

NAME

soxexam—SoX Examples

EXAMPLES**Introduction**

The core problem is that you need some experience in using effects in order to say ‘that any old sound file sounds with effects absolutely hip’. There isn’t any rule-based system which tell you the correct setting of all the parameters for every effect. But after some time you will become an expert in using effects.

Here are some examples which can be used with any music sample. (For a sample where only a single instrument is playing, extreme parameter setting may make well-known ‘typically’ or ‘classical’ sounds. Likewise, for drums, vocals or guitars.)

Single effects will be explained and some given parameter settings that can be used to understand the theory by listening to the sound file with the added effect.

Using multiple effects in parallel or in series can result either in a very nice sound or (mostly) in a dramatic overloading in variations of sounds such that your ear may follow the sound but you will feel unsatisfied. Hence, for the first time using effects try to compose them as minimally as possible. We don’t regard the composition of effects in the examples because too many combinations are possible and you really need a very fast machine and a lot of memory to play them in real-time.

However, real-time playing of sounds will greatly speed up learning and/or tuning the parameter settings for your sounds in order to get that ‘perfect’ effect.

Basically, we will use the ‘play’ front-end of SoX since it is easier to listen sounds coming out of the speaker or earphone instead of looking at cryptic data in sound files.

For easy listening of file.xxx (‘xxx’ is any sound format):

```
play file.xxx effect-name effect-parameters
```

Or more SoX-like (for ‘dsp’ output on a UNIX/Linux computer):

```
sox file.xxx -t ossdsp -2 -s /dev/dsp effect-name effect-parameters
```

or (for ‘au’ output):

```
sox file.xxx -t sunau -2 -s /dev/audio effect-name effect-parameters
```

And for date freaks:

```
sox file.xxx file.yyy effect-name effect-parameters
```

Additional options can be used. However, in this case, for real-time playing you’ll need a very fast machine.

Notes:

I played all examples in real-time on a Pentium 100 with 32 MB and Linux 2.0.30 using a self-recorded sample (3:15 min long in ‘wav’ format with 44.1 kHz sample rate and stereo 16 bit). The sample should not contain any of the effects. However, if you take any recording of a sound track from radio or tape or CD, and it sounds like a live concert or ten people are playing the same rhythm with their drums or funky-grooves, then take any other sample. (Typically, less than four different instruments and no synthesizer in the sample is suitable. Likewise, the combination vocal, drums, bass and guitar.)

Echo

An echo effect can be naturally found in the mountains, standing somewhere on a mountain and shouting a single word will result in one or more repetitions of the word (if not, turn a bit around and try again, or climb to the next mountain).

However, the time difference between shouting and repeating is the delay (time), its loudness is the decay. Multiple echos can have different delays and decays.

It is very popular to use echos to play an instrument with itself together, like some guitar players (Queen’s

Brain May) or vocalists do. For music samples of more than one instrument, echo can be used to add a second sample shortly after the original one.

This will sound as if you are doubling the number of instruments playing in the same sample:

```
play file.xxx echo 0.8 0.88 60 0.4
```

If the delay is very short, then it sound like a (metallic) robot playing music:

```
play file.xxx echo 0.8 0.88 6 0.4
```

Longer delay will sound like an open air concert in the mountains:

```
play file.xxx echo 0.8 0.9 1000 0.3
```

One mountain more, and:

```
play file.xxx echo 0.8 0.9 1000 0.3 1800 0.25
```

Echos

Like the echo effect, echos stand for 'ECHO in Sequel', that is the first echos takes the input, the second the input and the first echos, the third the input and the first and the second echos, ... and so on. Care should be taken using many echos (see introduction); a single echos has the same effect as a single echo.

The sample will be bounced twice in symmetric echos:

```
play file.xxx echos 0.8 0.7 700 0.25 700 0.3
```

The sample will be bounced twice in asymmetric echos:

```
play file.xxx echos 0.8 0.7 700 0.25 900 0.3
```

The sample will sound as if played in a garage:

```
play file.xxx echos 0.8 0.7 40 0.25 63 0.3
```

Chorus

The chorus effect has its name because it will often be used to make a single vocal sound like a chorus. But it can be applied to other instrument samples too.

It works like the echo effect with a short delay, but the delay isn't constant. The delay is varied using a sinusoidal or triangular modulation. The modulation depth defines the range the modulated delay is played before or after the delay. Hence the delayed sound will sound slower or faster, that is the delayed sound tuned around the original one, like in a chorus where some vocals are a bit out of tune.

The typical delay is around 40ms to 60ms, the speed of the modulation is best near 0.25Hz and the modulation depth around 2ms.

A single delay will make the sample more overloaded:

```
play file.xxx chorus 0.7 0.9 55 0.4 0.25 2 -t
```

Two delays of the original samples sound like this:

```
play file.xxx chorus 0.6 0.9 50 0.4 0.25 2 -t 60 0.32 0.4 1.3 -s
```

A big chorus of the sample is (three additional samples):

```
play file.xxx chorus 0.5 0.9 50 0.4 0.25 2 -t 60 0.32 0.4 2.3 -t 40 0.3 0.3 1.3 -s
```

Flanger

The flanger effect is like the chorus effect, but the delay varies between 0ms and maximal 5ms. It sound like wind blowing, sometimes faster or slower including changes of the speed.

The flanger effect is widely used in funk and soul music, where the guitar sound varies frequently slow or a bit faster.

Now, let's groove the sample:

play file.xxx flanger

listen carefully between the difference of sinusoidal and triangular modulation:

play file.xxx flanger triangle

Reverb

The reverb effect is often used in audience hall which are too small or contain too many visitors which disturb (dampen) the reflection of sound at the walls. Reverb will make the sound be perceived as if it were in a large hall. You can try the reverb effect in your bathroom or garage or sport hall by shouting loud some words. You'll hear the words reflected from the walls.

The biggest problem in using the reverb effect is the correct setting of the (wall) delays such that the sound is realistic and doesn't sound like music playing in a tin can or has overloaded feedback which destroys any illusion of playing in a big hall. To help you obtain realistic reverb effects, you should decide first how long the reverb should take place until it is not loud enough to be registered by your ears. This is done by varying the reverb time 't'. To simulate small halls, use 200ms. To simulate large halls, use 1000ms. Clearly, the walls of such a hall aren't far away, so you should define its setting by given every wall its delay time. However, if the wall is too far away for the reverb time, you won't hear the reverb, so the nearest wall will be best at 't/4' delay and the farthest at 't/2'. You can try other distances as well, but it won't sound very realistic. The walls shouldn't stand too close to each other and not in a multiple integer distance to each other (so avoid wall like: 200 and 202, or something like 100 and 200).

Since audience halls do have a lot of walls, we will start designing one beginning with one wall:

play file.xxx reverb 1 600 180

One wall more:

play file.xxx reverb 1 600 180 200

Next two walls:

play file.xxx reverb 1 600 180 200 220 240

Now, why not a futuristic hall with six walls:

play file.xxx reverb 1 600 180 200 220 240 280 300

If you run out of machine power or memory, then stop as many applications as possible (every interrupt will consume a lot of CPU time which for bigger halls is absolutely necessary).

Phaser

The phaser effect is like the flanger effect, but it uses a reverb instead of an echo and does phase shifting. You'll hear the difference in the examples comparing both effects. The delay modulation can be sinusoidal or triangular, preferable is the later for multiple instruments. For single instrument sounds, the sinusoidal phaser effect will give a sharper phasing effect. The decay shouldn't be too close to 1 which will cause dramatic feedback. A good range is about 0.5 to 0.1 for the decay.

We will take a parameter setting as before (gain-out is lower since feedback can raise the output dramatically):

play file.xxx phaser 0.8 0.74 3 0.4 0.5 -t

The drunken loudspeaker system (now less alcohol):

play file.xxx phaser 0.9 0.85 4 0.23 1.3 -s

A popular sound of the sample is as follows:

play file.xxx phaser 0.89 0.85 1 0.24 2 -t

The sample sounds if ten springs are in your ears:

play file.xxx phaser 0.6 0.66 3 0.6 2 -t

Compander

The compander effect allows the dynamic range of a signal to be compressed or expanded. For most situations, the attack time (response to the music getting louder) should be shorter than the decay time because our ears are more sensitive to suddenly loud music than to suddenly soft music.

For example, suppose you are listening to Strauss's 'Also Sprach Zarathustra' in a noisy environment such as a car. If you turn up the volume enough to hear the soft passages over the road noise, the loud sections will be too loud. You could try this:

```
play file.xxx compand 0.3,1 -90,-90,-70,-70,-60,-20,0,0 -5 0 0.2
```

The transfer function ('-90,...') says that *very* soft sounds between -90 and -70 decibels (-90 is about the limit of 16-bit encoding) will remain unchanged. That keeps the compander from boosting the volume on 'silent' passages such as between movements. However, sounds in the range -60 decibels to 0 decibels (maximum volume) will be boosted so that the 60dB dynamic range of the original music will be compressed 3-to-1 into a 20dB range, which is wide enough to enjoy the music but narrow enough to get around the road noise. The -5 dB output gain is needed to avoid clipping (the number is inexact, and was derived by experimentation). The 0 for the initial volume will work fine for a clip that starts with a bit of silence, and the delay of 0.2 has the effect of causing the compander to react a bit more quickly to sudden volume changes.

The following (one long) command shows how multi-band companding is typically used in FM radio:

```
play file.xxx
vol -3 db filter 8000- 32 100 mcompand "0.005,0.1
-47,-40,-34,-34,-17,-33 0 0 0" 100 "0.003,0.05
-47,-40,-34,-34,-17,-33 0 0 0" 400 "0.000625,0.0125
-47,-40,-34,-34,-15,-33 0 0 0" 1600 "0.0001,0.025
-47,-40,-34,-34,-31,-31,-0,-30 0 0 0" 6400
"0,0.025 -38,-31,-28,-28,-0,-25 0 0 0" vol 27 db vol -12
db highpass 22 highpass 22 filter -17500 256 vol +12 db
vol -3 db lowp 17801
```

The audio file is played with a simulated FM radio sound (or broadcast signal condition if the lowp at the end is skipped). Note that the pipeline is set up with US-style 75us preemphasis.

Changing the Rate of Playback

You can use stretch to change the rate of playback of an audio sample while preserving the pitch. For example to play at half the speed:

```
play file.wav stretch 2
```

To play a file at twice the speed:

```
play file.wav stretch 0.5
```

Other related options are 'speed' to change the speed of play (and changing the pitch accordingly), and pitch, to alter the pitch of a sample. For example to speed a sample so it plays in half the time (for those Mickey Mouse voices):

```
play file.wav speed 2
```

To raise the pitch of a sample 1 while note (100 cents):

```
play file.wav pitch 100
```

Reducing noise in a recording

First find a period of silence in your recording, such as the beginning or end of a piece. If the first 1.5 seconds of the recording are silent, do

```
sox file.wav -n trim 0 1.5 noiseprof /tmp/profile
```

Next, use the noisered effect to actually reduce the noise:

```
play file.wav noisered /tmp/profile
```

Making a recording

Thanks to Douglas Held for the following suggestion:

```
rec parameters filename other-effects silence 1 5 2%
```

This use of the **silence** effect allows you to start a recording session but only start writing to disk once non-silence is detected. For example, use this to start your favorite command line for recording and walk over to your record player and start the song. No periods of silence will be recorded.

Scripting with SoX

One of the benefits of a command-line tool is that it is easy to use it in scripts to perform more complex tasks. In marine radio, a Mayday emergency call is transmitted preceded by a 30-second alert sound. The alert sound comprises two audio tones at 1300Hz and 2100Hz alternating at a rate of 4Hz. The following shows how SoX can be used in a script to construct an audio file containing the alert sound. The scripting language shown is 'Bourne shell' (sh) but it should be simple to translate this to another scripting language if you do not have access to sh.

```
# Make sure we append to a file that's initially empty:
```

```
rm -f 2tones.raw
```

```
for freq in 1300 2200; do
```

```
  sox -c1 -r8000 -n -t raw - synth 0.25 sine $freq vol 0.7 >> 2tones.raw
```

```
done
```

```
# We need 60 copies of 2tones.raw (0.5 sec) to get 30 secs of audio:
```

```
iterations=60
```

```
# Make sure we append to a file that's initially empty:
```

```
rm -f alert.raw
```

```
while [ $iterations -ge 1 ]; do
```

```
  cat 2tones.raw >> alert.raw
```

```
  iterations=`expr $iterations - 1`
```

```
done
```

```
# Add a file header and save some disc space:
```

```
sox -sw -c1 -r8000 alert.raw alert.ogg
```

```
play alert.ogg
```

If you try out the above script, you may want to hit Ctrl-C fairly soon after the alert tone starts playing—it's not a pleasant sound! The **synth** effect is used to generate each of the tones; **-c1 -r8000** selects mono, 8kHz sampling-rate audio (i.e. relatively low fidelity, suitable for the marine radio transmission channel); each tone is generated at a length of 0.25 seconds to give the required 4Hz alternation. Note the use of 'raw' as the intermediary file format; a self-describing (header) format would just get in the way here. The self-describing header is added only at the final stage; in this case, **.ogg** is chosen, since lossy compression is appropriate for this application.

There are further practical examples of scripting with SoX available to download from the SoX web-site [1].

SEE ALSO

```
sox(1), libst(3)
```

References

- [1] *SoX—Sound eXchange / Scripts*, <http://sox.sourceforge.net/Docs/Scripts>

AUTHORS

This man page was written largely by Juergen Mueller (jmueller@uia.ua.ac.be). Other SoX authors and contributors are listed in the AUTHORS file that is distributed with the source code.