPV.doc / Chunk Specification

```
/* The chunk identifier DPPV */

#define ID_DPPV    MakeID('D','P','P','V')

typedef LONG LongFrac;
typedef struct ( LongFrac x,y,z; )  LFPoint;
typedef LongFrac  APoint[3];

typedef union {
   LFPoint l;
   APoint  a;
   } UPoint;

/* values taken by variable rotType */
#define ROT_EULER  0
#define ROT_INCR   1

/* Disk record describing Perspective state */

typedef struct {
   WORD     rotType;           /* rotation type */
   WORD     iA, iB, iC;        /* rotation angles (in degrees) */
   LongFrac Depth;             /* perspective depth */
   WORD     uCenter, vCenter;  /* coords of center perspective,
                                * relative to backing bitmap,
                                * in Virtual coords
                                */
   WORD     fixCoord;          /* which coordinate is fixed */
   WORD     angleStep;         /* large angle stepping amount */
   UPoint   grid;              /* gridding spacing in X,Y,Z */
   UPoint   gridReset;         /* where the grid goes on Reset */
   UPoint   gridBrCenter;      /* Brush center when grid was last on,
                                * as reference point
                                */
   UPoint   permBrCenter;      /* Brush center the last time the mouse
                                * button was clicked, a rotation performed,
                                * or motion along "fixed" axis
                                */
   LongFrac rot[3][3];         /* rotation matrix */
   } PerspState;
```

1.124 Supporting Software

```
DPaint II   by Dan Silva for Electronic Arts
```

1.125 A / IFF Third Party Public Form and Chunk Specification / ILBM.DRNG.doc

```
DPaint IV enhanced color cycle chunk (EA)

DRNG Chunk for FORM ILBM
```

```
------------------------

Submitted by Lee Taran

 Enhanced Color Cycling Capabilities
 DPaintIV DRNG Chunk
```

1.126 ILBM.DRNG.doc / Enhanced Color Cycling Capabilities

```
* DPaintIV supports a new color cycling model which does NOT
  require that color cycles contain a contiguous range of color
  registers.

For example:
  If your range looks like:  [1][3][8][2]
  then at each cycle tick
     temp = [2],
     [2] = [8],
     [8] = [3],
     [3] = [1],
     [1] = temp

* You can now cycle a single register thru a series of rgb values.
  For example:
     If your range looks like: [1] [orange] [blue] [purple]
     then at each cycle tick color register 1 will take on the
     next color in the cycle.

     ie:  t=0:  [1] = curpal[1]
          t=1:  [1] = purple
          t=2:  [1] = blue
          t=3:  [1] = orange
          t=4:  goto t=0

* You can combine rgb cycling with traditional color cycling.
  For example:
      Your range can look like:
          [1] [orange] [blue] [2] [green] [yellow]

      t=0: [1] = curpal[1], [2] = curpal[2]
      t=1: [1] = yellow,    [2] = blue
      t=2: [1] = green,     [2] = orange
      t=3: [1] = curpal[2], [2] = curpal[1]
      t=4: [1] = blue,      [2] = yellow
      t=5: [1] = orange,    [2] = green
      t=6: goto t=0

Note:
   * DPaint will save out an old style range CRNG if the range fits
     the CRNG model otherwise it will save out a DRNG chunk.
   * no thought has been given (yet) to interlocking cycles
```

1.127 ILBM.DRNG.doc / DPaintIV DRNG chunk

```
       DRNG ::= "DRNG" # { DRange DColor* DIndex* }

a <cell> is where the color or register appears within the range

The RNG_ACTIVE flags is set when the range is cyclable. A range
should only have the RNG_ACTIVE if it:
     1> contains at least one color register
     2> has a defined rate
     3> has more than one color and/or color register
If the above conditions are met then RNG_ACTIVE is a user/program
preference.  If the bit is NOT set the program should NOT cycle the
range.

The RNG_DP_RESERVED flags should always be 0!!!

typedef struct {
   UBYTE min;           /* min cell value */
   UBYTE max;           /* max cell value */
   SHORT rate;          /* color cycling rate, 16384 = 60 steps/second */
   SHORT flags;         /* 1=RNG_ACTIVE,4=RNG_DP_RESERVED */
   UBYTE ntrue;         /* number of DColor structs to follow */
   UBYTE nregs;         /* number of DIndex structs to follow */
   } DRange;

typedef struct { UBYTE cell; UBYTE r,g,b; } DColor; /* true color cell */
typedef struct { UBYTE cell; UBYTE index; } DIndex;
                                                /* color register cell */
```

1.128 A / IFF Third Party Public Form and Chunk Specification / ILBM.EPSF.doc

```
Encapsulated Postscript chunk

ILBM EPSF Chunk
---------------

Pixelations   Kevin Saltzman   617-277-5414

Chunk to hold encapsulated postscript

Used by PixelScript in their clip art.  Holds a postscript representation
of the ILBM's graphic image.

EPSF length
   ; Bounding box
   WORD lowerleftx;
   WORD lowerlefty;
   WORD upperrightx;
   WORD upperrighty;
   CHAR []    ; ascii postscript file
```

1.129 A / IFF Third Party Public Form and Chunk Specification / MTRX.doc

```
Numerical data storage (MathVision - Seven Seas)

MTRX FORM, for matrix data storage                    19-July-1990

Submitted by: Doug Houck
              Seven Seas Software

 Introduction
 Chunks
```

1.130 MTRX.doc / Introduction

Numerical data, as it comes from the real world, is an ill-mannered beast. Often much is assumed about the data, such as the number of dimensions, formatting, compression, limits, and sizes. As such, data is not portable. The MTRX FORM will both store the data, and completely describe its format, such that programs no longer need to guess the parameters of a data file. There needs to be but one program to read ascii files and output MTRX IFF files.

A matrix, by our definition, is composed of three types of things. Firstly, the atomic data, such as an integer, or floating point number. Secondly, arrays, which are simply lists of things which are all the same. Thirdly, structures, which are lists of things which are different. Both arrays and structures may be composed of things besides atomic data - they may contain other structures and arrays as well. This concept of nesting structures may be repeated to any desired depth.

For example, a list of data pairs could be encoded as an array of structures, where each structure contains two numbers. A two-dimensional array is simply an array of arrays.

Since space conservation is often desirable, there is provision for representing each number with fewer bits, and compressing the bits together.

1.131 MTRX.doc / Chunks

The MTRX FORM is composed of the definition of the structure, followed by the BODY which contains the data which is defined. Usually, there is only one set of data, but a smarter IFF read could use the definition as a PROPerty, with identically formatted data sets (BODYs) in a LIST.

```
  FORM MTRX
    definition (ARRY | STRU | DTYP)
    BODY
```

ARRY: The array chunk defines a counted list of similar items. The first (required) chunk in an ARRY is ELEM, which gives the number of elements in the array. Optionally, there may be limits given, (LOWR and UPPR), which

could be used in scaling during sampling of the data. Lastly is the definition of an element of the array, which may be a nested definition like everything else.

```
  ARRY ::= "ARRY" #{ ELEM [LOWR] [UPPR] [PACK] ARRY|STRU|DTYP }
```

STRU: The structure chunk defines a counted list of dissimilar things. The first (required) chunk in a STRU is FLDS, which gives the number of fields in the structure. Lastly are definitions of each field in the structure. Again, each field may have a nested definition like everything else.

```
 STRU ::= "STRU" #{ FLDS ([PACK] ARRY|STRU|DTYP)* }
```

VALU: The value contains a datatype, and then a constant of that type. The datatype contains the size of the constant, so this chunk has variable size. VALU is used in the ARRY chunk to give the scaling limits of the array.

BODY: This is the actual data we went to so much effort to describe. It is stored in "row-first" format, that is, items at the bottom of the nested description are stored next to each other. In most cases, it should be sufficient to simply block-read the whole chunk from disk, unless the reader needs to adjust byte-ordering or store in a more time-efficient format in memory. Data is assumed to be byte-aligned.

PACK: The PACK chunk is necessary when the bit length of the data is not a multiple of 8, that is, not byte-aligned, and the user wishes to conserve space by packing data items together. PACK is simply a number - the number of items to bit-pack before aligning on a byte. A PACK is in effect for the remainder of its nested scope, or until overridden by a new specification. A STRU or ARRY is assumed to have a PACK of 1 by default - it is not affected by PACKs in definitions above. A PACK of 0 means to byte-align before processing the next definition. The PACK specifier should be normalized. For example, when packing a large array of 3-bit numbers, PACK should be 8 since 3*8 = 24. In this case 8 is the smallest PACK number which aligns on a byte naturally.

DTYP: The DataType is the most interesting chunk, as it attempts to define every conceivable type of numeric data with 32 bits. The 32 bits are broken down into three fields, 1) the size in bits, 2) the Class, and 3) SubClass. The Class makes the most major distinction, separating integers from floating point numbers from Binary Coded Decimal and etc. Within each class is a SubClass, which gives the specific encoding used. Finally, the Size tells what how much room the data occupies. The basic division of datatypes is given in the tree structure below.

```
    Class             SubClass     Size  Final Specific Type
    =====             ========     ====  ===================
     |
    Binary Unsigned - 0 ------------ 8   UByte
     |                               16   UWord
     |                               32   ULong
     |
    Binary Signed --- 0 ------------  8  Byte
     |                                16  Word
     |                                32  ULong
     |
```

```
    Real ------------Ieee38 -------- 32  Ieee Single Precision
     |                |
     |               Ieee308 ------- 64  Double Precision
     |                |              32  Truncated Double Precision
     |                |
     |               FFP ----------- 32  Motorola Fast Floating Point
     |
    Text ----------- Text0 --------- ??  Null-terminated text
     |                |
     |               CText --------- ??  Number of characters in first byte
     |                |
     |               FText --------- ??  Fixed length, space padded
     |
    BCD ------------ Nibble -------- ??
                      |
                     Character ----- ??
```

A design goal was to create a classification system which other people can easily plug into. Many data types are simply size variations on existing data types. For example, a 4-bit integer can be specified by giving the size as four bits in the Signed Binary class. Be aware that not all MTRX readers may support your new type, but there will not be any type clashes or ambiguities by following these rules. If you have a truly unique Class or SubClass, you will need to register it with Commodore to prevent clashes.

A second design goal was to create a format which is easily decoded by software. By aligning fields on bytes, you have the option of redefining the datatype as a structure, so as to avoid shifting when accessing the fields. Since the numbers are sequentially assigned, they are suitable as array indicies, and may be optimized in a C switch statement.

A third design goal was allowing for naive and sophisticated readers. In checking for a certain datatype, a naive reader can simply compare the whole datatype with a small set of known types, which assumes that each different Size defines a unique datatype. Sophisticated readers will consider the Class, SubClass and Size separately, so as to support arbitrary size integers, and truncated Floating Point numbers, for example.

```
 *
 * MTRX ::= "FORM" #{ "MTRX" ARRY|STRU|DTYP BODY       } Matrix
 * ARRY ::= "ARRY" #{ ELEM [LOWR] [UPPR] [PACK] ARRY|STRU|DTYP } Array
 * STRU ::= "STRU" #{ FLDS ([PACK] ARRY|STRU|DTYP)*    } Structure
 * ELEM ::= "ELEM" #{ elements                         } Array elements
 * LOWR ::= "LOWR"  { VALU                             } Minimum limit
 * UPPR ::= "UPPR"  { VALU                             } Maximum limit
 * VALU ::=        #{ dtyp value                       } Value (in union)
 * dtyp ::=         { size, subclass, class            } Data Type (scalar)
 * DTYP ::= "DTYP" #{ dtyp                             }
 * FLDS ::= "FLDS" #{ number of fields                 } Number of Fields
 * PACK ::= "PACK" #{ units packed b4 byte alignment   } Packing
 * BODY ::= "BODY" #{ inner-first binary dump          } Data
 *
 *   [] means optional
 *   #  means the size of the unit following
 *   *  means one or more of
```

```
 *
```

1.132 A / IFF Third Party Public Form and Chunk Specification / PGTB.doc

```
Program traceback (SAS Institute)

FORM PGTB

Proposal:
        New IFF chunk type, to be named PGTB, meaning ProGram TraceBack.

 Format
```

1.133 PGTB.doc / Format

```
    'PGTB'       - chunk identifier
    length       - longword for length of chunk

    'FAIL'       - subfield giving environment at time of crash
    length       - longword length of subfield
    NameLen      - length of program name in longwords (BSTR)
    Name         - program name packed in longwords
    Environment  - copy of AttnFlags field from ExecBase,
                   gives type of processor, and existence of
                   math chip
    VBlankFreq   - copy of VBlankFrequency field from ExecBase
    PowerSupFreq - copy of PowerSupplyFrequency field from ExecBase
                   above fields may be used to determine whether
                   machine was PAL or NTSC
    Starter      - non-zero = CLI, zero = WorkBench
    GURUNum      - exception number of crash
    SegCount     - number of segments for program
    SegList      - copy of seglist for program
                   (Includes all seglist pointers, paired with
                    sizes of the segments)

    'REGS'       - register dump subfield
    length       - length of subfield in longwords
    GURUAddr     - PC at time of crash
    Flags        - copy of Condition Code Register
    DDump        - dump of data registers
    ADump        - dump of address registers

    'VERS'       - revision of program which created this file
    length       - length of subfield in longwords
    version      - main version of writing program
    revision     - minor revision level of writing program
    TBNameLen    - length of name of writing program
    TBName       - name of writing program packed in longwords (BSTR)

    'STAK'       - stack dump subfield
    length       - length of subfield in longwords
```

```
    (type)       - tells type of stack subfield, which can be any of
                   the following:

    ------------------------------------------------------------
    Info           - value 0
    StackTop       - address of top of stack
    StackPtr       - stack pointer at time of crash
    StackLen       - number of longwords on stack

    ------------------------------------------------------------
    Whole stack    - value 1
                     only used if total stack to be dumped is 8k
                     or less in size
    Stack          - dump of stack from current to top

    ------------------------------------------------------------
    Top 4k         - value 2
                     if stack used larger than 8k, this part
                     is a dump of the top 4k
    Stack          - dump of stack from top - 4k to top

    ------------------------------------------------------------
    Bottom 4k      - value 3
                     if stack used larger than 8k, this part
                     is a dump of the bottom 4k
    Stack          - dump of stack from current to current + 4k

In other words, we will dump a maximum of 8k of stack data.  This does NOT
mean the stack must be less than 8k in size to dump the entire stack, just
that the amount of stack USED be less than 8k.

    'UDAT'       - Optional User DATa chunk.  If the user assigns
                   a function pointer to the label "_ONGURU", the
                   catcher will call this routine prior to closing
                   the SnapShot file, passing one parameter on the
                   stack - an AmigaDOS file pointer to the SnapShot
                   file.  Spec for the _ONGURU routine:

                         void <function name>(fp)
                         long fp;

                   In other words, your routine must be of type 'void'
                   and must take one parameter, an AmigaDOS file
                   handle (which AmigaDOS wants to see as a LONG).
    length       - length of the UserDATa chunk, calculated after the
                   user routine terminates.
```

1.134 A / IFF Third Party Public Form and Chunk Specification / PRSP.doc

```
DPaint IV perspective move form (EA)

Submitted by Lee Taran

/* ------------------------------------------------------------------
   IFF Information:
```

```
      PRSP ::= "FORM" # {"PSRP" MOVE }
      MOVE ::= "MOVE" # { MoveState  }
 * ------------------------------------------------------------------ */

   typedef struct {
      BYTE reserved;        /* initialize to 0 */
      BYTE moveDir;         /* 0 = from point  1 = to point  */
      BYTE recordDir;       /* 0 = FORWARD,    1 = STILL, 2 = BACKWARD */
      BYTE rotationType;    /* 0 = SCREEN_RELATIVE, 1 = BRUSH_RELATIVE */
      BYTE translationType; /* 0 = SCREEN_RELATIVE, 1 = BRUSH_RELATIVE */
      BYTE cyclic;          /* 0 = NO, 1 = YES */
      SHORT distance[3];    /* x,y,z distance displacement */
      SHORT angle[3];       /* x,y,z rotation angles */
      SHORT nframes;        /* number of frames to move */
      SHORT easeout;        /* number of frames to ease out */
      SHORT easein;         /* number of frames to ease in */
      } MoveState;
```

1.135 A / IFF Third Party Public Form and Chunk Specification / RGBN-RGB8.doc

RGB image forms, Turbo Silver (Impulse)

FORM RGBN and FORM RGB8

RGBN and RGB8 files are used in Impulse's Turbo Silver and Imagine. They are almost identical to FORM ILBM's except for the BODY chunk and slight differences in the BMHD chunk.

A CAMG chunk IS REQUIRED.

The BMHD chunk specfies the number of bitplanes as 13 for type RGBN and 25 for type RGB8, and the compression type as 4.

The FORM RGBN uses 12 bit RGB values, and the FORM RGB8 uses 24 bit RGB values.

The BODY chunk contains RGB values, a "genlock" bit, and repeat counts. In Silver, when "genlock" bit is set, a "zero color" is written into the bitplanes for genlock video to show through. In Diamond and Light24 (Impulse 12 & 24 bit paint programs), the genlock bit is ignored if the file is loaded as a picture (and the RGB color is used instead), and if the file is loaded as a brush the genlock bit marks pixels that are not part of the brush.

For both RGBN and RGB8 body chunks, each RGB value always has a repeat count. The values are written in different formats depending on the magnitude of the repeat count.

RGBN BODY Chunk
RGB8 Body Chunk
Sample BODY Code

1.136 RGBN-RGB8.doc / RGBN BODY Chunk

For each RGB value, a WORD (16-bits) is written: with the 12 RGB bits in the MSB (most significant bit) positions; the "genlock" bit next; and then a 3 bit repeat count. If the repeat count is greater than 7, the 3-bit count is zero, and a BYTE repeat count follows. If the repeat count is greater than 255, the BYTE count is zero, and a WORD repeat count follows. Repeat counts greater than 65536 are not supported.

1.137 RGBN-RGB8.doc / RGB8 Body Chunk

For each RGB value, a LONG-word (32 bits) is written: with the 24 RGB bits in the MSB positions; the "genlock" bit next, and then a 7 bit repeat count.

In a previous version of this document, there appeared the following line:

"If the repeat count is greater than 127, the same rules apply as in the RGBN BODY."

But Impulse has never written more than a 7 bit repeat count, and when Imagine and Light24 were written, they didn't support reading anything but 7 bit counts.

1.138 RGBN-RGB8.doc / Sample BODY Code

```
    if(!count) {
            if (Rgb8) {
                    fread (&w, 4, 1, RGBFile);
                    lock = w & 0x00000080;
                    rgb = w >> 8;
                    count = w & 0x0000007f;
            } else {
                    w = (UWORD) getw (RGBFile);
                    lock = w & 8;
                    rgb = w >> 4;
                    count = w & 7;
            }
            if (!count)
                    if (!(count = (UBYTE) getc (RGBFile)))
                            count = (UWORD) getw (RGBFile);
    }
```

The pixels are scanned from left to right across horizontal lines, processing from top to bottom. The (12 or 24 bit) RGB values are stored with the red bits as the MSB's, the green bits next, and the blue bits as the LSB's.

Special note: As of this writing (Sep 88), Silver does NOT support anything but black for color zero.

1.139 A / IFF Third Party Public Form and Chunk Specification / SAMP.doc

```
Sampled sound format

                    IFF FORM "SAMP" Sampled Sound

 Date:   Dec 3,1989
 From:   Jim Fiore and Jeff Glatt, dissidents
```

The form "SAMP" is a file format used to store sampled sound data in some ways like the current standard, "8SVX". Unlike "8SVX", this new format is not restricted to 8 bit sample data. There can be more than one waveform per octave, and the lengths of different waveforms do not have to be factors of 2. In fact, the lengths (waveform size) and playback mapping (which musical notes each waveform will "play") are independently determined for each wave- form. Furthermore, this format takes into account the MIDI sample dump stan- dard (the defacto standard for musical sample storage), while also incorpo- rating the ability to store Amiga specific info (for example, the sample data that might be sent to an audio channel which is modulating another channel).

Although this form can be used to store "sound effects" (typically oneShot sounds played at a set pitch), it is primarily intended to correct the many deficiencies of the "8SVX" form in regards to musical sampling. Because the emphasis is on musical sampling, this format relies on the MIDI (Musical Instrument Digital Interface) method of describing "sound events" as does virtually all currently manufactured, musical samplers. In addition, it at- tempts to incorporate features found on many professional music samplers, in anticipation that future Amiga models will implement 16 bit sampling, and thus be able to achieve this level of performance. Because this format is more complex than "8SVX", programming examples to demonstrate the use of this format have been included in both C and assembly. Also, a library of func- tions to read and write SAMP files is available, with example applications.

SEMANTICS: When MIDI literature talks about a sample, usually it means a collection of many sample points that make up what we call a "wave".

1.140 SAMP.doc / Similarities and Differences from the 8SVX Form

Like "8SVX", this new format uses headers to separate the various sections of the sound file into chunks. Some of the chunks are exactly the same since there wasn't a need to improve them. The chunks that remain unchanged are as follows:

```
"(c) "
"AUTH"
"ANNO"
```

Since these properties are all described in the original "8SVX" document, please refer to that for a description of these chunks and their uses. Like the "8SVX" form, none of these chunks are required to be in a sound file. If they do appear, they must be padded out to an even number of bytes.

Furthermore, two "8SVX" chunks no longer exist as they have been incorporated into the "BODY" chunk. They are:

```
"ATAK"
"RLSE"
```

Since each wave can be completely different than the other waves in the sound file (one wave might be middle C on a piano, and another might be a snare drum hit), it is necessary for each wave to have its own envelope description, and name.

The major changes from the "8SVX" format are in the "MHDR", "NAME", and "BODY" chunks.

1.141 SAMP.doc / The SAMP Header

At the very beginning of a sound file is the "SAMP" header. This is used to determine if the disk file is indeed a SAMP sound file. It's attributes are as follows:

```
    #define ID_SAMP MakeID('S','A','M','P')
```

In assembly, this looks like:

```
    CNOP 0,2  ;word-align

SAMP         dc.b  'SAMP'
sizeOfChunks dc.l  [sizes of all subsequent chunks summed]
```

1.142 SAMP.doc / The MHDR Chunk

The required "MHDR" chunk immediately follows the "SAMP" header and consists of the following components:

```
    #define ID_MHDR MakeID('M','H','D','R')
```

```
/* MHDR size is dependant on the size of the imbedded PlayMap. */

 typedef struct{
  UBYTE NumOfWaves, /* The number of waves in this file */
    Format,         /* # of ORIGINAL significant bits from 8-28 */
    Flags,          /* Various bits indicate various functions */
    PlayMode,       /* determines play MODE of the PlayMap */
    NumOfChans,
    Pad,
    PlayMap[128*4], /* a map of which wave numbers to use for
                       each of 128 possible Midi Notes. Default to 4 */
  } MHDRChunk;
```

The PlayMap is an array of bytes representing wave numbers. There can be a total of 255 waves in a "SAMP" file. They are numbered from 1 to 255. A wave number of 0 is reserved to indicate "NO WAVE". The Midi Spec 1.0 designates that there are 128 possible note numbers (pitches), 0 to 127. The size of an MHDR's PlayMap is determined by (NumOfChans * 128). For example, if NumOfChans = 4, then an MHDR's PlayMap is 512 bytes. There are 4 bytes in the PlayMap for EACH of the 128 Midi Note numbers. For example, the first 4 bytes in PlayMap pertain to Midi Note #0. Of those 4 bytes, the first byte is the wave number to play back on Amiga audio channel 0. The second byte is the wave number to play back on Amiga audio channel 1, etc. In this way, a single Midi Note Number could simultaneously trigger a sound event on each of the 4 Amiga audio channels. If NumOfChans is 1, then the PlayMap is 128 bytes and each midi note has only 1 byte in the PlayMap. The first byte pertains to midi note #0, the second pertains to midi note #1, etc. In this case, a player program might elect to simply play back the PlayMap wave number on any available amiga audio channel. If NumOfChans = 0, then there is no imbedded PlayMap in the MHDR, no midi note assignments for the waves, and an application should play back waves on any channel at their default sampleRates.

In effect, the purpose of the PlayMap array is to determine which (if any) waves are to be played back for each of the 128 possible Midi Note Numbers. Usually, the MHDR's NumOfChans will be set to 4 since the Amiga has 4 audio channels. For the rest of this document, the NumOfChans is assumed to be 4.

As mentioned, there can be a total of 255 waves in a "SAMP" file, numbered from 1 to 255. A PlayMap wave number of 0 is reserved to indicate that NO WAVE number should be played back. Consider the following example: ? The first 4 bytes of PlayMap are 1,3,0,200.

If a sample playing program receives (from the serial port or another task perhaps) Midi Note Number 0, the following should occur:

1) The sampler plays back wave 1 on Amiga audio channel number 0 (because the first PlayMap byte is 1).
2) The sampler plays back wave 3 on Amiga audio channel number 1 (because the second PlayMap byte is 3).
3) The sampler does not effect Amiga audio channel 2 in any way (because the third PlayMap byte is a 0).
4) The sampler plays back wave 200 on Amiga audio channel number 4 (because the fourth PlayMap byte is 200).

```
(This assumes INDEPENDANT CHANNEL play MODE to be discussed later in this
 document.)

All four of the PlayMap bytes could even be the same wave number. This
would cause that wave to be output of all 4 Amiga channels simultaneously.

NumOfWaves is simply the number of waves in the sound file.

Format is the number of significant bits in every sample of a wave. For
example, if Format = 8, then this means that the sample data is an 8 bit
format, and that every sample of the wave can be expressed by a single
BYTE. (A 16 bit sample would need a WORD for every sample point).

Each bit of the Flags byte, when set, means the following:

Bit #0 -
   File continued on another disc. This might occur if the SAMP file
   was too large to fit on 1 floppy. The accepted practice (as incor-
   porated by Yamaha's TX sampler and Casio's FZ-1 for example) is to
   dump as much as possible onto one disc and set a flag to indicate
   that more is on another disc's file. The name of the files must
   be the related. The continuation file should have its own SAMP header
   MHDR, and BODY chunks. This file could even have its continuation
   bit set, etc. Never chop a sample wave in half. Always close the
   file on 1 disc after the last wave which can be completely saved.
   Resume with the next wave within the BODY of the continuation file.
   Also, the NumOfWaves in each file's BODY should be the number saved
   on that disc (not the total number in all combined disk files).
   See the end of this document for filename conventions.

In C, here is how the PlayMap is used when receiving a midi note-on event:

   MapOffset = (UBYTE) MidiNoteNumber * numOfChans;
   /* MidiNoteNumber is the received note number (i.e. the second byte of
     a midi note-on event. numOfChans is from the SAMP MHDR. */
   chan0waveNum = (UBYTE) playMap[MapOffset];
   chan1waveNum = (UBYTE) playMap[MapOffset+1];
   chan2waveNum = (UBYTE) playMap[MapOffset+2];
   chan3waveNum = (UBYTE) playMap[MapOffset+3];

   if (chan0waveNum != 0)
   { /* get the pointer to wave #1's data, determine the values
        that need to be passed to the audio device, and play this
        wave on Amiga audio channel #0 (if INDEPENDANT PlayMode) */
   }

   /* do the same with the other 3 channel's wave numbers */

In assembly, the "MHDR" structure looks like this:

             CNOP 0,2
MHDR        dc.b 'MHDR'
sizeOfMHDR  dc.l [this is 6 + (NumOfChans * 128) ]
NumOfWaves  dc.b [a byte count of # of waves in the file]
Format      dc.b [a byte count of # of significant bits in a sample point]
Flags       dc.b [bit mask]
PlayMode    dc.b [play MODE discussed later]
```

```
NumOfChans  dc.b [# of bytes per midi note for PlayMap]
PlayMap     ds.b [128 x NumOfChans bytes of initialized values]

and a received MidiNoteNumber is interpreted as follows:

   moveq   #0,d0
   move.b  MidiNoteNumber,d0  ;this is the received midi note #
   bmi.s   Illegal_Number     ;exit, as this is an illegal midi note #
   moveq   #0,d1
   move.b  NumOfChans,d1
   mulu.w  d1,d0              ;MidiNoteNumber x NumOfChans
   lea     PlayMap,a0
   adda.l  d0,a0
   move.b  (a0)+,chan0waveNum
   move.b  (a0)+,chan1waveNum
   move.b  (a0)+,chan2waveNum
   move.b  (a0),chan3waveNum

   tst.b   chan0waveNum
   beq.s   Chan1
 ;Now get the address of this wave number's sample data, determine the
 ;values that need to be passed to the audio device, and output the wave's
 ;data on Amiga chan 0 (assuming INDEPENDANT PlayMode).

Chan1 tst.b chan1waveNum
      beq.s Chan2
 ;do the same for the other wave numbers, etc.
```

1.143 SAMP.doc / The NAME Chunk

```
   #define ID_NAME MakeID('N','A','M','E')
```

If a NAME chunk is included in the file, then EVERY wave must have a name. Each name is NULL-terminated. The first name is for the first wave, and it is immediately followed by the second wave's name, etc. It is legal for a wave's name to be simply a NULL byte. For example, if a file contained 4 waves and a name chunk, the chunk might look like this:

```
              CNOP 0,2

   Name       dc.b 'NAME'
   sizeOfName dc.l 30
              dc.b 'Snare Drum',0  ;wave 1
              dc.b 'Piano 1',0     ;wave 2
              dc.b 'Piano A4',0    ;wave 3
              dc.b 0               ;wave 4
              dc.b 0
```

NAME chunks should ALWAYS be padded out to an even number of bytes. (Hence the extra NULL byte in this example). The chunk's size should ALWAYS be even consequently. DO NOT USE the typical IFF method of padding a chunk out to an even number of bytes, but allowing an odd number size in the header.

1.144 SAMP.doc / The BODY Chunk

The "BODY" chunk is CONSIDERABLY different than the "8SVX" form. Like all chunks it has an ID.

```
    #define ID_BODY MakeID('B','O','D','Y')
```

Every wave has an 80 byte waveHeader, followed by its data. The waveHeader structure is as follows:

```
    typedef struct {
      ULONG  WaveSize;    /* total # of BYTES in the wave (MUST be even) */
      UWORD  MidiSampNum; /* ONLY USED for Midi Dumps */
      UBYTE  LoopType,    /* ONLY USED for Midi Dumps */
      InsType;            /* Used for searching for a certain instrument */
      ULONG  Period,      /* in nanoseconds at original pitch */
      Rate,               /* # of samples per second at original pitch */
      LoopStart,          /* an offset in BYTES (from the beginning of the
                             of the wave) where the looping portion of the
                               wave begins. Set to WaveSize if no loop. */
      LoopEnd;            /* an offset in BYTES (from the beginning of the
                             of the wave) where the looping portion of the
                             wave ends. Set to WaveSize if no loop. */
      UBYTE  RootNote,    /* the Midi Note # that plays back orig. pitch */
      VelStart;           /* 0 = NO velocity effect, 128 =
                                   negative direction, 64 = positive
                                   direction (it must be one of these 3) */
      UWORD VelTable[16]; /* contains 16 successive offset values
                                  in BYTES from the beginning of the wave */

     /* The ATAK and RLSE segments contain an EGPoint[] piece-wise
        linear envelope just like 8SVX. The structure of an EGPoint[]
        is the same as 8SVX. See that document for details. */

      ULONG  ATAKsize,    /* # of BYTES in subsequent ATAK envelope.
                                 If 0, then no ATAK data for this wave. */
      RLSEsize,           /* # of BYTES in subsequent RLSE envelope
                                 If 0, then no RLSE envelope follows */

     /* The FATK and FRLS segments contain an EGPoint[] piece-wise
        linear envelope for filtering purposes. This is included in
        the hope that future Amiga audio will incorporate a VCF
        (Voltage Controlled Filter). Until then, if you are doing any
        non-realtime digital filtering, you could store info here. */

     sizeOfFATK,          /* # of BYTES in FATK segment */
     sizeOfFRLS,          /* # of BYTES in FRLS segment */

     USERsize;            /*   # of BYTES in the following data
                              segment (not including USERtype).
                              If zero, then no user data */
     UWORD  USERtype;     /* See explanation below. If USERsize
                               = 0, then ignore this. */

    /* End of the waveHeader. */
```

```
    /* The data for any ATAK, RLSE, FATK, FRLS, USER, and the actual wave
       data for wave #1 follows in this order:
       Now list each EGPoint[] (if any) for the VCA's
       (Voltage Controlled Amp) attack portion.
       Now list each EGPoint[] for the VCA's (Voltage Controlled Amp)
       release portion.
       List EGPoints[] (if any) for FATK.
       List EGPoints[] if any for FRLS */

   /*  Now include the user data here if there is any.Just pad it out to an
   even number of bytes and have USERsize reflect that. Finally,here is the
   actual sample data for the wave. The size (in BYTES) of this data is
   WaveSize. It MUST be padded out to an even number of bytes. */

    } WaveFormInfo;

 /* END OF WAVE #1 */

 /* The waveHeader and data for the next wave would now follow. It is
    the same form as the first wave */
```

In assembly, the BODY chunk looks like this:

```
           CNOP 0,2
BodyHEADER dc.b 'BODY'
sizeOfBody dc.l  [total bytes in the BODY chunk not counting 8 byte header]

   ; Now for the first wave
WaveSize       dc.l  ;[total # of BYTES in this wave (MUST be even)]
MidiSampNum    dc.w  ;[from Midi Sample Dump]  ; ONLY USED for Midi Dumps
LoopType       dc.b  ;[0 or 1]                 ; ONLY USED for Midi Dumps
InsType        dc.b  0
Period         dc.l  ;[period in nanoseconds at original pitch]
Rate           dc.l  ;[# of samples per second at original pitch]
LoopStart      dc.l    ;[an offset in BYTES (from the beginning of the
                       ; of the wave) to where the looping
                       ; portion of the wave begins.]
LoopEnd        dc.l    ;[an offset in BYTES (from the beginning of the
                       ; of the wave) to where the looping
                       ; portion of the wave ends]
RootNote       dc.b    ;[the Midi Note # that plays back original pitch]
VelStart       dc.b    ;[0, 64, or 128]
VelTable       dc.w    ;[first velocity offset]
               dc.w    ;[second velocity offset]...etc
               ds.w 14 ;...for a TOTAL of 16 velocity offsets

ATAKsize       dc.l  ;# of BYTES in subsequent ATAK envelope.
                     ;If 0, then no ATAK data for this wave.
RLSEsize       dc.l  ;# of BYTES in subsequent RLSE envelope
                     ;If 0, then no RLSE data
FATKsize       dc.l  ;# of BYTES in FATK segment
FRLSsize       dc.l  ;# of BYTES in FRLS segment
USERsize       dc.l  ;# of BYTES in the following User data
                     ;segment (not including USERtype).
                     ;If zero, then no user data
USERtype       dc.w  ; See explanation below. If USERsize
                  ; = 0, then ignore this.
```

```
;Now include the EGpoints[] (data) for the ATAK if any
;Now the EGpoints for the RLSE
;Now the EGpoints for the FATK
;Now the EGpoints for the FLSR
;Now include the user data here if there is any. Just pad
;it out to an even number of bytes.
;After the userdata (if any) is the actual sample data for
;the wave. The size (in BYTES) of this segment is WaveSize.
;It MUST be padded out to an even number of bytes.

; END OF WAVE #1
```

1.145 SAMP.doc / Structure of an Individual Sample Point

Even though the next generation of computers will probably have 16 bit audio, and 8 bit sampling will quickly disappear, this spec has sizes expressed in BYTES. (ie LoopStart, WaveSize, etc.) This is because each successive address in RAM is a byte to the 68000, and so calculating address offsets will be much easier with all sizes in BYTES. The Midi sample dump, on the other hand, has sizes expressed in WORDS. What this means is that if you have a 16 bit wave, for example, the WaveSize is the total number of BYTES, not WORDS, in the wave.

Also, there is no facility for storing a compression type. This is because sample data should be stored in linear format (as per the MIDI spec). Currently, all music samplers, regardless of their internal method of playing sample data must transmit and expect to receive sample dumps in a linear format. It is up to each device to translate the linear format into its own compression scheme. For example, if you are using an 8 bit compression scheme that yields a 14 bit linear range, you should convert each sample data BYTE to a decom- pressed linear WORD when you save a sound file. Set the MHDR's Format to 14. It is up to the application to do its own compression upon loading a file. The midi spec was set up this way because musical samplers need to pass sample data between each other, and computers (via a midi interface). Since there are almost as many data compression schemes on the market as there are musical products, it was decided that all samplers should expect data received over midi to be in LINEAR format. It seems logical to store it this way on disc as well. Therefore, any software program "need not know" how to decompress another software program's SAMP file. When 16 bit sampling is eventually implemented there won't be much need for compression on playback anyway. The continuation Flag solves the problem of disc storage as well.

Since the 68000 can only perform math on BYTES, WORDS, or LONGS, it has been decided that a sample point should be converted to one of these sizes when saved in SAMP as follows:

ORIGINAL significant bits	SAMP sample point size
8	BYTE
9 to 16	WORD
17 to 28	LONG

Furthermore, the significant bits should be left-justified since it is

easier to perform math on the samples.

So, for example, an 8 bit sample point (like 8SVX) would be saved as a BYTE with all 8 bits being significant. The MHDR's Format = 8. No conversion is necessary.

A 12 bit sample point should be stored as a WORD with the significant bits being numbers 4 to 15. (i.e shift the 12-bit WORD 4 places to the left). Bits 0, 1, 2 and 3 may be zero (unless some 16-bit math was performed and you wish to save these results). The MHDR's Format = 12. In this way, the sample may be loaded and manipulated as a 16-bit wave, but when transmitted via midi, it can be converted back to 12 bits (rounded and shifted right by 4).

A 16 bit sample point would be saved as a WORD with all 16 bits being significant. The MHDR's Format = 16. No conversion is necessary.

1.146 SAMP.doc / The Waveheader Explained

The WaveSize is, as stated, the number of BYTES in the wave's sample table. If your sample data consisted of the following 8 bit samples:

```
   BYTE  100,-90,80,-60,30,35,40,-30,-35,-40,00,12,12,10
```

then WaveSize = 14. (PAD THE DATA OUT TO AN EVEN NUMBER OF BYTES!)

The MidiSampNum is ONLY used to hold the sample number received from a MIDI Sample Dump. It has no bearing on where the wave should be placed in a SAMP file. Also, the wave numbers in the PlayMap are between 1 to 255, with 1 being the number of the first wave in the file. Remember that a wave number of 0 is reserved to mean "no wave to play back". Likewise, the LoopType is only used to hold info from a MIDI sample dump.

The InsType is explained at the end of this document. Often it will be set to 0.

The RootNote is the Midi Note number that will play the wave back at it's original, recorded pitch. For example, consider the following excerpt of a PlayMap:

```
   PlayMap  {2,0,0,4       /* Midi Note #0 channel assignment */
             4,100,1,0     /* Midi Note #1    "        "  */
             1,4,0,0       /* Midi Note #2    "        "  */
             60,2,1,1...}  /* Midi Note #3    "        "  */
```

Notice that Midi Notes 0, 1, and 2 are all set to play wave number 4 (on Amiga channels 3, 0, and 1 respectively). If we set wave 4's RootNote = 1, then receiving Midi Note number 1 would play back wave 4 (on Amiga channel 0) at it's original pitch. If we receive a Midi Note number 0, then wave 4 would be played back on channel 3) a half step lower than it's original pitch. If we receive Midi Note number 2, then wave 4 would be played (on channel 1) a half step higher than it's original pitch. If we receive Midi Note number 3, then wave 4 would not be played at all because it isn't specified in the PlayMap bytes for Midi Note number 3.

The Rate is the number of samples per second of the original pitch. For example, if Rate = 20000, then to play the wave at it's original pitch, the sampling period would be:

```
(1/20000)/.279365 = .000178977

#define AUDIO_HARDWARE_FUDGE .279365
```

where .279365 is the Amiga Fudge Factor (a hardware limitation). Since the Amiga needs to see the period in terms of microseconds, move the decimal place to the right 6 places and our sampling period = 179 (rounded to an integer).

In order to play the wave at higher or lower pitches, one would need to "transpose" this period value. By specifying a higher period value, the Amiga will play back the samples slower, and a lower pitch will be achieved. By specifying a lower period value, the amiga will play back the sample faster, and a higher pitch will be achieved. By specifying this exact period, the wave will be played back exactly "as it was recorded (sampled)". ("This period is JUST RIGHT!", exclaimed GoldiLocks.) Later, a method of transposing pitch will be shown using a "look up" table of periods. This should prove to be the fastest way to transpose pitch, though there is nothing in the SAMP format that compels you to do it this way.

The LoopStart is a BYTE offset from the beginning of the wave to where the looping portion of the wave begins. For example, if SampleData points to the start of the wave, then SampleData + LoopStart is the start address of the looping portion. In 8SVX, the looping portion was referred to as repeatHiSamples. The data from the start of the wave up to the start of the looping portion is the oneShot portion of the wave. LoopEnd is a BYTE offset from the beginning of the wave to where the looping portion ends. This might be the very end of the wave in memory, or perhaps there might be still more data after this point. You can choose to ignore this "trailing" data and play back the two other portions of the wave just like an 8SVX file (except that there are no other interpolated octaves of this wave).

VelTable contains 16 BYTE offsets from the beginning of the wave. Each successive value should be greater (or equal to) the preceding value. If VelStart = POSITIVE (64), then for each 8 increments in Midi Velocity above 0, you move UP in the table, add this offset to the wave's beginning address (start of oneShot), and start playback at that address. Here is a table relating received midi note-on velocity vs. start playback address for POSITIVE VelStart. SamplePtr points to the beginning of the sample.

```
If midi velocity = 0, then don't play a sample, this is a note off
If midi velocity = 1 to 7, then start play at SamplePtr + VelTable[0]
If midi velocity = 8 to 15, then start at SamplePtr + VelTable[1]
If midi velocity = 16 to 23, then start at SamplePtr + VelTable[2]
If midi velocity = 24 to 31, then start at SamplePtr + VelTable[3]
If midi velocity = 32 to 39, then start at SamplePtr + VelTable[4]
If midi velocity = 40 to 47, then start at SamplePtr + VelTable[5]
If midi velocity = 48 to 55, then start at SamplePtr + VelTable[6]
If midi velocity = 56 to 63, then start at SamplePtr + VelTable[7]
If midi velocity = 64 to 71, then start at SamplePtr + VelTable[8]
If midi velocity = 72 to 79, then start at SamplePtr + VelTable[9]
```

```
    If midi velocity = 80 to 87, then start at SamplePtr + VelTable[10]
    If midi velocity = 88 to 95, then start at SamplePtr + VelTable[11]
    If midi velocity = 96 to 103, then start at SamplePtr + VelTable[12]
    If midi velocity = 104 to 111, then start at SamplePtr + VelTable[13]
    If midi velocity = 112 to 119, then start at SamplePtr + VelTable[14]
    If midi velocity = 120 to 127, then start at SamplePtr + VelTable[15]
```

We don't want to specify a scale factor and use integer division to find the sample start. This would not only be slow, but also, it could never be certain that the resulting sample would be a zero crossing if the start point is calcu- lated "on the fly". The reason for having a table is so that the offsets can be be initially set on zero crossings via an editor. This way, no audio "clicks" is guaranteed. This table should provide enough resolution.

If VelStart = NEGATIVE (128), then for each 8 increments in midi velocity, you start from the END of VelTable, and work backwards. Here is a table for NEGATIVE velocity start.

```
    If midi velocity = 0, then don't play a sample, this is a note off
    If midi velocity = 1 to 7, then start play at SamplePtr + VelTable[15]
    If midi velocity = 8 to 15, then start at SamplePtr + VelTable[14]
    If midi velocity = 16 to 23, then start at SamplePtr + VelTable[13]
    If midi velocity = 24 to 31, then start at SamplePtr + VelTable[12]
    If midi velocity = 32 to 39, then start at SamplePtr + VelTable[11]
    If midi velocity = 40 to 47, then start at SamplePtr + VelTable[10]
    If midi velocity = 48 to 55, then start at SamplePtr + VelTable[9]
    If midi velocity = 56 to 63, then start at SamplePtr + VelTable[8]
    If midi velocity = 64 to 71, then start at SamplePtr + VelTable[7]
    If midi velocity = 72 to 81, then start at SamplePtr + VelTable[6]
    If midi velocity = 80 to 87, then start at SamplePtr + VelTable[5]
    If midi velocity = 88 to 95, then start at SamplePtr + VelTable[4]
    If midi velocity = 96 to 103, then start at SamplePtr + VelTable[3]
    If midi velocity = 104 to 111, then start at SamplePtr + VelTable[2]
    If midi velocity = 112 to 119, then start at SamplePtr + VelTable[1]
    If midi velocity = 120 to 127, then start at SamplePtr + VelTable[0]
```

In essence, increasing midi velocity starts playback "farther into" the wave for POSITIVE VelStart. Increasing midi velocity "brings the start point back" toward the beginning of the wave for NEGATIVE VelStart.

If VelStart is set to NONE (0), then the wave's playback start should not be affected by the table of offsets.

What is the use of this feature? As an example, when a snare drum is hit with a soft volume, its initial attack is less pronounced than when it is struck hard. You might record a snare being hit hard. By setting VelStart to a NEGATIVE value and setting up the offsets in the Table, a lower midi velocity will "skip" the beginning samples and thereby tend to soften the initial attack. In this way, one wave yields a true representation of its instrument throughout its volume range. Furthermore, stringed and plucked instruments (violins, guitars, pianos, etc) exhibit different attacks at different volumes. VelStart makes these kinds of waves more realistic via a software implementation. Also, an application program can allow the user to enable/ disable this feature. See the section "Making the Velocity Table" for info on how to best choose the 16 table values.

1.147 SAMP.doc / MIDI Velocity vs. Amiga Channel Volume

```
The legal range for Midi Velocity bytes is 0 to 127. (A midi velocity of 0
should ALWAYS be interpreted as a note off).

The legal range for Amiga channel volume is 0 to 64. Since this is half of
the midi range, a received midi velocity should be divided by 2 and add 1
(but only AFTER checking for a received midi velocity of 0).

An example of how to implement a received midi velocity in C:

    If ( ReceivedVelocity != 0 && ReceivedVelocity < 128 )
    {   /* the velocity byte of a midi message */
        If (velStart != 0)
        {
            tableEntry = ReceivedVelocity / 8;
            If (velStart == 64)
            {    /* Is it POSITIVE */
                 startOfWave = SamplePtr + velTable[tableEntry];
                             /* ^where to find the sample start point */
            }
            If (velStart == 128)
            {    /* Is it NEGATIVE */
                 startOfWave = SamplePtr + velTable[15 - tableEntry];
            }
            volume = (receivedVelocity/2 + 1;  /* playback volume */
            /* Now playback the wave */
        }
    }

In assembly,

    lea      SampleData,a0        ;the start addr of the sample data
    moveq    #0,d0
    move.b   ReceivedVelocity,d0  ;the velocity byte of a midi message
    beq      A_NoteOff            ;If zero, branch to a routine to
                                  ;process a note-off message.

    bmi      Illegal_Vol          ;exit if received velocity > 127
    ;---Check for velocity start feature ON, and direction
    move.b   VelStart,d1
    beq.s    Volume               ;skip the velocity offset routine if 0
    bmi.s    NegativeVel          ;is it NEGATIVE? (128)

    ;---Positive velocity offset
    move.l   d0,d1                ;duplicate velocity
    lsr.b    #3,d1                ;divide by 8
    add.b    d1,d1                ;x 2 because we need to fetch a word
    lea      VelTable,a1     ;start at table's HEAD
    adda.l   d1,a1                ;go forward
    move.w   (a1),d1              ;get the velocity offet
    adda.l   d1,a0          ;where to start actual playback
    bra.s    Volume

  NegativeVel:
    ;---Negative velocity offset
```

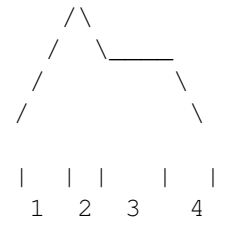
```
    move.l   d0,d1                ;duplicate velocity
    lsr.b    #3,d1                ;divide by 8
    add.b    d1,d1                ;x 2 because we need to fetch a word
    lea      VelTable+30,a1  ;start at table's END
    suba.l   d1,a1                ;go backwards
    move.w   (a1),d1              ;get the velocity offset
    adda.l   d1,a0          ;where to start actual playback

    ;---Convert Midi velocity to an Amiga volume
  Volume  lsr.b    #1,d0          ;divide by 2
          addq.b   #1,d0          ;an equivalent Amiga volume

   ;---Now a0 and d0 are the address of sample start, and volume
```

1.148 SAMP.doc / An EGpoint (Envelope Generator)

A single EGpoint is a 6 byte structure as follows:

```
EGpoint1: dc.w ;[the duration in milliseconds]
          dc.l ;[the volume factor - fixed point,16 bits to the left of the
               ;decimal point and 16 to the right.]
```

The volume factor is a fixed point where 1.0 ($00010000) represents the MAXIMUM volume possible. (i.e. No volume factor should exceed this value.) The last EGpoint in the ATAK is always the sustain point. Each EG's volume is determined from 0.0, not as a difference from the previous EG's volume. I hope that this clears up the ambiguity in the original 8SVX document. So, to recreate an amplifier envelope like this:



Stages 1, 2, and 3 would be in the ATAK data, like so:

```
;Stage 1
dc.w  100       ;take 100ms
dc.l  $00004000 ;go to this volume
dc.w  100
dc.l  $00008000
dc.w  100
dc.l  $0000C000
dc.w  100
dc.l  $00010000 ;the "peak" of our attack is full volume
;Stage 2
dc.w  100
dc.l  $0000C000 ;back off to this level
dc.l  100
dc.l  $00008000 ;this is where we hold (SUSTAIN) until the note
                ;is turned off. (We are now holding at stage 3)
```

```
Now the RLSE data would specify stage 4 as follows:
dc.w  100
dc.l  $00004000
dc.w  100
dc.l  $00000000 ;the volume is 0
```

1.149 SAMP.doc / Additional User Data Section

There is a provision for storing user data for each wave. This is where an application can store Amiga hardware info, or other, application specific info. The waveHeader's USERtype tells what kind of data is stored. The current types are:

```
#define SPECIFIC 0
#define VOLMOD   1
#define PERMOD   2
#define LOOPING  3
```

SPECIFIC (0) - application specific data. It should be stored in a format that some application can immediately recognize. (i.e. a "format within" the SAMP format) If the USERtype is SPECIFIC, and an application doesn't find some sort of header that it can recognize, it should conclude that this data was put there by "someone else", and ignore the data.

VOLMOD (1) - This data is for volume modulation of an Amiga channel as described by the ADKCON register. This data will be sent to the modulator channel of the channel set to play the wave.

PERMOD (2) - This data is for period modulation of an Amiga channel as described by the ADKCON register. This data will be sent to the modulator channel of the channel set to play the wave.

LOOPING (3) - This contains more looping points for the sample. There are some samplers that allow more than just one loop (Casio products primarily). Additional looping info can be stored in this format:

```
UWORD numOfLoops;  /* number of loop points to follow */

ULONG StartLoop1,  /* BYTE offset from the beginning of
                      the sample to the start of loop1 */
EndLoop1,          /* BYTE offset from the beginning of
                      the sample to the end of loop1 */

StartLoop2,        /* ...etc */
```

1.150 SAMP.doc / Converting Midi Sample Dump to SAMP

SEMANTICS: When MIDI literature talks about a sample, usually it means a collection of many sample points that make up what we call "a wave". Therefore, a Midi Sample Dump sends all the sample data that makes up ONE wave. A SAMP file is designed to hold up to 255 of these waves (midi dumps).

The Midi Sample Dump specifies playback rate only in terms of a sample PERIOD in nanoseconds. SAMP also expresses playback in terms of samples per second (frequency). The Amiga needs to see its period rounded to the nearest microsecond. If you take the sample period field of a Midi sample Dump (the 8th, 9th, and 10th bytes of the Dump Header LSB first) which we will call MidiSamplePer, and the Rate of a SAMP file, here is the relationship:

```
   Rate = (1/MidiSamplePer) x 10E9
```

Also the number of samples (wave's length) in a Midi Sample Dump (the 11th, 12th, and 13th bytes of the Dump header) is expressed in WORDS. SAMP's WaveSize is expressed in the number of BYTES. (For the incredibly stupid), the relationship is:

```
   WaveSize = MidiSampleLength x 2
```

A Midi sample dump's LoopStart point and LoopEnd point are also in WORDS as versus the SAMP equivalents expressed in BYTES.

A Midi sample dump's sample number can be 0 to 65535. A SAMP file can hold up to 255 waves, and their numbers in the playmap must be 1 to 255. (A single, Midi Sample Dump only sends info on one wave.) When recieving a Midi Sample Dump, just store the sample number (5th and 6th bytes of the Dump Header LSB first) in SAMP's MidiSampNum field. Then forget about this number until you need to send the wave back to the Midi instrument from whence it came.

A Midi Dump's loop type can be forward, or forward/backward. Amiga hardware supports forward only. You should store the Midi Dump's LoopType byte here, but ignore it otherwise until/unless Amiga hardware supports "reading audio data" in various ways. If so, then the looptype is as follows:

```
    forward = 0, backward/forward = 1
```

A Midi Dump's sample format byte is the same as SAMP's.

1.151 SAMP.doc / Interpreting the Playmode

PlayMode specifies how the bytes in the PlayMap are to be interpreted. Remember that a PlayMap byte of 0 means "No Wave to Play".

```
    #define INDEPENDANT 0
    #define MULTI       1
    #define STEREO      2
    #define PAN         3
```

```
PlayMode types:

 INDEPENDANT(0)-The wave #s for a midi note are to be output on
               Amiga audio channels 0, 1, 2, and 3 respectively.
               If the NumOfChans is < 4, then only use that many channels.

 MULTI      (1)-The first wave # (first of the PlayMap bytes) for a
               midi note is to be output on any free channel. The other
               wave numbers are ignored. If all four channels are in
               play, the application can decide whether to "steal" a
               channel.

 STEREO     (2)-The first wave # (first of the PlayMap bytes) is to be
               output of the Left stereo jack (channel 1 or 3) and if
               there is a second wave number (the second of the PlayMap
               bytes), it is to be output the Right jack (channel 2 or 4).
               The other wave numbers are ignored.

 PAN        (3)-This is just like STEREO except that the volume of wave 1
               should start at its initial volume (midi velocity) and
               fade to 0. At the same rate, wave 2 should start at 0
               volume and rise to wave #1's initial level. The net
               effect is that the waves "cross" from Left to Right in
               the stereo field. This is most effective when the wave
               numbers are the same. (ie the same wave) The application
               program should set the rate. Also, the application can
               reverse the stereo direction (ie Right to Left fade).

The most important wave # to be played back by a midi note should be the
first of the PlayMap bytes. If the NumOfChans > 1, the second PlayMap byte
should be a defined wave number as well (even if it is deliberately set to
the same value as the first byte). This insures that all 4 PlayModes will
have some effect on a given SAMP file. Also, an application should allow
the user to change the PlayMode at will. The PlayMode stored in the SAMP
file is only a default or initial set-up condition.
```

1.152 SAMP.doc / Making A Transpose Table

In order to allow a wave to playback over a range of musical notes, (+/- semitones), its playback rate must be raised or lowered by a set amount. From one semitone to the next, this set amount is by a factor of the 12th root of 2 (assuming a western, equal-tempered scale). Here is a table that shows what factor would need to be multiplied by the sampling rate in order to transpose the wave's pitch.

Pitch in relation to the Root	Multiply Rate by this amount
DOWN 6 semitones	0.5
DOWN 5 1/2 semitones	0.529731547
DOWN 5 semitones	0.561231024
DOWN 4 1/2 semitones	0.594603557
DOWN 4 semitones	0.629960525
DOWN 3 1/2 semitones	0.667419927
DOWN 3 semitones	0.707106781
DOWN 2 1/2 semitones	0.749153538

```
    DOWN 2     semitones              0.793700526
    DOWN 1 1/2 semitones              0.840896415
    DOWN 1     semitones              0.890898718
    DOWN 1/2   semitone               0.943874312
  ORIGINAL_PITCH                      1.0           /* rootnote's pitch */
    UP   1/2   semitone               1.059463094
    UP   1     semitones              1.122562048
    UP   1 1/2 semitones              1.189207115
    UP   2     semitones              1.259921050
    UP   2 1/2 semitones              1.334839854
    UP   3     semitones              1.414213562
    UP   3 1/2 semitones              1.498307077
    UP   4     semitones              1.587401052
    UP   4 1/2 semitones              1.681792830
    UP   5     semitones              1.781797436
    UP   5 1/2 semitones              1.887748625
    UP   6     semitones              2
```

For example, if the wave's Rate is 18000 hz, and you wish to play the wave UP 1 semitone, then the playback rate is:

```
    18000 x 1.122562048 = 20206.11686 hz
```

The sampling period for the Amiga is therefore:

```
    (1/20206.11686)/.279365 = .000177151
```

and to send it to the Audio Device, it is rounded and expressed in micro-seconds: 177

Obviously, this involves floating point math which can be time consuming and impractical for outputing sound in real-time. A better method is to construct a transpose table that contains the actual periods already calculated for every semitone. The drawback of this method is that you need a table for EVERY DIFFERENT Rate in the SAMP file. If all the Rates in the file happened to be the same, then only one table would be needed. Let's assume that this is the case, and that the Rate = 18000 hz. Here is a table containing enough entries to transpose the waves +/- 6 semitones.

```
  Pitch in relation to the Root     Amiga Period (assuming rate = 18000 hz)
  -----------------------------     ---------------------------------------
    Transposition_table[TRANS_TABLE_SIZE]={
    /* DOWN 6     semitones  */            398,
    /* DOWN 5 1/2 semitones  */            375,
    /* DOWN 5     semitones  */            354,
    /* DOWN 4 1/2 semitones  */            334,
    /* DOWN 4     semitones  */            316,
    /* DOWN 3 1/2 semitones  */            298,
    /* DOWN 3     semitones  */            281,
    /* DOWN 2 1/2 semitones  */            265,
    /* DOWN 2     semitones  */            251,
    /* DOWN 1 1/2 semitones  */            236,
    /* DOWN 1     semitones  */            223,
    /* DOWN 1/2   semitone   */            211,
    /* ORIGINAL_PITCH        */            199,    /* rootnote's pitch */
    /* UP   1/2   semitone   */            187,
    /* UP   1     semitones  */            177,
```

```
    /* UP   1 1/2 semitones  */           167,
    /* UP   2     semitones  */           157,
    /* UP   2 1/2 semitones  */           148,
    /* UP   3     semitones  */           141,
    /* UP   3 1/2 semitones  */           133,
     /* Since the minimum Amiga period = 127 the following
        are actually out of range. */
    /* UP   4     semitones  */           125,
    /* UP   4 1/2 semitones  */           118,
    /* UP   5     semitones  */           112,
    /* UP   5 1/2 semitones  */           105,
    /* UP   6     semitones  */           99   };
```

Let's assume that (according to the PlayMap) midi note #40 is set to play wave number 3. Upon examining wave 3's structure, we discover that the Rate = 18000, and the RootNote = 38. Here is how the Amiga sampling period is calulated using the above 18000hz "transpose chart" in C:

```
   /* MidiNoteNumber is the received midi note's number (here 40) */

   #define ORIGINAL_PITCH     TRANS_TABLE_SIZE/2 + 1
  /* TRANS_TABLE_SIZE is the number of entries in the transposition table
     (dynamic, ie this can change with the application) */

  transposeAmount = (LONG) (MidiNoteNumber - rootNote);
                                          /* make it a SIGNED LONG */

  amigaPeriod     = Transposition_table[ORIGINAL_PITCH + transposeAmount];
```

In assembly, the 18000hz transpose chart and above example would be:

```
Table       dc.w  398
            dc.w  375
            dc.w  354
            dc.w  334
            dc.w  316
            dc.w  298
            dc.w  281
            dc.w  265
            dc.w  251
            dc.w  236
            dc.w  223
            dc.w  211
ORIGINAL_PITCH  dc.w  199   ; rootnote's pitch
            dc.w  187
            dc.w  177
            dc.w  167
            dc.w  157
            dc.w  148
            dc.w  141
            dc.w  133
 ; Since the minimum Amiga period = 127, the following
 ; are actually out of range.
            dc.w  125
```

```
            dc.w  118
            dc.w  112
            dc.w  105
            dc.w  99

  lea     ORIGINAL_PITCH,a0
  move.b  MidiNoteNumber,d0  ;the received note number
  sub.b   RootNote,d0        ;subtract the wave's root note
  ext.w   d0
  ext.l   d0                 ;make it a signed LONG
  add.l   d0,d0              ;x 2 in order to fetch a WORD
  adda.l  d0,a0
  move.w  (a0),d0            ;the Amiga Period (WORD)
```

Note that these examples don't check to see if the transpose amount is beyond the number of entries in the transpose table. Nor do they check if the periods in the table are out of range of the Amiga hardware.

1.153 SAMP.doc / Making the Velocity Table

The 16 entries in the velocity table should be within the oneShot portion of the sample (ie not in the looping portion). THe first offset, VelTable[0] should be set to zero (in order to play back from the beginning of the data). The subsequent values should be increasing numbers. If you are using a graphic editor, try choosing offsets that will keep you within the initial attack portion of the wave. In practice, these values will be relatively close together within the wave. Always set the offsets so that when they are added to the sample start point, the resulting address points to a sample value of zero (a zero crossing point). This will eliminate pops and clicks at the beginning of the playback.

In addition, the start of the wave should be on a sample with a value of zero. The last sample of the oneShot portion and the first sample of the looping portion should be approximately equal, (or zero points). The same is true of the first and last samples of the looping portion. Finally, try to keep the slopes of the end of the oneShot, the beginning of the looping, and the end of the looping section, approximately equal. All this will eliminate noise on the audio output and provide "seamless" looping.

1.154 SAMP.doc / The Instrument Type

Many SMUS players search for certain instruments by name. Not only is this slow (comparing strings), but if the exact name can't be found, then it is very difficult and time-consuming to search for a suitable replacement. For this reason, many SMUS players resort to "default" instruments even if these are nothing like the desired instruments. The InsType byte in each waveHeader is meant to be a numeric code which will tell an SMUS player exactly what the instrument is. In this way, the SMUS player can search for the correct "type" of instrument if it can't find the desired name. The type byte is divided into 2 nibbles (4 bits for you

C programmers) with the low 4 bits representing the instrument "family" as follows:

  1 = STRING, 2 = WOODWIND, 3 = KEYBOARD, 4 = GUITAR, 5 = VOICE,
  6 = DRUM1, 7 = DRUM2, 8 = PERCUSSION1, 9 = BRASS1, A = BRASS2,
  B = CYMBAL, C = EFFECT1, D = EFFECT2, E = SYNTH,
  F is undefined at this time

Now, the high nibble describes the particular type within that family.

For the STRING family, the high nibble is as follows:

  1 = VIOLIN BOW, 2 = VIOLIN PLUCK, 3 = VIOLIN GLISSANDO,
  4 = VIOLIN TREMULO, 5 = VIOLA BOW, 6 = VIOLA PLUCK, 7 = VIOLA GLIS,
  8 = VIOLA TREM, 9 = CELLO BOW, A = CELLO PLUCK, B = CELLO GLIS,
  C = CELLO TREM, D = BASS BOW, E = BASS PLUCK (jazz bass), F = BASS TREM

For the BRASS1 family, the high nibble is as follows:

  1 = BARITONE SAX, 2 = BARI GROWL, 3 = TENOR SAX, 4 = TENOR GROWL,
  5 = ALTO SAX, 6 = ALTO GROWL, 7 = SOPRANO SAX, 8 = SOPRANO GROWL,
  9 = TRUMPET, A = MUTED TRUMPET, B = TRUMPET DROP, C = TROMBONE,
  D = TROMBONE SLIDE, E = TROMBONE MUTE

For the BRASS2 family, the high nibble is as follows:

  1 = FRENCH HORN, 2 = TUBA, 3 = FLUGAL HORN, 4 = ENGLISH HORN

For the WOODWIND family, the high nibble is as follows:

  1 = CLARINET, 2 = FLUTE, 3 = PAN FLUTE, 4 = OBOE, 5 = PICCOLO,
  6 = RECORDER, 7 = BASSOON, 8 = BASS CLARINET, 9 = HARMONICA

For the KEYBOARD family, the high nibble is as follows:

  1 = GRAND PIANO, 2 = ELEC. PIANO, 3 = HONKYTONK PIANO, 4 = TOY PIANO,
  5 = HARPSICHORD, 6 = CLAVINET, 7 = PIPE ORGAN, 8 = HAMMOND B-3,
  9 = FARFISA ORGAN, A = HARP

For the DRUM1 family, the high nibble is as follows:

  1 = KICK, 2 = SNARE, 3 = TOM, 4 = TIMBALES, 5 = CONGA HIT,
  6 = CONGA SLAP, 7 = BRUSH SNARE, 8 = ELEC SNARE, 9 = ELEC KICK,
  A = ELEC TOM, B = RIMSHOT, C = CROSS STICK, D = BONGO, E = STEEL DRUM,
  F = DOUBLE TOM

For the DRUM2 family, the high nibble is as follows:

  1 = TIMPANI, 2 = TIMPANI ROLL, 3 = LOG DRUM

For the PERCUSSION1 family, the high nibble is as follows:

  1 = BLOCK, 2 = COWBELL, 3 = TRIANGLE, 4 = TAMBOURINE, 5 = WHISTLE,
  6 = MARACAS, 7 = BELL, 8 = VIBES, 9 = MARIMBA, A = XYLOPHONE,
  B = TUBULAR BELLS, C = GLOCKENSPEIL

For the CYMBAL family, the high nibble is as follows:

```
  1 = CLOSED HIHAT, 2 = OPEN HIHAT, 3 = STEP HIHAT, 4 = RIDE,
  5 = BELL CYMBAL, 6 = CRASH, 7 = CHOKE CRASH, 8 = GONG, 9 = BELL TREE,
  A = CYMBAL ROLL

 For the GUITAR family, the high nibble is as follows:

  1 = ELECTRIC, 2 = MUTED ELECTRIC, 3 = DISTORTED, 4 = ACOUSTIC,
  5 = 12-STRING, 6 = NYLON STRING, 7 = POWER CHORD, 8 = HARMONICS,
  9 = CHORD STRUM, A = BANJO, B = ELEC. BASS, C = SLAPPED BASS,
  D = POPPED BASS, E = SITAR, F = MANDOLIN
   (Note that an acoustic picked bass is found in the STRINGS - Bass Pluck)

For the VOICE family, the high nibble is as follows:

  1 = MALE AHH, 2 = FEMALE AHH, 3 = MALE OOO, 4 = FEMALE OOO,
  5 = FEMALE BREATHY, 6 = LAUGH, 7 = WHISTLE

For the EFFECTS1 family, the high nibble is as follows:

  1 = EXPLOSION, 2 = GUNSHOT, 3 = CREAKING DOOR OPEN, 4 = DOOR SLAM,
  5 = DOOR CLOSE, 6 = SPACEGUN, 7 = JET ENGINE, 8 = PROPELLER,
  9 = HELOCOPTER, A = BROKEN GLASS, B = THUNDER, C = RAIN, D = BIRDS,
  E = JUNGLE NOISES, F = FOOTSTEP

For the EFFECTS2 family, the high nibble is as follows:

  1 = MACHINE GUN, 2 = TELEPHONE, 3 = DOG BARK, 4 = DOG GROWL,
  5 = BOAT WHISTLE, 6 = OCEAN, 7 = WIND, 8 = CROWD BOOS, 9 = APPLAUSE,
  A = ROARING CROWDS, B = SCREAM, C = SWORD CLASH, D = AVALANCE,
  E = BOUNCING BALL, F = BALL AGAINST BAT OR CLUB

For the SYNTH family, the high nibble is as follows:

  1 = STRINGS, 2 = SQUARE, 3 = SAWTOOTH, 4 = TRIANGLE, 5 = SINE, 6 = NOISE

So, for example if a wave's type byte was 0x26, this would be a SNARE
DRUM. If a wave's type byte is 0, then this means "UNKNOWN" instrument.
```

1.155 SAMP.doc / The Order of the Chunks

```
The SAMP header obviously must be first in the file, followed by the MHDR
chunk. After this, the ANNO, (c), AUTH and NAME chunks may follow in any
order, though none of these need appear in the file at all. The BODY chunk
must be last.
```

1.156 SAMP.doc / Filename Conventions

```
For when it becomes necessary to split a SAMP file between floppies using
the Continuation feature, the filenames should be related. The method is
the following:
```

The "root" file has the name that the user chose to save under. Subsequent files have an ascii number appended to the name to indicate what sublevel the file is in. In this way, a program can reload the files in the proper order.

For example, if a user saved a file called "Gurgle", the first continuation file should be named "Gurgle1", etc.

1.157 SAMP.doc / Why Does Anyone Need Such a Complicated File?

(or "What's wrong with 8SVX anyway?")

In a nutshell, 8SVX is not adequate for professional music sampling. First of all, it is nearly impossible to use multi-sampling (utilizing several, different samples of any instrument throughout its musical range). This very reason alone makes it impossible to realistically reproduce a musical in- strument, as none in existance (aside from an electronic organ) uses inter- polations of a single wave to create its musical note range.

Also, stretching a sample out over an entire octave range does grotesque (and VERY unmusical) things to such elements as the overtone structure, wind/percussive noises, the instrument's amplitude envelope, etc. The 8SVX format is designed to stretch the playback in exactly this manner.

8SVX ignores MIDI which is the de facto standard of musical data transmission. 8SVX does not allow storing data for features that are commonplace to pro- fessional music samplers. Such features are: velocity sample start, separate filter and envelopes for each sample, separate sampling rates, and various playback modes like stereo sampling and panning.

SAMP attempts to remedy all of these problems with a format that can be used by a program that simulates these professional features in software. The format was inspired by the capabilities of the following musical products:

EMU's	EMAX, EMULATOR
SEQUENTIAL CIRCUIT's	PROPHET 2000, STUDIO 440
ENSONIQ's	MIRAGE
CASIO's	FZ-1
OBERHEIM's	DPX
YAMAHA	TX series

So why does the Amiga need the SAMP format? Because professional musician's are buying computers. With the firm establishment of MIDI, musician's are buying and using a variety of sequencers, patch editors, and scoring programs. It is now common knowledge amoung professional musicians that the Amiga lags far behind IBM clones, Macintosh, and Atari ST computers in both music software and hardware support. Both Commodore and the current crop of short- sighted 3rd party Amiga developers are pigeon-holing the Amiga as "a video computer". It is important for music software to exploit whatever capabili- ties the Amiga offers before the paint and animation programs, genlocks, frame-grabbers, and video breadboxes are the only applications selling for the Amiga. Hopefully, this format, with the SAMP disk I/O library will make it possible for Amiga software to attain the level of professionalism that the other

machines now boast, and the Amiga lacks.

1.158 A / IFF Third Party Public Form and Chunk Specification / TDDD.doc

```
3-D rendering data, Turbo Silver (Impulse)

                                FORM TDDD
                                ---------
FORM TDDD is used by Impulse's Turbo Silver 3.0 for 3D rendering data.
TDDD stands for "3D data description".  The files contain object and
(optionally) observer data.

Turbo Silver's successor, "Imagine", uses an upgraded FORM TDDD when it
reads/writes object data.

Currently, in "standard IFF" terms, a FORM TDDD has only two chunk types:
an INFO chunk describing observer data;  and an OBJ chunk describing an
object heirarchy.  The INFO chunk appears only in Turbo Silver's "cell"
files, and the OBJ chunk appears in both "cell" files and "object" files.

The FORM has an (optional) INFO chunk followed by some number of OBJ
chunks.  (Note:  OBJ is followed by a space - ckID = "OBJ ")

The INFO and OBJ chunks, in turn, are made up of smaller chunks with the
standard IFF structure:  <ID> <data-size> <data>.

The INFO "sub-chunks" are relatively straightforward to interpret.

The OBJ "sub-chunks" support object heirarchies, and are slightly more
difficult to interpret.  Currently, there are 3 types of OBJ sub-chunks:
an EXTR chunk, describing an "external" object in a seperate file; a DESC
chunk, describing one node of a heirarchy; and a TOBJ chunk marking the
end of a heirarchy chain.  For each DESC chunk, there must be a
corresponding TOBJ chunk.  And an EXTR chunk is equivalent to a DESC/TOBJ
pair.

In Turbo Silver and Imagine, the structure of the object heirarchy is as
follows.  There is a head object, and its (sexist) brothers. Each brother
may have child objects.  The children may have grandchildren, and so on.
The brother nodes are kept in a doubly linked list, and each node has a
(possibly NULL) pointer to a doubly linked "child" list. The children
point to the "grandchildren" lists, and so on.  (In addition, each node
has a "back" pointer to its parent).

Each of the "head" brothers is written in a seperate OBJ chunk, along with
all its descendants.  The descendant heirarchy is supported as follows:

    for each node of a doubly linked list,

    1)  A DESC chunk is written, describing its object.
    2)  If it has children, steps 1) to 3) are performed
            for each child.
    3)  A TOBJ chunk is written, marking the end of the children.

For "external" objects, steps 1) to 3) are not performed, but an EXTR
```

chunk is written instead. (This means that an external object cannot have children unless they are stored in the same "external" file).

The TOBJ sub-chunks have zero size -- and no data. The DESC and EXTR sub-chunks are made up of "sub-sub-chunks", again, with the standard IFF structure: <ID> <data-size> <data>.

("External" objects were used by Turbo Silver to allow a its "cell" data files to refer to an "object" data file that is "external" to the cell file. Imagine abandons the idea of individual cell files, and deals only in TDDD "object" files. Currently, Imagine does not support EXTR chunks in TDD files.)

Reader software WILL FOLLOW the standard IFF procedure of skipping over any un-recognized chunks -- and "sub-chunks" or "sub-sub-chunks". The <data-size> field indicates how many bytes to skip. In addition it WILL OBSERVE the IFF rule that an odd <data-size> may appear, in which case the corredponding <data> field will be padded at the end with one extra byte to give it an even size.

Now, on with the details

1.159 TDDD.doc / Now, on with the details

First, there are several numerical fields appearing in the data, describing object positions, rotation angles, scaling factors, etc. They are stored as "32-bit fractional" numbers, such that the true number is the 32-bit number divided by 65536. So as an example, the number 3.14159 is stored as (hexadecimal) $0003243F. This allows the data to be independant of any particular floating point format. And it (actually) is the internal format used in the "integer" version of Turbo Silver. Numbers stored in this format are called as "FRACT"s below.

Second, there are several color (or RGB) fields in the data. They are always stored as three UBYTEs representing the red, green and blue components of the color. Red is always first, followed by green, and then blue. For some of the data chunks, Turbo Silver reads the color field into the 24 LSB's of a LONGword. In such cases, the 3 RGB bytes are preceded by a zero byte in the file.

The following "typedef"s are used below:

```
    typedef LONG    FRACT;                /* 4 bytes */
    typedef UBYTE   COLOR[3];             /* 3 bytes */

    typedef struct vectors {
        FRACT X;          /* 4 bytes */
        FRACT Y;          /* 4 bytes */
        FRACT Z;          /* 4 bytes */
    } VECTOR;             /* 12 bytes total */

    typedef struct matrices {
        VECTOR I;         /* 12 bytes */
        VECTOR J;         /* 12 bytes */
        VECTOR K;         /* 12 bytes */
```

```
    } MATRIX;             /* 36 bytes total */

    typedef struct _tform {
        VECTOR r;         /* 12 bytes - position */
        VECTOR a;         /* 12 bytes - x axis */
        VECTOR b;         /* 12 bytes - y axis */
        VECTOR c;         /* 12 bytes - z axis */
        VECTOR s;         /* 12 bytes - size */
    } TFORM;              /*  60 bytes total */
```

The following structure is used in generating animated cells from a single cell. It can be attached to an object or to the camera. It is also used for Turbo Silver's "extrude along a path" feature. (It is ignored and forgotten by Imagine.)

```
    typedef struct story {
        UBYTE  Path[18];  /* 18 bytes */
        VECTOR Translate; /* 12 bytes */
        VECTOR Rotate;    /* 12 bytes */
        VECTOR Scale;     /* 12 bytes */
        UWORD  info;      /*  2 bytes */
    } STORY;              /* 56 bytes total */
```

The Path[] name refers to a named object in the cell data. The path object should be a sequence of points connected with edges. The object moves from the first point of the first edge, to the last point of the last edge. The edge ordering is important. The path is interpolated so that the object always moves an equal distance in each frame of the animation. If there is no path the Path[] field should be set to zeros. The Translate vector is not currently used. The Rotate "vector" specifies rotation angles about the X, Y, and Z axes. The Scale vector specfies X,Y, and Z scale factors. The "info" word is a bunch of bit flags:

```
    ABS_TRA    0x0001    - translate in world coorinates (not used)
    ABS_ROT    0x0002    - rotation in world coorinates
    ABS_SCL    0x0004    - scaling in world coorinates
    LOC_TRA    0x0010    - translate in local coorinates (not used)
    LOC_ROT    0x0020    - rotation in local coorinates
    LOC_SCL    0x0040    - scaling in local coorinates
    X_ALIGN    0x0100    - (not used)
    Y_ALIGN    0x0200    - align Y axis to path's direction
    Z_ALIGN    0x0400    - (not used)
    FOLLOW_ME  0x1000    - children follow parent on path
```

```
 DESC sub-sub-chunks
 DESC notes
 INFO sub-chunks
 EXTR sub-sub-chunks
```

1.160 Details / DESC sub-sub-chunks

```
NAME - size 18

    BYTE    Name[18];       ; a name for the object.
```

```
    Used for camera tracking, specifying story paths, etc.

SHAP - size 4

    WORD    Shape;          ; number indicating object type
    WORD    Lamp;           ; number indicating lamp type

    Lamp numbers are composed of several bit fields:

            Bits 0-1:
        0 - not a lamp
        1 - like sunlight
        2 - like a lamp - intensity falls off with distance.
        3 - unused/reserved

            Bits 2:
                    0 - non-shadow-casting light
                    4 - shadow-casting light

            Bits 3-4:
        0  - Spherical light source
        8  - Cylindrical light source.
        16 - Conical light source.
        24 - unused/reserved

    Shape numbers are:

        0 - Sphere
        1 - Stencil         ; not supported by Imagine
        2 - Axis            ; custom objects with points/triangles
        3 - Facets          ; illegal - for internal use only
        4 - Surface         ; not supported by Imagine
        5 - Ground

    Spheres have thier radius set by the X size parameter.
    Stencils and surfaces are plane-parallelograms, with one
    point at the object's position vector; one side lying along
    the object's X axis with a length set by the X size; and
    another side starting from the position vector and going
    "Y size" units in the Y direction and "Z size" units in
    the X direction.  A ground object is an infinte plane
    perpendicular to the world Z axis.  Its Z coordinate sets
    its height, and the X and Y coordinates are only relevant
    to the position of the "hot point" used in selecting the
    object in the editor.  Custom objects have points, edges
    and triangles associated with them.  The size fields are
    relevant only for drawing the object axes in the editor.
    Shape number 3 is used internally for triangles of custom
    objects, and should never appear in a data file.

POSI - size 12

    VECTOR  Position;       ; the object's position.

    Legal coordinates are in the range -32768 to 32767 and 65535/65536.
    Currently, the ray-tracer only sees objects in the -1024 to 1024
    range.  Light sources, and the camera may be placed outside that
```

```
    range, however.

AXIS - size 36

    VECTOR  XAxis;
    VECTOR  YAxis;
    VECTOR  ZAxis;

    These are direction vectors for the object coordinate system.
    They must be "orthogonal unit vectors" - i.e. the sum of the
    squares of the vevtor components must equal one (or close to it),
    and the vectors must be perpendicular.

SIZE - size 12

    VECTOR  Size;

    See SHAP chunk above.  The sizes are used in a variety of ways
    depending on the object shape.  For custom objects, they are
    the lengths of the coordinate axes drawn in the editor.  If the
    object has its "Quickdraw" flag set, the axes lengths are also
    used to set the size of a rectangular solid that is drawn rather
    than drawing all the points and edges.

PNTS - size 2 + 12 * point count

    UWORD   PCount;         ; point count
    VECTOR  Points[];       ; points

    This chunk has all the points for custom objects.  The are
    refered to by thier position in the array.

EDGE - size 4 + 4 * edge cout

    UWORD   ECount;         ; edge count
    UWORD   Edges[][2];     ; edges

    This chunk contins the edge list for custom objects.
    The Edges[][2] array is pairs of point numbers that
    are connected by the edges.  Edges are refered to by thier
    position in the Edges[] array.

FACE - size 2 + 6 * face count

    UWORD   TCount;         ; face count
    UWORD   Connects[][3];  ; faces

    This chunk contains the triangle (face) list for custom objects.
    The Connects[][3] array is triples of edge numbers that are
    connected by triangles.

PTHD - size 2 + 6 * axis count - Imagine only

    UWORD   ACount;         ; axis count
    TFORM   PData[][3];     ; axis data

    This chunk contains the axis data for Imagine "path" objects.
```

```
    The PData array contains a TFORM structure for each point along
    the path.  The "Y size" item for the last point on the path tells
    whether the path is closed or not.  Zero means closed, non-zero
    means open.  Otherwise the Y size field is the distance along
    the path to the next path point/axis.

COLR - size 4
REFL - size 4
TRAN - size 4
SPC1 - size 4 - Imagine only

    BYTE    pad;            ; pad byte - must be zero
    COLOR   col;            ; RGB color

    These are the main object RGB color, and reflection, transmission
    and specularity coefficients.

CLST - size 2 + 3 * count
RLST - size 2 + 3 * count
TLST - size 2 + 3 * count

    UWORD   count;          ; count of colors
    COLOR   colors[];       ; colors

    These are the color, reflection and transmission coefficients
    for each face in custom objects. The count should match the
    face count in the FACE chunk. The ordering corresponds to the
    face order.

TPAR - size 64 - not written by Imagine - see TXT1 below

    FRACT   Params[16];     ; texture parameters

    This is the list of parameters for texture modules when
    texture mapping is used.
TXT1 - variable size - Imagine only

    This chunk contains texture data when texture mapping is used.

    UWORD   Flags;          ; texture flags:
                            ;    1 - TXTR_CHILDREN - apply to child objs
    TFORM   TForm;          ; local coordinates of texture axes.
    FRACT   Params[16];     ; texture parameters
    UBYTE   PFlags[16];     ; parameter flags (currently unused)
    UBYTE   Length;         ; length of texture file name
    UBYTE   Name[Length];   ; texture file name (not NULL terminated)
    UBYTE   pad;            ; (if necessary to make an even length)

BRS1 - variable size - Imagine only (version 1.0)
BRS2 - variable size - Imagine only (version 1.1)

    UWORD   Flags;          ; brush type:
                            ;    0 - Color
                            ;    1 - Reflection
                            ;    2 - Filter
                            ;    3 - Altitude
    UWORD   WFlags;         ; brush wrapping flags:
```

```
                            ;    1   WRAP_X        - wrap type
                            ;    2   WRAP_Z        - wrap type
                            ;    4   WRAP_CHILDREN - apply to children
                            ;    8   WRAP_REPEAT   - repeating brush
                            ;    16  WRAP_FLIP     - flip with repeats
    TFORM   TForm;          ; local coordinates of brush axes.
    (UWORD   FullScale;)    ; full scale value
    (UWORD   MaxSeq;)       ; highest number for sequenced brushes
    UBYTE   Length;         ; length of brush file name
    UBYTE   Name[Length];   ; brush file name (not NULL terminated)
    UBYTE   pad;            ; (if necessary to make an even length)

    The FullScale and MaxSeq items are in BRS2 chunks only.

SURF - size 5 - not written by Imagine

    BYTE    SProps[5];      ; object properties

    This chunk contains object (surface) properties used by Turbo Silver.

    SProps[0] - PRP_SURFACE ; surface type
                            ;   0 - normal
                            ;   4 - genlock
                            ;   5 - IFF brush
    SProps[1] - PRP_BRUSH   ; brush number (if IFF mapped)
    SProps[2] - PRP_WRAP    ; IFF brush wrapping type
                            ;   0 - no wrapping
                            ;   1 - wrap X
                            ;   2 - wrap Z
                            ;   3 - wrap X and Z
    SProps[3] - PRP_STENCIL ; stencil number for stencil objects
    SProps[4] - PRP_TEXTURE ; texture number if texture mapped

MTTR - size 2 - not written by Imagine - see PRP1 chunk.

    UBYTE   Type;           ; refraction type (0-4)
    UBYTE   Index;          ; custom index of refraction

    This chunk contains refraction data for transparent or glossy
    objects.  If the refraction type is 4, the object has a "custom"
    refractive index stored in the Index field.  The Index field is
    100 * (true index of refraction - 1.00) -- so it must be in the
    range of 1.00 to 3.55.  The refraction types 0-3 specify 0) Air
    - 1.00, 1) Water - 1.33, 2) Glass - 1.67 or 3) Crystal 2.00.

SPEC - size 2 - not written by Imagine - see SPC1 above.

    UBYTE   Specularity;    ; range of 0-255
    UBYTE   Hardness;       ; specular exponent (0-31)

    This chunk contains specular information.  The Specularity
    field is the amount of specular reflection -- 0 is none,
    255 is fully specular.  The "specular exponent" controls
    the "tightness" of the specular spots.  A value of zero
    gives broad specular spots and a value of 31 gives smaller
    spots.
```

```
PRP0 - size 6 - not written by Imagine

    UBYTE   Props[6];       ; more object properties

    This chunk contains object properties that programs other
    than Turbo Silver might support.

    Props[0] - PRP_BLEND    ; blending factor (0-255)
    Props[1] - PRP_SMOOTH   ; roughness factor
    Props[2] - PRP_SHADE    ; shading on/off flag
    Props[3] - PRP_PHONG    ; phong shading on/off flag
    Props[4] - PRP_GLOSSY   ; glossy on/off flag
    Props[5] - PRP_QUICK    ; Quickdraw on/off flag

    The blending factor controls the amount of dithering used
    on the object - 255 is fully dithered.  The roughness factor
    controls how rough the object should appear - 0 is smooth, 255
    is max roughness.  The shading flag is interpreted differently
    depending on whether the object is a light source or not.  For
    light sources, it sets the light to cast shadows or not.  For
    normal objects, if the flag is set, the object is always
    considered as fully lit - i.e., it's color is read directly
    from the object (or IFF brush), and is not affected by light
    sources.  The phong shading is on by default - a non-zero value
    turns it off.  The glossy flag sets the object to be glossy or
    not.  If the object is glossy, the "transmit" colors and the
    index of refraction control the amount of "sheen".  The glossy
    feature is meant to simulate something like a wax coating
    on the object with the specified index of refraction. The
    trasmission coefficients control how much light from the
    object makes it through the wax coating.
    The Quickdraw flag, if set, tells the editor not to draw
    all the points and edges for the object, but to draw a
    rectanglular solid centered at the object position, and
    with sizes detemined by the axis lengths.

PRP1 - size 8 - Imagine only

    UBYTE   IProps[8];      ; more object properties

    This chunk contains object properties that programs other
    than Imagine might support.

    IProps[0] - IPRP_DITHER   ; blending factor (0-255)
    IProps[1] - IPRP_HARD     ; hardness factor (0-255)
    IProps[2] - IPRP_ROUGH    ; roughness factor (0-255)
    IProps[3] - IPRP_SHINY    ; shinyness factor (0-255)
    IProps[4] - IPRP_INDEX    ; index of refraction
    IProps[5] - IPRP_QUICK    ; flag - Quickdraw on/off
    IProps[6] - IPRP_PHONG    ; flag - Phong shading on/off
    IProps[7] - IPRP_GENLOCK  ; flag - Genlock on/off

    The blending factor controls the amount of dithering used
    on the object - 255 is fully dithered.
    The hardness factor controls how tight the specular spot
    should be - 0 is a big soft spot, 255 is a tight hot spot
    The roughness factor controls how rough the object should
```

```
    appear - 0 is smooth, 255 is max roughness.
    The shiny factor in interaction with the object's filter
    values controls how shiny the object appears.  Setting it
    to anything but zero forces the object to be non-transparent
    since then the filter values are used in the shiny (reflection)
    calculations.  A value of 255 means maximum shinyness.

INTS - size 4 - not written by Imagine

    FRACT   Intensity;      ; light intensity

    This is the intensity field for light source objects.
    an intensity of 255 for a sun-like light fully lights
    object surfaces which are perpendicular to the direction
    to the light source.  For lamp-like light sources, the
    necessary intensity will depend on the distance to the light.

INT1 - size 12 - Imagine only

    VECTOR  Intensity;      ; light intensity

    This is like INTS above, but has seperate R, G & B intensities.

STRY - size 56 - not written by Imagine

    STORY   story;          ; a story structure for the object.

    The story structure is described above.

ANID - size 64 - Imagine only

    LONG    Cellno;         ; cell number
    TFORM   TForm;          ; object position/axes/size in that cell.

    For Imagine's "Cycle" objects, within EACH DESC chunk in the
    file - that is, for each object of the group, there will be
    a series of ANID chunks.  The cell number sequences of each
    part of the must agree with the sequence for the head object,
    and the first cell number must be zero.

FORD - size 56 + 12 * PC - Imagine only

    WORD    NumC;           ; number of cross section points
    WORD    NumF;           ; number of slices
    WORD    Flags;          ; orientation flag
    WORD    pad;            ; reserved
    MATRIX  TForm;          ; object rotation/scaling transformation
    VECTOR  Shift;          ; object translation
    VECTOR  Points[PC];     ; "Forms" editor points

    For Imagine's "Forms" objects, the "PNTS" chunk above is not
    written out, but this structure is written instead.  The point
    count is PC = NumC + 4 * NumF.  The object's real points are
    then calculated from these using a proprietary algorithm.
    The tranformation parameters above allow the axes of the
    real object be moved around relative to the "Forms" points.
```

1.161 Details / DESC notes

```
Again, most of these fields are optional, and defaults are supplied.
However, if there is a FACE chunk, there must also be a CLST chunk, an
RLST chunk and a TLST chunk -- all with matching "count" fields. The SHAP
chunk is not optional.

Defaults are:  Colors set to (240,240,240); reflection and transmission
coefficients set to zero; illegal shape; no story or special surface
types; position at (0,0,0); axes aligned to the world axes; size fields
all 32.0; intensity at 300; no name; no points/edges or faces; texture
parameters set to zero; refraction type 0 with index 1.00; specular,
hardness and roughness set to zero; blending at 255; glossy off; phong
shading on; not a light source; not brightly lit;

In Imagine, defaults are the same, but with colors (255,255,255).
```

1.162 Details / INFO sub-chunks

```
BRSH - size 82

    WORD    Number;        ; Brush number (between 0 and 7)
    CHAR    Filename[80];  ; IFF ILBM filename

    There may be more than one of these.

STNC - size 82

    Same format as BRSH chunk.

TXTR - size 82

    Same format as BRSH chunk.  The Filename field is the name of
    a code module that can be loaded with LoadSeg().

OBSV - size 28

    VECTOR  Camera;         ; Camera position
    VECTOR  Rotate;         ; Camera rotation angles
    FRACT   Focal;          ; Camera focal length

    This tells where the camera is, how it is aimed, and its
    focal length.  The rotation angles are in degrees, and specify
    rotations around the X, Y, and Z axes.  The camera looks down
    its own Y axis, with the top of the picture in the direction of
    the Z axis.  If the rotation angles are all zero, its axes
    are aligned with the world coordinate axes.  The rotations are
    performed in the order ZXY about the camera axes.  A positive
    angle rotates Y toward Z, Z toward X, and X toward Y for
    rotations about the X, Y, and Z axes respectively.  To
    understand the focal length, imagine a 320 x 200 pixel
    rectangle perpendicular to, and centered on the camera's
    Y axis.  Any objects in the infinite rectangular cone defined
    by the camera position and the 4 corners of the rectangle will
```

```
    appear in the picture.
OTRK - size 18

    BYTE    Trackname[18];

    This chunk specifies the name of an object that the camera is
    "tracked" to.  If the name is NULL, the camera doesn't track
    Otherwise, if the object is moved inside Turbo Silver, the
    camera will follow it.

OSTR - size 56

    STORY   CStory;         ; a STORY structure for the camera

    The story structure is defined above.

FADE - size 12

    FRACT   FadeAt;         ; distance to start fade
    FRACT   FadeBy;         ; distance of total fade
    BYTE    pad;            ; pad byte - must be zero
    COLOR   FadeTo;         ; RGB color to fade to

SKYC - size 8

    BYTE    pad;            ; pad byte - must be zero
    COLOR   Horizon;        ; horizon color
    BYTE    pad;            ; pad byte - must be zero
    COLOR   Zenith;         ; zenith color

AMBI - size 4

    BYTE    pad;            ; pad byte - must be zero
    COLOR   Ambient;        ; abmient light color

GLB0 - size 8

    BYTE    Props[8];       ; an array of 8 "global properties" used
                            ; by Turbo Silver.

    Props[0] - GLB_EDGING       ; edge level (globals requester)
    Props[1] - GLB_PERTURB      ; perturbance (globals requester)
    Props[2] - GLB_SKY_BLEND    ; sky blending factor (0-255)
    Props[3] - GLB_LENS         ; lens type (see below)
    Props[4] - GLB_FADE         ; flag - Sharp/Fuzzy focus (globals)
    Props[5] - GLB_SIZE         ; "apparant size" (see below)
    Props[6] - GLB_RESOLVE      ; resolve depth (globals requester)
    Props[7] - GLB_EXTRA        ; flag - genlock sky on/off

    The edging and perturbance values control the heuristics in ray
    tracing.  The sky blending factor is zero for no blending, and 255
    for full blending.  The lens type is a number from 0-4, corresponding
    to the boxes in the "camera" requester, and correspond to 0) Manual,
    1) Wide angle, 2) Normal, 3) Telephoto, and 4) Custom.  It is used
    in setting the camera's focal length if the camera is tracked to an
    object.  The Sharp/Fuzzy flag turns the "fade" feature on and off -
    non-zero means on.  The "apparant size" parameter is 100 times the
```

```
    "custom size" parameter in the camera requester.  And is used to set
    the focal length for a custom lens.  The "resolve depth" controls
    the number of rays the ray tracer will shoot for a single pixel.
    Each reflective/refractive ray increments the depth counter, and
    the count is never allowed to reach the "resolve depth".  If both
    a reflective and a refractive ray are traced, each ray gets its
    own version of the count - so theoretically, a resolve depth of
    4 could allow much more than 4 rays to be traced.  The "genlock
    sky" flag controls whether the sky will be colored, or set to
    the genlock color (color 0 - black) in the final picture.


All of the INFO sub-chunks are optional, as is the INFO chunk.
Default values are supplied if the chunks are not present.  The
defaults are:  no brushes, stencils, or textures defined; no story
for the camera; horizon and zenith and ambient light colors set
to black; fade color set to (80,80,80);  un-rotated, un-tracked
camera at (-100, -100, 100); and global properties array set to
[30, 0, 0, 0, 0, 100, 8, 0].
```

1.163 Details / EXTR sub-sub-chunks

```
MTRX - size 60

    VECTOR  Translate;      ; translation vector
    VECTOR  Scale;          ; X,Y and Z scaling factors
    MATRIX  Rotate;         ; rotation matrix

    The translation vector is i world coordinates.
    The scaling factors are with respect to local axes.
    The rotation matrix is with respect to the world axes,
    and it should be a "unit matrix".
    The rotation is such that a rotated axis's X,Y, and Z
    components are the dot products of the MATRIX's I,J,
    and K vectors with the un-rotated axis vector.

LOAD - size 80

    BYTE    Filename[80];   ; the name of the external file

    This chunk contains the name of an external object file.
    The external file should be a FORM TDDD file.  It may contain
    an any number of objects possibly grouped into heirarchy(ies).

Both of these chunks are required.
```

1.164 A / IFF Third Party Public Form and Chunk Specification / WORD.doc

```
ProWrite document format (New Horizons)

TITLE:  WORD  (word processing FORM used by ProWrite)
```

```
IFF FORM / CHUNK DESCRIPTION

Form/Chunk IDs:
   FORM   WORD
   Chunks FONT,COLR,DOC,HEAD,FOOT,PCTS,PARA,TABS,PAGE,TEXT,FSCC,PINF

Date Submitted: 03/87
Submitted by:   James Bayless - New Horizons Software, Inc.

 FORM
 Chunks
```

1.165 WORD.doc / FORM

```
FORM ID:  WORD

FORM Purpose:  Document storage (supports color, fonts, pictures)

FORM Description:

This include file describes FORM WORD and its Chunks

   /*
    *      IFF Form WORD structures and defines
    *      Copyright (c) 1987 New Horizons Software, Inc.
    *
    *      Permission is hereby granted to use this file in any and all
    *      applications.  Modifying the structures or defines included
    *      in this file is not permitted without written consent of
    *      New Horizons Software, Inc.
    */

   #include ":IFF/ILBM.h"        /* Makes use of ILBM defines */

   #define ID_WORD      MakeID('W','O','R','D')      /* Form type */

   #define ID_FONT      MakeID('F','O','N','T')      /* Chunks */
   #define ID_COLR      MakeID('C','O','L','R')
   #define ID_DOC       MakeID('D','O','C',' ')
   #define ID_HEAD      MakeID('H','E','A','D')
   #define ID_FOOT      MakeID('F','O','O','T')
   #define ID_PCTS      MakeID('P','C','T','S')
   #define ID_PARA      MakeID('P','A','R','A')
   #define ID_TABS      MakeID('T','A','B','S')
   #define ID_PAGE      MakeID('P','A','G','E')
   #define ID_TEXT      MakeID('T','E','X','T')
   #define ID_FSCC      MakeID('F','S','C','C')
   #define ID_PINF      MakeID('P','I','N','F')

   /*
    *   Special text characters for page number, date, and time
    *   Note:  ProWrite currently supports only PAGENUM_CHAR, and only in
    *      headers and footers
    */
```

```
   #define PAGENUM_CHAR   0x80
   #define DATE_CHAR      0x81
   #define TIME_CHAR      0x82
```

1.166 WORD.doc / Chunks

```
/*
 *   FONT - Font name/number table
 *   There are one of these for each font/size combination
 *   These chunks should appear at the top of the file
 *   (before document data)
 */

   typedef struct {
      UBYTE   Num;          /* 0 .. 255 */
      UWORD   Size;
   /* UBYTE   Name[];      */   /* NULL terminated, without ".font" */
   } FontID;

/*
 *   COLR - Color translation table
 *   Translates from color numbers used in file to ISO color numbers
 *   Should be at top of file (before document data)
 *   Note:  Currently ProWrite only checks these values to be its
 *      current map, it does no translation as it does for FONT chunks
 */

   typedef struct {
      UBYTE   ISOColors[8];
   } ISOColors;

/*
 *   DOC - Begin document section
 *   All text and paragraph formatting following this chunk and up to a
 *   HEAD, FOOT, or PICT chunk belong to the document section
 */

   #define PAGESTYLE_1   0      /* 1, 2, 3 */
   #define PAGESTYLE_I   1      /* I, II, III */
   #define PAGESTYLE_i   2      /* i, ii, iii */
   #define PAGESTYLE_A   3      /* A, B, C */
   #define PAGESTYLE_a   4      /* a, b, c */

   typedef struct {
      UWORD   StartPage;      /* Starting page number */
      UBYTE   PageNumStyle;   /* From defines above */
      UBYTE   pad1;
      LONG    pad2;
   } DocHdr;

/*
 *   HEAD/FOOT - Begin header/footer section
 *   All text and paragraph formatting following this chunk and up to a
 *   DOC, HEAD, FOOT, or PICT chunk belong to this header/footer
 *   Note:  This format supports multiple headers and footers, but
```

```
 *      currently ProWrite only allows a single header and footer per
 *      document
 */

   #define PAGES_NONE   0
   #define PAGES_LEFT   1
   #define PAGES_RIGHT  2
   #define PAGES_BOTH   3

   typedef struct {
      UBYTE   PageType;       /* From defines above */
      UBYTE   FirstPage;      /* 0 = Not on first page */
      LONG   pad;
   } HeadHdr;

/*
 *   PCTS - Begin picture section
 *   Note:  ProWrite currently requires NPlanes to be three (3)
 */

   typedef struct {
      UBYTE   NPlanes;      /* Number of planes used in picture bitmaps */
      UBYTE   pad;
   } PictHdr;

/*
 *   PARA - New paragraph format
 *   This chunk should be inserted first when a new section is started
 *      (DOC, HEAD, or FOOT), and again whenever the paragraph format
 *      changes
 */

   #define SPACE_SINGLE   0
   #define SPACE_DOUBLE   0x10

   #define JUSTIFY_LEFT    0
   #define JUSTIFY_CENTER  1
   #define JUSTIFY_RIGHT   2
   #define JUSTIFY_FULL    3

   #define MISCSTYLE_NONE   0
   #define MISCSTYLE_SUPER  1      /* Superscript */
   #define MISCSTYLE_SUB    2      /* Subscript */

   typedef struct {
      UWORD   LeftIndent; /* In decipoints (720 dpi) */
      UWORD   LeftMargin;
      UWORD   RightMargin;
      UBYTE   Spacing;    /* From defines above */
      UBYTE   Justify;    /* From defines above */
      UBYTE   FontNum;    /* FontNum, Style, etc. for first char in para*/
      UBYTE   Style;      /* Standard Amiga style bits */
      UBYTE   MiscStyle;  /* From defines above */
      UBYTE   Color;      /* Internal number, use COLR to translate */
      LONG    pad;
   } ParaFormat;
```

```
/*
 *   TABS - New tab stop types/locations
 *   Use an array of values in each chunk
 *   Like the PARA chunk, this should be inserted whenever the tab
 *      settings for a paragraph change
 *   Note:  ProWrite currently does not support TAB_CENTER
 */

   #define TAB_LEFT     0
   #define TAB_CENTER   1
   #define TAB_RIGHT    2
   #define TAB_DECIMAL  3

   typedef struct {
      UWORD   Position;      /* In decipoints */
      UBYTE   Type;
      UBYTE   pad;
   } TabStop;

/*
 *   PAGE - Page break
 *   Just a marker - this chunk has no data
 */

/*
 *   TEXT - Paragraph text (one block per paragraph)
 *   Block is actual text, no need for separate structure
 *   If the paragraph is empty, this is an empty chunk -- there MUST be
 *   a TEXT block for every paragraph
 *   Note:  The only ctrl characters ProWrite can currently handle in TEXT
 *   chunks are Tab and PAGENUM_CHAR, ie no Return's, etc.
 */

/*
 *   FSCC - Font/Style/Color changes in previous TEXT block
 *   Use an array of values in each chunk
 *   Only include this chunk if the previous TEXT block did not have
 *      the same Font/Style/Color for all its characters
 */

   typedef struct {
      UWORD   Location;  /* Character location in TEXT chunk of change */
      UBYTE   FontNum;
      UBYTE   Style;
      UBYTE   MiscStyle;
      UBYTE   Color;
      UWORD   pad;
   } FSCChange;

/*
 *   PINF - Picture info
 *   This chunk must only be in a PCTS section
 *   Must be followed by ILBM BODY chunk
 *   Pictures are treated independently of the document text (like a
 *      page-layout system), this chunk includes information about what
 *      page and location on the page the picture is at
 *   Note:  ProWrite currently only supports mskTransparentColor and
```

```
 *      mskHasMask masking
 */

   typedef struct {
      UWORD         Width, Height;   /* In pixels */
      UWORD         Page;         /* Which page picture is on (0..max) */
      UWORD         XPos, YPos;      /* Location on page in decipoints */
      Masking       Masking;      /* Like ILBM format */
      Compression   Compression;   /* Like ILBM format */
      UBYTE         TransparentColor;   /* Like ILBM format */
      UBYTE         pad;
   } PictInfo;

/* end */
```