VOX AUDIO FILE CONVERSION

The programs *vox* and *devox* are C programs developed in 1997 when I worked at Sprint to convert audio files between OKI ADPCM (Dialogic vox) file format and linear audio files, which work with PC audio hardware. Conversions to/from other formats can be accomplished with sox.

No current development work is going on with the *vox* program. It is believed to do its intended job, and I am longer work in the voice processing industry.

Vox works on all known versions of Unix, including Linux.

THE STANDARD

I originally wrote the program based on a description of the algorithm found in the book *PC Telephony - The complete guide to designing, building and programming systems using Dialogic and Related Hardware* by Bob Edgar, pg 272-276, third edition, 1995, Flatiron Publishing, Inc., New York. Relavent pages are shown here (scanned). I think the book is out of print, so hopefully listing a few pages from the book is okay.

Dialogic Standard ADPCM

The digitization method originally used by Dialogic is a 4-bit ADPCM variant at a rate of 6053 samples/second . The "4-bit" designation means that each sample is represented by a 4-bit value. Older Dialogic literature sometimes referred to ADPCM files as *VOX* files (vox is the Latin for "voice"), and used the DOS file extension .VOX for files stored in this format. Dialogic derived their standard from the pre-existing *Oki* ADPCM standard, so you may sometimes see this format referred to as Oki or Oki ADPCM.

The first of the four bits is the *sign*, in other words whether the current sample is greater or less than the previous sample. If the current sample is greater, the sign is zero; if the current sample is less, the sign is one. The remaining three bits represent a value between zero and seven which represent the approximate magnitude of the difference between the samples. The representation of the magnitude is *non-linear*, which means that doubling the value in

(Edgar, P272)

the magnitude does not necessarily double the amplitude of the encoded sound.

The ADPCM encoding algorithm works with three input values: two signed twelve-bit amplitude samples Sn, the current sample and Sn-1, the previous sample; and the current step size SS. The procedure is as follows:

1. Set these values to zero: B0, B1, B2, B3

2. Calculate the difference, Dn = Sn - Sn-1. If Dn is less than zero, set B3 to 1.

3. Set E to be the absolute value of Dn, ie. if Dn is greater than zero, set E to Dn; if Dn is less than zero, set E to -Dn.

4. If E >= SS, set B2 to 1 and subtract SS from E.

5. If E >= SS/2, set B1 to 1, and subtract SS/2 from E.

6. If E >= SS/4, set B0 to 1.

After following these steps, the 4-bit ADPCM sample is the four bits:

B3 B2 B1 B0

The step size is re-calculated each time using the previous step size and the current 3-bit ADPCM magnitude Mn.

The step size SS can take one of 49 different values from the following table:

(Edgar, P273)

Nr	*SS*	*Nr*	*SS*	*Nr*	*SS*	*Nr*	*SS*
1	16	13	50	25	157	37	494
2	17	14	55	26	173	38	544
3	19	15	60	27	190	39	598
4	21	16	66	28	209	40	658
5	23	17	73	29	230	41	724
6	25	18	80	30	253	42	796
7	28	19	88	31	279	43	876
8	31	20	97	32	307	44	963
9	34	21	107	33	337	45	1060
10	37	22	118	34	371	46	1166
11	41	23	130	35	408	47	1282
12	45	24	143	36	449	48	1408
						49	1552

Note that a signed twelve bit sample ranges in value from -2048 to 2047, so the maximum step size of 1552 can take a sample from minimum to maximum in three steps. To calculate the step size, use the current 3-bit magnitude Mn from the current ADPCM sample and find X from the following table:

Mn	*X*
000	-1
001	-1
010	-1
011	-1
100	2
101	4
110	6
111	8

Use the value to adjust the current position of the step size in the table. For example, if the current step size is number 24, with step size 143, and Mn is 100, giving a change of 2, the new step size

(Edgar, P274)

will be number 26, ie. 173. If this would change the step size to less than number 16, use 16; if this would change the step size to greater than number 49, use 49.

This procedure may seem strange, but it was developed by extensive analysis of speech: the algorithm is effective for storing the human voice.

To decode an ADPCM value, the above procedure is reversed. The step size is adjusted from sample to sample in exactly the same way as for encoding. To decode, start by calculating the amplitude of the difference Mn:

Mn = B2*SS + B1*(SS/2) + B0*(SS/4) + SS/8

If B3 is 0, set Dn = Mn

If B3 is 1, set Dn = -Mn

Then the new output 12-bit linear sample Sn is calculated from the previous:

Sn = Sn-1 + Dn

To initialize, the sample before the first twelve-bit sample is considered to be zero (at the middle of the scale), and the step size is set to the minimum value of 16 (number 1 in the table).

An important feature, and sometimes drawback, of ADPCM encoding methods is that they are *context-dependent* — in other words, the interpretation of a given set of samples depends on the samples which precedes that set. This means that you cannot simply "cut and paste" fragments of an ADPCM file without adjusting samples to accommodate the current context.

(Edgar, P275)

The 4-bit ADPCM algorithm can be "reset" to its initial state by a sequence of 48 samples of plus and minus zero (0000, 1000) in either order.

(Edgar, P276)

Since developing the program, I found that Dialogic published a standard for the ADPCM algorithm on it's web page. I don't know if they still post the standard.

Disclaimer: (dated 6/26/2003) A few people have pointed out to me that there is a minor difference between my program and the Dialogic standard. The difference relates to how the step size is calculated. The difference is also reflected in Edgar's text, which my code is based on. I don't how or why the difference arouse. I have a few comments about said difference:

1. I wrote the program to a published algorithm available to me at the time.
2. I might be able work on a new version of the program which will match the Dialogic standard; however, since I am not in the voice processing industry, I don't have any hardware to use to test the new code. If you have some hardware and are willing to help with testing, let me know. I don't get the impression that there is much demand for a new version of it.
3. The differences seem to be fairly minor. If you just want to be able to do simple conversions for listening purposes, my code, as it is now, will work for you. If you require exact conversion, then my code may not be what you want.

READ ME DOCUMENTATION

```
vox and devox:
This package is for conversions between Oki ADPCM and linear voice
files. (see the man page)

The makefile is for gcc.  If you have another compiler, change the
makefile.  Also note the INSTALLDIR if you want make to also install
the binaries and man pages.

To just build:
    make all
To build and install:
    make install

Other recommended utilities:

sox (general purpose voice file format conversion)
Generic source:ftp.cwi.nl, in directory /pub/audio/sox<version>.tar.Z.
You may be able to locate a nearer version.
Linux users look in: sunsite.unc.edu /pub/Linux/apps/sound/convert/

mxv (waveform viewer/editor) ftp.ccmrc.ucsb.edu/pub/MixViews
```

MAKEFILE CODE

```
OBJ=vox.o adpcm.o
DOBJ=devox.o adpcm.o
CC=gcc
CFLAGS=
INSTALLDIR=/usr/local

all: devox vox


install: all
      cp vox $(INSTALLDIR)/bin
      cp devox $(INSTALLDIR)/bin

      cp vox.1 $(INSTALLDIR)/man/man1
```

```
	ln -s $(INSTALLDIR)/man/man1/vox.1 $(INSTALLDIR)/man/man1/devox.1

devox: ${DOBJ}
	$(CC) -o $@ ${DOBJ}

vox: ${OBJ}
	$(CC) -o $@ ${OBJ}

vox.o:vox.c adpcm.h
	$(CC) $(CFLAGS) -c vox.c

devox.o:devox.c adpcm.h
	$(CC) $(CFLAGS) -c devox.c

adpcm.o:adpcm.c adpcm.h
	$(CC) $(CFLAGS) -c adpcm.c
```

VOX.1 CODE

```
.de Sh
.br
.ne 5
.PP
\fB\\$1\fR
.PP
..
.de Sp
.if t .sp .5v
.if n .sp
..
.TH VOX 1
.SH NAME
vox, devox - programs to convert voice files between linear and Oki
(Dialogic)
ADPCM format.
.SH SYNOPSIS
.B vox [-b 8 | -b 16] \fIinfile outfile \fB
.br
```

.B devox [-b 8 | -b 16] \fIinfile outfile \fB
.br
.SH DESCRIPTION
.I vox
translates sound files from linear (8 or 16 bit) to Oki ADPCM format.
.I devox
translates sound files from Oki ADPCM to linear (8 or 16 bit) format.
The Oki ADPCM format is commonly found on platforms using voice
processing hardware from Dialogic for computer telephony applications.
The default Dialogic file is
titled with a '.vox' suffix and is sampled at 6022 samples per
second -- thus considered a 24 Kbit/sec coder. Sampling at 8000
samples per second is also popular in the computer telephony world.
.SH OPTIONS
The option syntax is pretty simple
.br
      vox file.8bit file.32K
.br
translates a sound sample in 8 bit linear file
into Oki ADPCM format, while
.br
      devox file.32K file.8bit
.br
does the reverse.
.PP
Linear File options:
.TP 10
.B -b 8
(Default) The linear file is in 8 bit (unsigned byte) format.
The option is not needed since it is the default.
.TP 10
.B -b 16
The linear file is in 16 bit (signed word) format.
.SH FILE TYPES
.I vox
and
.I devox
only support raw (no header) binary files. The
ADPCM files are compatible with Dialogic's so-called vox files.
They contain two 4 bit samples stored in one unsigned char.

```
The sampling rate does not matter to these programs (may be
6 or 8 kHz).  The linear files are either in 8 bit linear (not mu-law)
or 16 bit linear format.
.SH BUGS
Only supports raw files.  Use the sox program to convert the linear
files to/from other file formats.
.SH AUTHOR
      Tim Bower, tim@cis.ksu.edu
.SH NOTICES
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted.
This software is provided "as is" without express or implied warranty.
```

VOX.C CODE

```
/*  Filename: vox.c
    Description:  Kind of like the sox program.  It converts voice
          file formats.  Converts 16 bit and 8 bit raw voice files to
          the Dialogic or Oki ADPCM (foo.vox or foo.32K) file format.
          Of course, the files have to sampled at the right amount for
          them to work with Dialogic hardware.
          Files sampled at 8-kHz are converted to the 32K.
          Files sampled at 6053 Hz are converted to the 24K -- normal
          vox format.

    Usage: vox [-b 8|-b 16] infile outfile

        The -b is for 8 or 16 bit files (reference to input files)
        Default is 8 bits.
*/

#include <unistd.h>           /* needed for getopt */
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "adpcm.h"
```

```
int read12(int, int, short *, int );    /* program to read and convert
                                        * data to 12 bit format.
                                        */

int main (int argc, char **argv)
{
    int c, i, j;
    extern char *optarg;
    extern int optind;
    int infile, outfile, n;
    int buffer_size, sample_size;
    short *buffer12;
    char *adpcm;
    struct adpcm_status coder_stat;

    /*
     * Process the arguments.
     */
    sample_size = 1;   /* default to 8 bit */
    while ((c = getopt(argc, argv, "b:")) != -1) {
        switch (c) {
        case 'b':
            /*
            * set bits per sample to 8 or 16 - sample_size to
            * 1 or 2 bytes.
            */
            switch (atoi(optarg)) {
                case 8:
                    sample_size = 1;
                    break;
                case 16:
                    sample_size = 2;
                    break;
                default:
                    fprintf(stderr, "Wrong bit specification, 8 bit/sample
used.\n");
                    sample_size = 1;
                    break;
            }
```

```
        break;
    default:
        /*
        * set bits per sample to 8 - sample_size to 1 byte.
        */
        sample_size = 1;
        break;
    }
}

/*
 * Process extra arguments. (infile outfile)
 */
if (argc - optind != 2) {
    fprintf( stderr, "%s: USAGE: vox [-b 8|-b 16] infile outfile\n",
           argv[0]);
    exit(1);
}

/*
* Open the input file for reading.
*/
if ((infile = open(argv[optind], O_RDONLY)) < 0) {
    perror(argv[optind]);
    exit(1);
}

/*
* Open the output file for writing.
*/
if ((outfile = open(argv[++optind], O_WRONLY | O_CREAT, 0666)) < 0) {
    perror(argv[optind]);
    exit(1);
}

/*
* Read the input file and convert the samples to 12 bit --
* which is not support by Sound Blaster hardware, but is needed
* to accurately implement the Dialogic ADPCM algorithm.
*/
```

```
/*
* Allocate memory for the buffer of 12 bit data.
*/
buffer_size = 1024;
buffer12=(short*) calloc (buffer_size,sizeof(short));
/* Check that memory was allocated correctly */
if (buffer12==NULL) {
    fprintf (stderr,"%s: Malloc Error",argv[0] );
    exit(1);
}
/*
* Initialize the coder.
*/
adpcm_init( &coder_stat );
/*
* Allocate memory for the buffer of ADPCM samples.
*/
adpcm=(char*) calloc (buffer_size/2,sizeof(unsigned char));
/* Check that memory was allocated correctly */
if (adpcm==NULL) {
    fprintf (stderr,"%s: Malloc Error",argv[0] );
    exit(1);
}
/*
* Need different read commands for 8 bit and 16 bit data.
* Read the data; continue until end of file
*/

while ((n=read12(infile, sample_size, buffer12, buffer_size))>0 ) {
    /*
    * Convert data to Dialogic ADPCM format
    * Note that two ADPCM samples are stored in (8 bit) char,
    * because the ADPCM samples are only 4 bits.
    */
    j = 0;
    for(i=0; i<n/2; i++) {
        adpcm[i] = adpcm_encode( buffer12[j++], &coder_stat )<<4;
        if( j > n )  /* only true for last sample when n is odd */
            adpcm[i] |= adpcm_encode(0, &coder_stat);
        else
```

```
                adpcm[i] |= adpcm_encode(buffer12[j++], &coder_stat);
        }
        /*
        * now write the output file.
        */
        n /= 2;
        if(write(outfile, adpcm, n*sizeof(unsigned char)) < 0) {
            fprintf (stderr,"Error in writing file.");
            exit(1);
        }
    }
    /*
    * free allocated memory
    */
    free( buffer12 );
    free( adpcm );

    /*
    *  Close the input and output files
    */
    close(infile);
    close(outfile);

    exit(0);
}

/*
* program to read and convert data to 12 bit format.
*/

int read12(int infile,int sample_size, short *buffer12,int buffer_size)
{
    int i, n;
    short j, sign;
    unsigned char *buffer8;

    /*
    * The 8 bit case first.
    * The second bit of sample_size indicates whether 8 or 16 bit, hence
the
```

```
    * bitwise operation in the condition.
    */
    if (!(sample_size>>1)) {
        /*
        * Allocate memory for the buffer.
        */
        buffer8=(unsigned char*) calloc (buffer_size,sizeof(unsigned
char));
        /* Check that memory was allocated correctly */
        if (buffer8==NULL) {
            fprintf (stderr,"Malloc Error" );
            exit(1);
        }

        /*
        * Read the next block of data;
        */
        if ((n=read(infile,buffer8,buffer_size*sizeof(unsigned char)))<0 )
{
            fprintf (stderr,"Error in reading file.");
            exit(1);
        }
        /*
        * Convert the 8 bit samples to 12 bit
        * Need subtract 128 first because it is a unsigned char.
        */
        for (i=0; i<n; i++) {
            buffer12[i] = (short)buffer8[i] - 128;
            buffer12[i] *= 16;
        }
        free( buffer8 );
    }
    else {  /* now read the 16 bit data */
        /*
        * Read the next block of data;
        * Can use the 12-bit buffer to read the 16 bit data.
        */
        if ((n=read(infile,buffer12,buffer_size*sizeof(short)))<0 ) {
            fprintf (stderr,"Error in reading file.");
            exit(1);
```

```
        }
        /*
        * Convert the 16 bit samples to 12 bit.
        * Note that n is the number of bytes read, which is twice
        * the number samples read because sizeof(short) == 2.
        */
        n /= 2;
        for (i=0; i<n; i++) {
            buffer12[i] /= 16;
        }
    }
    return(n);
}
```

DEVOX.C CODE

```
/*  Filename: devox.c
    Description:  Kind of like the program sox.  It converts voice
          file formats.  Converts
          the Dialogic or Oki ADPCM (foo.vox or foo.32K) file format
          to 16 bit and 8 bit raw voice files.

    Usage: devox [-b 8|-b 16] infile outfile

        The -b is for 8 or 16 bit files (reference to input files)
        Default is 8 bits.
*/

#include <unistd.h>           /* needed for getopt */
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "adpcm.h"

void write12(int, int, short *, int );    /* program to write and convert
```

```
                                        * data to 12 bit format.
                                        */
int
main (int argc, char **argv)
{
    int c, i, j;
    extern char *optarg;
    extern int optind;
    int infile, outfile, n;
    int buffer_size, sample_size;
    short *buffer12;
    char *adpcm;
    struct adpcm_status coder_stat;

    /*
     * Process the arguments.
     */
    sample_size = 1;   /* default to 8 bit */
    while ((c = getopt(argc, argv, "b:")) != -1) {
        switch (c) {
        case 'b':
            /*
            * set bits per sample to 8 or 16 - sample_size to
            * 1 or 2 bytes.
            */
            switch (atoi(optarg)) {
                case 8:
                    sample_size = 1;
                    break;
                case 16:
                    sample_size = 2;
                    break;
                default:
                    fprintf(stderr, "Wrong bit specification, 8 bit/sample
used.\n");
                    sample_size = 1;
                    break;
            }
            break;
        default:
```

```
        /*
        * set bits per sample to 8 - sample_size to 1 byte.
        */
        sample_size = 1;
        break;
    }
}

/*
 * Process extra arguments. (infile outfile)
 */
if (argc - optind != 2) {
    fprintf( stderr, "%s: USAGE: devox [-b 8|-b 16] infile outfile\n",
           argv[0]);
    exit(1);
}

/*
* Open the input file for reading.
*/
if ((infile = open(argv[optind], O_RDONLY)) < 0) {
    perror(argv[optind]);
    exit(1);
}

/*
* Open the output file for writing.
*/
if ((outfile = open(argv[++optind], O_WRONLY | O_CREAT, 0666)) < 0) {
    perror(argv[optind]);
    exit(1);
}

/*
* convert the 12 bit samples linear to the final format of either
* 8 or 16 bit and write the output file.
*/
/*
* Allocate memory for the buffer of 12 bit data.
*/
```

```
    buffer_size = 1024;
    buffer12=(short*) calloc (buffer_size,sizeof(short));
    /* Check that memory was allocated correctly */
    if (buffer12==NULL) {
        fprintf (stderr,"%s: Malloc Error",argv[0] );
        exit(1);
    }
    /*
    * Initialize the coder.
    */
    adpcm_init( &coder_stat );
    /*
    * Allocate memory for the buffer of ADPCM samples.
    */
    adpcm=(char*) calloc (buffer_size/2,sizeof(char));
    /* Check that memory was allocated correctly */
    if (adpcm==NULL) {
        fprintf (stderr,"%s: Malloc Error",argv[0] );
        exit(1);
    }
    /*
    * Read the data; continue until end of file
    */
    while ((n=read(infile, adpcm, buffer_size*sizeof(char)/2))>0 ) {
        /*
        * Convert data to linear  format
        * Note that two ADPCM samples are stored in (8 bit) char,
        * because the ADPCM samples are only 4 bits.
        */
        j = 0;
        for(i=0; i<n; i++) {
            buffer12[j++] = adpcm_decode( (adpcm[i]>>4)&0x0f, &coder_stat
);
            buffer12[j++] = adpcm_decode( adpcm[i]&0x0f, &coder_stat );
        }
        /*
        * now convert from 12 bit to either 8 or 16 and write the output
file.
        */
        write12(outfile, sample_size, buffer12, n*2);
```

```
    }
    /*
    * free allocated memory
    */
    free( buffer12 );
    free( adpcm );

    /*
    *  Close the input and output files
    */
    close(infile);
    close(outfile);

    exit(0);
}

/*
* program to convert data from 12 bit format and write it.
*/

void write12(int outfile,int sample_size, short *buffer12,int buffer_size)
{
    int i;
    unsigned char *buffer8;

    /*
    * The 8 bit case first.
    * The second bit of sample_size indicates whether 8 or 16 bit, hence
the
    * bitwise operation in the condition.
    */
    if (!(sample_size>>1)) {
        /*
        * Allocate memory for the buffer.
        */
        buffer8=(unsigned char*) calloc (buffer_size,sizeof(unsigned
char));
        /* Check that memory was allocated correctly */
        if (buffer8==NULL) {
            fprintf (stderr,"Malloc Error" );
            exit(1);
```

```
        }
        /*
        * Convert the 12 bit samples to 8 bit
        * Need to add 128 because it is a unsigned char.
        */
        for (i=0; i<buffer_size; i++) {
            //buffer12[i] /= 16;
            buffer12[i] /= 32;
            buffer8[i] = (char)(buffer12[i] + 128);
        }
        /*
        * Write the next block of data;
        */
        if (write(outfile,buffer8,buffer_size*sizeof(char))<0 ) {
            fprintf (stderr,"Error in writing file.");
            exit(1);
        }
        free( buffer8 );
    }
    else {  /* now write the 16 bit data */

        /*
        * Convert the 12 bit samples to 16 bit.
        */
        for (i=0; i<buffer_size; i++) {
            buffer12[i] *= 16;
            // Compiler should implement at a shift left 4 bits.
            // The following was a temporary work around for a
            // clipping problem. -- believed to be fixed in version
            // 1.1 (see web page and adpcm.c line 99). TLB 3/30/04
            //buffer12[i] *= 8;
        }
        /*
        * write the next block of data;
        * Can use the 12-bit buffer for the 16 bit data.
        */
        if (write(outfile,buffer12,buffer_size*sizeof(short))<0 ) {
            fprintf (stderr,"Error in writing file.");
            exit(1);
        }
    }
    return;
}
```

ADPCM.H CODE

```
struct adpcm_status {
    short last;
    short step_index;
};

void adpcm_init(struct adpcm_status *);
char adpcm_encode( short, struct adpcm_status *);
short adpcm_decode( char, struct adpcm_status *);
```

ADPCM.C CODE

```
/*  File: adpcm.c
    Description: Routines to convert 12 bit linear samples to the
       Dialogic or Oki ADPCM coding format.
       I copied the algorithms out of the book "PC Telephony - The
       complete guide to designing, building and programming systems
       using Dialogic and Related Hardware" by Bob Edgar. pg 272-276.

*/

# include "adpcm.h"

/* Note: Edgar's book says that the second to last value is 1408; however,
 * The standard says it is 1411.
 * Changed on 1/17/2003.
*/

static short step_size[49] = { 16, 17, 19, 21, 23, 25, 28, 31, 34, 37, 41,
     45, 50, 55, 60, 66, 73, 80, 88, 97, 107, 118, 130, 143, 157, 173,
     190, 209, 230, 253, 279, 307, 337, 371, 408, 449, 494, 544, 598, 658,
     724, 796, 876, 963, 1060, 1166, 1282, 1411, 1552 };
/*
* one function local to this file only.
*/
```

```
short step_adjust ( char );

/*
* Initialze the data used by the coder.
*/
void adpcm_init( struct adpcm_status *stat ) {
    stat->last = 0;
    stat->step_index = 0;
    return;
}

/*
* Encode linear to ADPCM
*/
char adpcm_encode( short samp, struct adpcm_status *stat ) {
    short code;
    short diff, E, SS;

    /* printf( "%d\t", samp );
    */
    SS = step_size[stat->step_index];
    code = 0x00;
    if( (diff = samp - stat->last) < 0 )
        code = 0x08;
    E = diff < 0 ? -diff : diff;
    if( E >= SS ) {
        code = code | 0x04;
        E -= SS;
    }
    if( E >= SS/2 ) {
        code = code | 0x02;
        E -= SS/2;
    }
    if( E >= SS/4 ) {
        code = code | 0x01;
    }
/*    stat->step_index += step_adjust( code );
    if( stat->step_index < 0 ) stat->step_index = 0;
    if( stat->step_index > 48 ) stat->step_index = 48;
*/
```

```
    /*
    * Use the decoder to set the estimate of last sample.
    * It also will adjust the step_index for us.
    */
    stat->last = adpcm_decode(code, stat);
    return( code );
}

/*
* Decode Linear to ADPCM
*/
short adpcm_decode( char code, struct adpcm_status *stat ) {
    short diff, E, SS, samp;

    /* printf( "%x\t", code );
    */
    SS = step_size[stat->step_index];
    E = SS/8;
    if ( code & 0x01 )
        E += SS/4;
    if ( code & 0x02 )
        E += SS/2;
    if ( code & 0x04 )
        E += SS;
    diff = (code & 0x08) ? -E : E;
    samp = stat->last + diff;

    /*
    *  Clip the values to +(2^11)-1 to -2^11. (12 bits 2's
    *  compelement)
    *  Note: previous version errantly clipped at +2048, which could
    *  cause a 2's complement overflow and was likely the source of
    *  clipping problems in the previous version.  Thanks to Frank
    *  van Dijk for the correction.  TLB 3/30/04
    */
    if( samp > 2047 )
    {
      samp = 2047;
    }
    if( samp < -2048 )
    {
      samp = -2048;
    }
```

```
    stat->last = samp;
    stat->step_index += step_adjust( code );
    if( stat->step_index < 0 ) stat->step_index = 0;
    if( stat->step_index > 48 ) stat->step_index = 48;

    /* printf( "%d\n", samp );
    */
    return( samp );
}

/*
* adjust the step for use on the next sample.
*/
short step_adjust ( char code ) {
    switch( code & 0x07 ) {
        case 0x00:
            return(-1);
            break;
        case 0x01:
            return(-1);
            break;
        case 0x02:
            return(-1);
            break;
        case 0x03:
            return(-1);
            break;
        case 0x04:
            return(2);
            break;
        case 0x05:
            return(4);
            break;
        case 0x06:
            return(6);
            break;
        case 0x07:
            return(8);
            break;
    }
}
```