Exhibit 20

"8SVX" IFF 8-Bit Sampled Voice
Date: February 7, 1985
From: Steve Hayes and Jerry M ison, Electronic Arts
Status: Adopted

1. Introduction

This memo is the IFF supplement for FORM "8SVX". An 8SVX is an IFF "data section" or "FORM" (which can be an IFF file or a part of one) containing a digitally sampled audio voice consisting of 8-bit samples. A voice can be a one-shot sound orQwith repetition and pitch scalingQa musical instrument. "EA IFF 85" is Electronic Arts' standard interchange file format. [See "EA IFF 85" Standard for Interchange Format Files.]

The 8SVX format is designed for playback hardware that uses 8-bit samples attenuated by a volume control for good overall signal-to-noise ratio. So a FORM 8SVX stores 8-bit samples and a volume level.

A similar data format (or two) will be needed for higher resolution samples (typically 12 or 16 bits). Properly converting a high resolution sample down to 8 bits requires one pass over the data to find the minimum and maximum values and a second pass to scale each sample into the range -128 through 127. So it's reasonable to store higher resolution data in a different FORM type and convert between them.

For instruments, FORM 8SVX can record a repeating waveform optionally preceded by a startup transient waveform. These two recorded signals can be pre-synthesized or sampled from an acoustic instrument. For many instruments, this representation is compact. FORM 8SVX is less practical for an instrument whose waveform changes from cycle to cycle like a plucked string, where a long sample is needed for accurate results.

FORM 8SVX can store an "envelope" or "amplitude contour" to enrichen musical notes. A future voice FORM could also store amplitude, frequency, and filter modulations.

FORM 8SVX is geared for relatively simple musical voices, where one waveform per octave is sufficient, where the waveforms for the different octaves follows a factor-of-two size rule, and where one envelope is adequate for all octaves. You could store a more general voice as a LIST containing one or more FORMs 8SVX per octave. A future voice FORM could go beyond one "one-shot" waveform and one "repeat" waveform per octave.

Section 2 defines the required property sound header "VHDR", optional properties name "NAME", copyright "(c)J", and author "AUTH", the optional annotation data chunk "ANNO", the required data chunk "BODY", and optional envelope chunks "ATAK" and "RLSE". These are the "standard" chunks. Specialized chunks for private or future needs can be added later, e.g. to hold a frequency contour or Fourier series coefficients. The 8SVX syntax is summarized in Appendix A as a regular expression and in Appendix B as an example box diagram. Appendix C explains the optional Fibonacci-delta compression algorithm.

Caution: The VHDR structure Voice8Header changed since draft proposal #4! The new structure is incompatible with the draft version.

Reference:

"EA IFF 85" Standard for Interchange Format Files describes the underlying conventions for all IFF files.

Amiga[tm] is a trademark of Commodore-Amiga, Inc.

Electronic Arts[tm] is a trademark of Electronic Arts.

MacWrite[tm] is a trademark of Apple Computer, Inc.

2. Standard Data and Property Chunks

FORM 8SVX stores all the waveform data in one body chunk "BODY". It stores playback parameters in the required header chunk "VHDR". "VHDR" and any optional property chunks "NAME", "(c)J", and "AUTH" must all appear before the BODY chunk. Any of these properties may be shared over a LIST of FORMs 8SVX by putting them in a PROP 8SVX. [See "EA IFF 85" Standard for Interchange Format Files.]

Background

There are two ways to use FORM 8SVX: as a one-shot sampled sound or as a sampled musical instrument that plays "notes". Storing both kinds of sounds in the same kind of FORM makes it easy to play a one-shot sound as a (staccato) instrument or an instrument as a (one-note) sound.

A one-shot sound is a series of audio data samples with a nominal playback rate and amplitude. The recipient program can optionally adjust or modulate the amplitude and playback data rate.

For musical instruments, the idea is to store a sampled (or pre-synthesized) waveform that will be parameterized by pitch, duration, and amplitude to play each "note". The creator of the FORM 8SVX can supply a waveform per octave over a range of octaves for this purpose. The intent is to perform a pitch by selecting the closest octave's waveform and scaling the playback data rate. An optional "one-shot" waveform supplies an arbitrary startup transient, then a "repeat" waveform is iterated as long as necessary to sustain the note.

A FORM 8SVX can also store an envelope to modulate the waveform. Envelopes are mostly useful for variable-duration notes but could be used for one-shot sounds, too.

The FORM 8SVX standard has some restrictions. For example, each octave of data must be twice as long as the next higher octave. Most sound driver software and hardware imposes additional restrictions. E.g. the Amiga sound hardware requires an even number of samples in each one-shot and repeat waveform.

Required Property VHDR

The required property "VHDR" holds a Voice8Header structure as defined in these C declarations and following documentation. This structure holds the playback parameters for the sampled waveforms in the BODY chunk. (See "Data Chunk BODY", below, for the storage layout of these waveforms.)

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;
        /* A fixed-point value, 16 bits to the left of the point and 16
         * to the right. A Fixed is a number of 216ths, i.e. 65536ths.  */

#define Unity 0x10000L  /* Unity = Fixed 1.0 = maximum volume    */

/* sCompression: Choice of compression algorithm applied to the samples. */
```

```
#define sCmpNone        0       /* not compressed       */
#define sCmpFibDelta    1           ibonacci-delta encoding (Appe   x C)  */

/* Can be more kinds in the future.     */

typedef struct {
        ULONG oneShotHiSamples, /* # samples in the high octave 1-shot part */
        repeatHiSamples,        /* # samples in the high octave repeat part */
        samplesPerHiCycle;      /* # samples/cycle in high octave, else 0   */
        UWORD samplesPerSec;    /* data sampling rate   */
        UBYTE ctOctave,         /* # octaves of waveforms       */
        sCompression;           /* data compression technique used      */
        Fixed volume;           /* playback volume from 0 to Unity (full
                                 * volume). Map this value into the output
                                 * hardware's dynamic range.    */
        } Voice8Header;
```

[Implementation details. Fields are filed in the order shown. The UBYTE fields are byte-packed (2 per 16-bit word). MakeID is a C macro defined in the main IFF document and in the source file IFF.h.]

A FORM 8SVX holds waveform data for one or more octaves, each containing a one-shot part and a repeat part. The fields oneShotHiSamples and repeatHiSamples tell the number of audio samples in the two parts of the highest frequency octave. Each successive (lower frequency) octave contains twice as many data samples in both its one-shot and repeat parts. One of these two parts can be empty across all octaves.

Note: Most audio output hardware and software has limitations. The Amiga computer's sound hardware requires that all one-shot and repeat parts have even numbers of samples. Amiga sound driver software would have to adjust an odd-sized waveform, ignore an odd-sized lowest octave, or ignore odd FORMs 8SVX altogether. Some other output devices require all sample sizes to be powers of two.

The field samplesPerHiCycle tells the number of samples/cycle in the highest frequency octave of data, or else 0 for "unknown". Each successive (lower frequency) octave contains twice as many samples/cycle. The samplesPerHiCycle value is needed to compute the data rate for a desired playback pitch.

Actually, samplesPerHiCycle is an average number of samples/cycle. If the one-shot part contains pitch bends, store the samples/cycle of the repeat part in samplesPerHiCycle. The division repeatHiSamples/samplesPerHiCycle should yield an integer number of cycles. (When the repeat waveform is repeated, a partial cycle would come out as a higher-frequency cycle with a "click".)

More limitations: Some Amiga music drivers require samplesPerHiCycle to be a power of two in order to play the FORM 8SVX as a musical instrument in tune. They may even assume samplesPerHiCycle is a particular power of two without checking. (If samplesPerHiCycle is different by a factor of two, the instrument will just be played an octave too low or high.)

The field samplesPerSec gives the sound sampling rate. A program may adjust this to achieve frequency shifts or vary it dynamically to achieve pitch bends and vibrato. A program that plays a FORM 8SVX as a musical instrument would ignore samplesPerSec and select a playback rate for each musical pitch.

The field ctOctave tells how many octaves of data are stored in the BODY chunk. See "Data Chunk BODY", below, for the layout of the octaves.

The field sCompression indicates the compression scheme, if any, that was applied to the entire set of data samples stored in the BODY chunk. This field should contain one of the values defined above. Of course,

the matching decompression algori n must be applied to the BODY data before the sound can be played. e Fibonacci-delta encoding sch sCmpFibDelta is described in Ap dix C.) Note that the whole se: s of data samples is compressed as a unit.

The field volume gives an overall playback volume for the waveforms (all octaves). It lets the 8-bit data samples use the full range -128 through 127 for good signal-to-noise ratio and be attenuated on playback to the desired level. The playback program should multiply this value by a "volume control" and perhaps by a playback envelope (see ATAK and RLSE, below).

Recording a one-shot sound. To store a one-shot sound in a FORM 8SVX, set oneShotHiSamples = number of samples, repeatHiSamples = 0 , samplesPerHiCycle = 0, samplesPerSec = sampling rate, and ctOctave = 1. Scale the signal amplitude to the full sampling range -128 through 127. Set volume so the sound will playback at the desired volume level. If you set the samplesPerHiCycle field properly, the data can also be used as a musical instrument.

Experiment with data compression. If the decompressed signal sounds ok, store the compressed data in the BODY chunk and set sCompression to the compression code number.

Recording a musical instrument. To store a musical instrument in a FORM 8SVX, first record or synthesize as many octaves of data as you want to make available for playback. Set ctOctaves to the count of octaves. From the recorded data, excerpt an integral number of steady state cycles for the repeat part and set repeatHiSamples and samplesPerHiCycle. Either excerpt a startup transient waveform and set oneShotHiSamples, or else set oneShotHiSamples to 0. Remember, the one-shot and repeat parts of each octave must be twice as long as those of the next higher octave. Scale the signal amplitude to the full sampling range and set volume to adjust the instrument playback volume. If you set the samplesPerSec field properly, the data can also be used as a one-shot sound.

A distortion-introducing compressor like sCmpFibDelta is not recommended for musical instruments, but you might try it anway.

Typically, creators of FORM 8SVX record an acoustic instrument at just one frequency. Decimate (down- sample with filtering) to compute higher octaves. Interpolate to compute lower octaves.

If you sample an acoustic instrument at different octaves, you may find it hard to make the one-shot and repeat waveforms follow the factor-of-two rule for octaves. To compensate, lengthen an octave's one-shot part by appending replications of the repeating cycle or prepending zeros. (This will have minimal impact on the sound's start time.) You may be able to equalize the ratio one-shot-samples : repeat-samples across all octaves.

Note that a "one-shot sound" may be played as a "musical instrument" and vice versa. However, an instrument player depends on samplesPerHiCycle, and a one-shot player depends on samplesPerSec.

Playing a one-shot sound. To play any FORM 8SVX data as a one-shot sound, first select an octave if ctOctave > 1. (The lowest-frequency octave has the greatest resolution.) Play the one-shot samples then the repeat samples, scaled by volume, at a data rate of samplesPerSec. Of course, you may adjust the playback rate and volume. You can play out an envelope, too. (See ATAK and RLSE, below.)

Playing a musical note. To play a musical note using any FORM 8SVX, first select the nearest octave of data from those available. Play the one-shot waveform then cycle on the repeat waveform as long as

needed to sustain the note. Scale the signal by volume, perhaps also by an envelope, and by a desired voice volume. Select a playback d rate s samples/second to achieve the desired frequency (in Hz):
frequency = sJ/JsamplesPerHiCycle
for the highest frequency octave.

The idea is to select an octave and one of 12 sampling rates (assuming a 12-tone scale). If the FORM 8SVX doesn't have the right octave, you can decimate or interpolate from the available data.

When it comes to musical instruments, FORM 8SVX is geared for a simple sound driver. Such a driver uses a single table of 12 data rates to reach all notes in all octaves. That's why 8SVX requires each octave of data to have twice as many samples as the next higher octave. If you restrict samplesPerHiCycle to a power of two, you can use a predetermined table of data rates.

Optional Text Chunks NAME, (c), AUTH, ANNO

Several text chunks may be included in a FORM 8SVX to keep ancillary information.

The optional property "NAME" names the voice, for instance "tubular bells".

The optional property "(c)J" holds a copyright notice for the voice. The chunk ID "(c)J" serves as the copyright characters ")J". E.g. a "(c)J" chunk containing "1986 Electronic Arts" means ") 1986 Electronic Arts".

The optional property "AUTH" holds the name of the instrument's "author" or "creator".

The chunk types "NAME", "(c) ", and "AUTH" are property chunks. Putting more than one NAME (or other) property in a FORM is redundant. Just the last NAME counts. A property should be shorter than 256 characters. Properties can appear in a PROP 8SVX to share them over a LIST of FORMs 8SVX.

The optional data chunk "ANNO" holds any text annotations typed in by the author.

An ANNO chunk is not a property chunk, so you can put more than one in a FORM 8SVX. You can make ANNO chunks any length up to 231 - 1 characters, but 32767 is a practical limit. Since they're not properties, ANNO chunks don't belong in a PROP 8SVX. That means they can't be shared over a LIST of FORMs 8SVX.

Syntactically, each of these chunks contains an array of 8-bit ASCII characters in the range " " (SP, hex 20) through "~" (tilde, hex 7F), just like a standard "TEXT" chunk. [See "Strings, String Chunks, and String Properties" in "EA IFF 85" Electronic Arts Interchange File Format.] The chunk's ckSize field holds the count of characters.

```
#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.      */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.     */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.     */
```

Remember to store a 0 pad byte after any odd-length chunk.

Optional Data Chunks ATAK and RLSE

The optional data chunks ATAK and RLSE together give a piecewise-linear "envelope" or "amplitude contour". This contour may be used to modulate the sound during playback. It's especially useful for playing musical notes of variable durations. Playback programs may ignore the supplied envelope or substitute another.

```
#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')

typedef struct {
        UWORD duration; /* segment duration in milliseconds, > 0        */
        Fixed dest;     /* destination volume factor    */
        } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope.*/
/* The envelope defines a function of time returning Fixed values. It's
 * used to scale the nominal volume specified in the Voice8Header. */
```

To explain the meaning of the ATAK and RLSE chunks, we'll overview the envelope generation algorithm. Start at 0 volume, step through the ATAK contour, then hold at the sustain level (the last ATAK EGPoint's dest), and then step through the RLSE contour. Begin the release at the desired note stop time minus the total duration of the release contour (the sum of the RLSE EGPoints' durations). The attack contour should be cut short if the note is shorter than the release contour.

The envelope is a piecewise-linear function. The envelope generator interpolates between the EGPoints.

Remember to multiply the envelope function by the nominal voice header volume and by any desired note volume.

Figure 1 shows an example envelope. The attack period is described by 4 EGPoints in an ATAK chunk. The release period is described by 4 EGPoints in a RLSE chunk. The sustain period in the middle just holds the final ATAK level until it's time for the release.

Figure 1. Amplitude contour.

Note: The number of EGPoints in an ATAK or RLSE chunk is its ckSize / sizeof(EGPoint). In RAM, the playback program may terminate the array with a 0 duration EGPoint.

Issue: Synthesizers also provide frequency contour (pitch bend), filtering contour (wah-wah), amplitude oscillation (tremolo), frequency oscillation (vibrato), and filtering oscillation (leslie). In the future, we may define optional chunks to encode these modulations. The contours can be encoded in linear segments. The oscillations can be stored as segments with rate and depth parameters.

Data Chunk BODY

The BODY chunk contains the audio data samples.

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

typedef character BYTE; /* 8 bit signed number, -128 through 127.      */

/* BODY chunk contains a BYTE[], array of audio data samples.   */
```

The BODY contains data samples grouped by octave. Within each octave are one-shot and repeat portions. Figure 2 depicts this arrangement of samples for an 8SVX where oneShotHiSamples = 24, repeatHiSamples = 16, samplesPerHiCycle = 8, and ctOctave = 3. The major divisions are octaves, the intermediate divisions separate the one-shot and repeat portions, and the minor divisions are cycles.

□

Figure 2. BODY subdivisions.

In general, the BODY has ctOctave octaves of data. The highest frequency octave comes first, comprising the fewest samples: oneShotHiSamples + repeatHiSamples. Each successive octave contains twice as many samples as the next higher octave but the same number of cycles. The lowest frequency octave comes last with the most samples: 2ctOctave-1 * (oneShotHiSamples + repeatHiSamples).

The number of samples in the BODY chunk is

$$(2^{0} + \ldots + 2^{(ctOctave-1)}) * (oneShotHiSamples + repeatHiSamples)$$

Figure 3, below, looks closer at an example waveform within one octave of a different BODY chunk. In this example, oneShotHiSamples / samplesPerHiCycle = 2 cycles and repeatHiSamples / samplesPerHiCycle = 1 cycle.

□

Figure 3. Example waveform.

To avoid playback "clicks", the one-shot part should begin with a small sample value, and the one-shot part should flow smoothly into the repeat part, and the end of the repeat part should flow smoothly into the beginning of the repeat part.

If the VHDR field sCompression - sCmpNone, the BODY chunk is just an array of data bytes to feed through the specified decompresser function. All this stuff about sample sizes, octaves, and repeat parts applies to the decompressed data.

Be sure to follow an odd-length BODY chunk with a 0 pad byte.

Other Chunks

Issue: In the future, we may define an optional chunk containing Fourier series coefficients for a repeating waveform. An editor for this kind

of synthesized voice could modify he coefficients and regenerate the waveform.

□

Appendix A. Quick Reference

Type Definitions

```
#define ID_8SVX MakeID('8', 'S', 'V', 'X')
#define ID_VHDR MakeID('V', 'H', 'D', 'R')

typedef LONG Fixed;     /* A fixed-point value, 16 bits to the left
                         * of the point and 16 to the right. A Fixed
                         * is a number of 216ths, i.e. 65536ths. */

#define Unity 0x10000L  /* Unity = Fixed 1.0 = maximum volume    */

/* sCompression: Choice of compression algorithm applied to the samples */

#define sCmpNone       0        /* not compressed       */
#define sCmpFibDelta   1        /* Fibonacci-delta encoding (Appendix C) */
      /* Can be more kinds in the future.     */

typedef struct {
        ULONG oneShotHiSamples, /* # samples in the high octave 1-shot part */
        repeatHiSamples,        /* # samples in the high octave repeat part */
        samplesPerHiCycle;      /* # samples/cycle in high octave, else 0   */
        UWORD samplesPerSec;    /* data sampling rate   */
        UBYTE ctOctave,         /* # octaves of waveforms       */
        sCompression;           /* data compression technique used      */
        Fixed volume;           /* playback volume from 0 to Unity (full
                                 * volume). Map this value into the output
                                 * hardware's dynamic range.    */
        } Voice8Header;

#define ID_NAME MakeID('N', 'A', 'M', 'E')
/* NAME chunk contains a CHAR[], the voice's name.      */

#define ID_Copyright MakeID('(', 'c', ')', ' ')
/* "(c) " chunk contains a CHAR[], the FORM's copyright notice. */

#define ID_AUTH MakeID('A', 'U', 'T', 'H')
/* AUTH chunk contains a CHAR[], the author's name.     */

#define ID_ANNO MakeID('A', 'N', 'N', 'O')
/* ANNO chunk contains a CHAR[], author's text annotations.     */

#define ID_ATAK MakeID('A', 'T', 'A', 'K')
#define ID_RLSE MakeID('R', 'L', 'S', 'E')

typedef struct {
        UWORD duration; /* segment duration in milliseconds, > 0        */
        Fixed dest;     /* destination volume factor    */
        } EGPoint;

/* ATAK and RLSE chunks contain an EGPoint[], piecewise-linear envelope. */
/* The envelope defines a function of time returning Fixed values. It's
 * used to scale the nominal volume specified in the Voice8Header.  */

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

typedef character BYTE; /* 8 bit signed number, -128 through 127.       */

/* BODY chunk contains a BYTE[], array of audio data samples.   */
```

8SVX Regular Expression

Here's a regular expression summary of the FORM 8SVX syntax. This could be an IFF file or part of one.

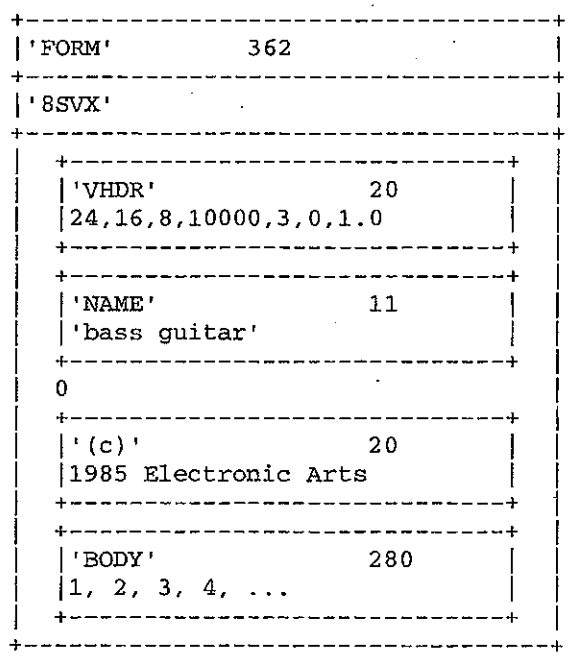
```
8SVX     ::= "FORM" #{   "8SVX" VHDR [NAME] [Copyright] [AUTH] ANNO*
                [ATAK] [RLSE] BODY }

VHDR     ::= "VHDR" #{ Voice8Header      }
NAME     ::= "NAME" #{ CHAR*     } [0]
Copyright       ::= "(c) " #{ CHAR*     } [0]
AUTH     ::= "AUTH" #{ CHAR*     } [0]
ANNO     ::= "ANNO" #{ CHAR*     } [0]

ATAK     ::= "ATAK" #{ EGPoint*  }
RLSE     ::= "RLSE" #{ EGPoint*  }
BODY     ::= "FORM" #{ BYTE*     } [0]
```

The token "#" represents a ckSize LONG count of the following {braced} data bytes. E.g. a VHDR's "#" should equal sizeof(Voice8Header). Literal items are shown in "quotes", [square bracket items] are optional, and "*" means 0 or more replications. A sometimes-needed pad byte is shown as "[0]".

Actually, the order of chunks in a FORM 8SVX is not as strict as this regular expression indicates. The property chunks VHDR, NAME, Copyright, and AUTH may actually appear in any order as long as they all precede the BODY chunk. The optional data chunks ANNO, ATAK, and RLSE don't have to precede the BODY chunk. And of course, new kinds of chunks may appear inside a FORM 8SVX in the future.

Appendix B. 8SVX Example

Here's a box diagram for a simple example containing the three octave BODY shown earlier in Figure 2.



The "0" after the NAME chunk is a pad byte.

Appendix B. Standards Committee

The following people contributed the design of this IFF standa

Bob "Kodiak" Burns, Commodore-Amiga
R. J. Mical, Commodore-Amiga
Jerry Morrison, Electronic Arts
Greg Riker, Electronic Arts
Steve Shaw, Electronic Arts
Barry Walsh, Commodore-Amiga

The "0" after the NAME chunk is a pad byte.

□

Appendix C. Fibonacci Delta Compression

This is Steve Hayes' Fibonacci Delta sound compression technique. It's like the traditional delta encoding but encodes each delta in a mere 4 bits. The compressed data is half the size of the original data plus a 2-byte overhead for the initial value. This much compression introduces some distortion, so try it out and use it with discretion.

To achieve a reasonable slew rate, this algorithm looks up each stored 4-bit value in a table of Fibonacci numbers. So very small deltas are encoded precisely while larger deltas are approximated. When it has to make approximations, the compressor should adjust all the values (forwards and backwards in time) for minimum overall distortion.

Here is the decompressor written in the C programming language.

```
/* Fibonacci delta encoding for sound data. */

BYTE codeToDelta[16] = {-34,-21,-13,-8,-5,-3,-2,-1,0,1,2,3,5,8,13,21};

/* Unpack Fibonacci-delta encoded data from n byte source buffer into 2*n byte
 * dest buffer, given initial data value x. It returns the last data value x
 * so you can call it several times to incrementally decompress the data. */
short D1Unpack(source, n, dest, x)
        BYTE source[], dest[];
        LONG n;
        BYTE x;
        {
        BYTE d;
        LONG i, lim;

        lim = n <<<< 1;
        for (i = 0; i << lim; ++i)
                {
                /* Decode a data nybble; high nybble then low nybble. */
                d = source[i >> 1];     /* get a pair of nybbles */
                if (i & 1)              /* select low or high nybble? */
                        d &= 0xf;       /* mask to get the low nybble */
                else
                        d >>= 4;        /* shift to get the high nybble */
                x += codeToDelta[d];    /* add in the decoded delta */
                dest[i] = x;            /* store a 1-byte sample */
                }
        return(x);
        }

/* Unpack Fibonacci-delta encoded data from n byte source buffer into 2*(n-2)
 * byte dest buffer. Source buffer has a pad byte, an 8-bit initial value,
 * followed by n-2 bytes comprising 2*(n-2) 4-bit encoded samples. */
```

```
void DUnpack(source, n, dest)
        BYTE source[], dest[];
        LONG n;
        {
                D1Unpack(source + 2, n - 2, dest, source[1]);
        }
```