Composers' Desktop Project

CDP SFEDIT Functions

(with Command Line Usage)

Functions to Edit soundfiles

(Names in brackets mean that these are separate programs. The others are sub-modules of SFEDIT.)

[CANTOR]
Cut holes in a sound in the manner of a cantor set (holes within holes within holes)
[CONSTRICT]
Shorten the durations of any zero-level sections in a sound
CUT
Cut and keep a segment of a sound
CUTEND
Cut out and keep the end part of a soundfile
CUTMANY
Cut and keep several segments of a sound
EXCISE
Remove a segment from a soundfile and close up the gap
EXCISES
Remove segments of a soundfile and close up the gaps
INSERT
Insert 2nd sound into 1st (overwriting or spreading first sound)
INSIL
Insert silence into a sound (overwriting or spreading the sound apart)
[ISOLATE]
Disjunct portions of soundfile are specified by textfile or dB loudness and saved to separate files
JOIN
Join files together, one after another
JOINDYN
Join in loudness-patterned sequence
JOINSEQ
Join in patterned sequence
[MANYSIL]
Insert many silences into a soundfile
MASKS
Mask specified chunks of a sound, with silence
NOISECUT
Suppress noise in a (mono) sound file, replacing with silence
[PACKET]
Isolate or generate a sound packet
[PARTITION]
Partition a mono soundfile into disjunct files in blocks defined by groups of wavesets
[PREFIX SILENCE]
Add silence to the beginning of a soundfile

CANTOR – Cut holes in a sound in the manner of a cantor set (holes within holes within holes)

Usage

cantor set 1-2 *infile outfile holesize holedig depth-trig splicelen maxdur* [**-e**]
cantor set 3 *infile outfile holelev holedig layercnt layerdec maxdur*

Example command line to create cantor set type holes in a soundfile:

```
cantor set 1 insndfile.wav outsndfile.wav 0.5 0.5 0.1 5 8
```

Modes

1 *holesize* is a percentage
2 *holesize* is a (fixed) duration
3 Use superimposed vibrato envelopes

Parameters

infile – input soundfile (mono)
outfile – output soundfile: use a generic root name for the output soundfiles. Numerals starting at 0 are appended to distinguish the outputs.
Modes 1 & 2:
holesize – Mode **1**: the percentage of current segment-time taken up by a hole. The size of the hole depends on the size of the segment being cut. Mode **2**: the (fixed) duration of the holes
holedig – the depth of each cut as the hole is gradually created. Range: >0 to 1
depth-trig – the level depth of the hole triggering the next hole-cutting
splicelen – splicelength in milliseconds
maxdur – the maximum output duration of all the output sound
-e – extend the sound beyond the *splicelen* limits
Mode 3:
holelev – the level of signal at the base of the holes
holedig – how many repeats before full-depth is reached
layercnt – the number of vibrato layers used
layerdec – the depth of the next vibrato in relation to the previous one
maxdur – the maximum total duration of all the output sound

Understanding the CANTOR SET Process

CANTOR gradually cuts a hole in the central third of the input sound, which must be mono, a 'hole' being a reduction in level (see **SNDINFO FINDHOLE**). It then cuts holes in the central third of the remaining segments, and so on. The output soundfile consists of a sequence of sounds with more and more holes cut in it. Note that Mode **3** uses superimposed vibrato envelopes as well.

End of CANTOR

CONSTRICT – Shorten the durations of any zero-level sections in a sound

Usage

constrict constrict *infile outfile constriction*

Example command line to time-contract silent gaps in a soundfile :

```
    constrict constrict count 50
```

Parameters

infile – input soundfile
outfile – output soundfile
constriction – percentage deletion of zero-level areas. For example, *constriction* = 20 reduces any silences by 20% or one-fifth.

Understanding the CONSTRICT Process

This is a form of time-contraction which does not time-stretch the sonic-substance of the source. Note that the *constriction* is a percentage. Thus:

- 50 will halve the silent gaps
- 75 will take away three quarters of the silent gaps
- 20 will reduce them to one-fifth their original length
- 100 will eliminate the silences
- 150 will cause the non-silent portions to **overlap** by half the duration of the original silences

It works only with soundfiles which contain areas of zero-level signal which can be contracted by the process.

Musical Applications

CONSTRICT can be used to take a sequence of rapidfile events separated by silences and make them even tighter.

End of CONSTRICT

SFEDIT CUT – Cut and keep a segment of a sound

Usage

sfedit cut mode *infile outfile start end* [**-w***splice*]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – cut section saved as new soundfile
start – time in *infile* where segment to keep begins
end – time in *infile* where segment to keep ends
-w*splice* – splice window in milliseconds (Default: 15ms)

Understanding the SFEDIT CUT Process

The start and end locations for a block of sound are specified, and that block is saved as a new soundfile. A long splice will give a smooth cutoff, a short splice an abrupt cutoff, and a zerio splice will usually produce a click. The splice window is applied to both the beginning and the end of the sound, so cannot be larger than half the length of the block.

This function presumes the use of a graphic sound editor in order precisely to locate the places at which to begin end the block to be cut. CDP's VIEWSF provides a display accurate to the individual sample (Zoom level 0), so very precise locations can be specified. Marks can be saved to a textfile for future reference.

In *Sound Loom*, ALT CLICK on a **Workspace** file, or clicking on the (new) **View Source** button on the **Parameters** page, gives a graphic display of the file and the facility to play any selected part of it (and then cut the segment, if desired).

Musical Applications

Musical applications are many and varied:

- simply save a favoured portion of a soundfile
- select a short, timbrally evolving, section of a soundfile and time-stretch it, cut part of the result and time-stretch again, etc.

- process a portion of a sound, and then reinsert it into the original. For example, isolate key portions of two sounds and pre-process each of them in preparation for a morph-transition; then reinsert them into their respective original soundfiles and do the morph

- collect segments of various soundfiles in preparation for making a musical collage

  Removing silence or unwanted portions of a soundfile should be done with **SFEDIT EXCISE**.

End of SFEDIT CUT

SFEDIT CUTEND – Cut and keep the end portion of a sound

Usage

sfedit cutend mode *infile outfile length* [**-w***splice*]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – cut section saved as new soundfile
length – length of sound to keep, ending at the end of *infile*
-w*splice* – splice window in milliseconds (Default: 15ms)

Understanding the SFEDIT CUTEND Process

This function enables you to use a specified *length* of the last section of a sound without having to work out where that length begins. You just specify the length you want. Needless to say, it has to be shorter than the whole soundfile.

Musical Applications

The way a sound ends varies a great deal and often has extra attributes, such as resonance, reverb or echoes. This material can therefore be useful in itself. For example, a piano tone starts percussively and ends gradually if left to ring on. When reversed, the sound swells, sounding very much like an organ. SFEDIT CUTEND can quickly cut the end portion, starting automatically after the beginning. Given a generous splice envelope and reversed with **MODIFY RADICAL** Mode 1, it can become quite a different sound altogether.

End of SFEDIT CUTEND

SFEDIT CUTMANY – Cut and keep several segments of a sound

Usage

sfedit cutmany mode *infile outgenericfilename cuttimes splicelen*

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel-count
3 Time as grouped-sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – generic output filename (the numbers '1', '2' etc. are added to this generic name to name the various output soundfiles)
cuttimes – text file of time-pairs for the start and end of each segment
splicelen – the duration of the splice window in milliseconds: i.e., the amount of time to rise from and fall back to zero amplitude. **NB:** REQUIRED (not optional as in the other CUT functions).

Understanding the SFEDIT CUTMANY Process

This is an extension of the basic "edit cutout and keep" (**SFEDIT CUT**) to allow several segments to be cut from a file at a single pass. The *start* and *end* times of the cuts are placed in a textfile which the process reads.

Musical Applications

This function is useful if you want to extract several interesting features from a source sound. You can search the sound first, noting down the edit times of the sections you want, write them in a textfile, and then use the textfile to cut those segments from the source in a single pass, saving them to a generic name. For example, if your generic name is **pop**, the various cuts will be named **pop1**, **pop2** etc.).

ALSO SEE **SFEDIT SYLLABLES**.

End of SFEDFIT CUTMANY

SFEDIT EXCISE – Remove a segment from a soundfile and close up the gap

Usage

sfedit excise mode *infile outfile start end* [**-w***splice*]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – cut section saved as new soundfile
start – time in *infile* where segment to remove begins
end – time in *infile* where segment to remove ends
-w*splice* – splice window in milliseconds (Default: 15ms)

Understanding the SFEDIT EXCISE Process

Here the block *start* and *end* points mark a block to be removed. The size of the splice determines the smoothness (long splice) or abruptness (short splice) of the cuts. A zero splice usually creates a click at the splice point, except in the special case where the signal is zero. The splice 'window' enables the use to reshape the amplitude envelope at the point where the cuts are made.

Musical Applications

A frequent use of this program will be to remove silence, glitches, or otherwise unwanted material from a sound. It could also be used to chop up a sound in a rough sort of way in order to create unexpected juxtapositions of material, e.g., words.

SFEDIT EXCISE can be used to 'top and tail' a sound (remove silence at the beginning and the end) when special attention to detail is needed. Otherwise, **HOUSEKEEP EXTRACT** Mode 3 can be used.

ALSO SEE: **SFEDIT EXCISES** which uses a text file of cut points to remove several chunks in one operation.

End of SFEDIT EXCISE

SFEDIT EXCISES – Remove segments of a soundfile and close up the gaps

Usage

sfedit excises mode *infile outfile excisefile* [**-w***splice*]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – cut sections joined up and saved as new soundfile
excisefile – text file with (paired) start and end times of chunks to be removed. These must be in increasing time order.
-w*splice* – splice window in milliseconds (Default: 15ms)

Understanding the SFEDIT EXCISES Process

The times in *excisefile* are given in seconds, a pair on each line, separated by a space or a tab.

Musical Applications

Multiple cuts may be useful when removing a series of glitches, or when chopping up a sound as mentioned above in **SFEDIT EXCISE** to create unexpected juxtapositions: collage techniques.

End of SFEDIT EXCISES

SFEDIT INSERT – Insert 2nd sound into 1st (overwriting or spreading first sound)

Usage

sfedit insert mode *infile insert outfile time* [**-w***splice*] [**-l***level*] [**-o**]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
insert – soundfile ('chunk') to insert
outfile – combination saved as new soundfile
time – time in seconds in *infile* at which the insert is to begin
-w*splice* – splice window in milliseconds (Default: 15ms)
-l*level* – gain multiplier on inserted file (Default: 1.0)
-o – overwrite the original file with the inserted file (Default: the insert pushes the *infile* apart)

Understanding the SFEDIT INSERT Process

Note the difference between placing a sound into the midst of another sound (pushing apart the two separated portions of the original), and actually overwriting the original. In the first instance, none of the original is lost. In the second, that part of the original which lasts until the end of the insert is lost – but if there is still more original soundfile after this point, it will carry on after the insert has finished.

Musical Applications

Besides normal joinings and juxtapositions, SFEDIT INSERT can be used with more far-reaching objectives in mind. For example, a soundfile could be constructed out of widely diverse materials in order to pave the way for timbral transformations which will greatly alter the original sources (making them unrecognisable). E.g., blur, trace, extract spectral envelope, spread peaks, invert spectrum ...

ALSO SEE: SFEDIT REPLACE.

End of SFEDIT INSERT

SFEDIT INSIL – Insert silence into a sound (overwriting or spreading the sound apart)

Usage

sfedit insil mode *infile outfile time duration* [**-w***splicelen*] [**-o**] [**-s**]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – combination saved as new soundfile
time – time in seconds in *infile* at which the silence is to begin
duration – length of silence in seconds
-w*splicelen* – splice window in milliseconds (Default: 15ms)
-o – overwrite the original file with the inserted file (Default: the silence pushes the *infile* apart)
-s – retains any silence written over file end (Default: rejects silence added at file end)

Understanding the SFEDIT INSIL Process

This process will create a gap in the *infile* at a specified point in time, either pushing apart or overwriting the original sound for the duration of the silence.

Musical Applications

This can be used at the beginning of a sound to 'hard-wire' a gap into a mix. Another application would be to spread the timing of two events in a sound by a specified amount.

ALSO SEE: PREFIX SILENCE and **SILEND**

End of SFEDIT INSIL

ISOLATE – Disjunct portions of soundfile are specified by textfile or dB loudness and saved to separate files

Usage

isolate isolate 1-2 *insndfile outnam cutsfile* [**-s***splice*] [**-x**] [**-r**]
isolate isolate 3 *insndfile outnam dBon dBoff* [**-s***splice*] [**-m***min*] [**-l***len*] [**-x**] [**-r**]
isolate isolate 4 *insndfile outname slicefile* [**-s***splice*] [**-x**] [**-r**]
isolate isolate 5 *insndfile outname slicefile* [**-s***splice*] [**-d***dovetail*] [**-x**] [**-r**]

Example command line to create moulded snippets:

```
isolate isolate 1 asound snip cuts.txt
```

Modes

1 Create *several* output soundfiles each of which contains *one* segment of source (*cutsfile*)
2 Create *several* output soundfiles each of which contains *several* segments of source (*cutsfile*)
3 Create *one* output soundfile consisting of *several* disjunct segments (*dBon* & *dBoff*)
4 Cut the *entire* soundfile into disjunct segments (*slicefile*)
5 Cut as in Mode **4** but also *overlap* the segments slightly: separates speech syllables (*slicefile*)

Parameters

insndfile – input soundfile
outname – generic root name for the output soundfiles. Numerals starting at 0 are appended to distinguish the outputs.
cutsfile – a textfile containing *start end* time-pairs in (increasing) time order at which to cut segments. Note that the file for Mode **3** is allows overlaps:

- Modes **1-2**: none of the segments in any of the lines of the file are allowed to overlap
- Mode **3**: in this Mode the *cutsfile* is the same, but the segment starts and ends are located using threshold-on and threshold-off (*dBon* and *dBoff*). Also, if *len* is set, only the start portion of a segment of length *len* is kept.
- In Modes **1-3** an *extra file of remnants* (if any) is created. Also, note that if cuts abutt or are so close that *end+start* splices overlap, the end of the first cut is moved back, and the start of the second cut is moved forward such that they will overlap by (only) a single splicelength.

dBon – Mode **3**: the dB level at which a segment is recognised
dBoff – Mode **3**: the dB level at which a recognised segment is triggered to end
slicefile – Modes **4 & 5**: a textfile containing a list of increasing times at which the sound is to be cut. Note that for Mode **5** the times are designed to overlap slightly.
-s*splice* – the length of the splice in milliseconds (Range: 0 to 500. Default: 15)
-m*min* – Mode **3**: the minimum duration in milliseconds of segments to accept (Range: > 2 * *splice*)

-l*len* – Mode **3**: the duration in milliseconds of the (part-)segments to keep
-d*dovetail* – Mode **5**: the overlap of cut segments in milliseconds (Range: 0 to 20. Default 5)
-x – add silence to the end of the segments files so they become the same length as the source
-r – reverse all the cut-segment files (but not the remnant file)

Understanding the ISOLATE Process

ISOLATE is similar to **PARTITION** in that the primary operation is to cut a soundfile into disjunct pieces and assign these to different output soundfiles. It does this in a special way so that the time-position of these pieces in the original soundfile is retained: silence is inserted between the cut pieces in the outputs to achieve this ('silent surrounds'). The result is that the disjunct pieces can be reassembled in their original positions (remixed with everything synchronised at time zero):

One difference between ISOLATE and PARTITION is that the number of output files is user-defined in PARTITION, whereas in ISOLATE it depends on the Mode selected. Another difference is that Modes **1**, **2** and **3** of ISOLATE generate a file containing all the materials left over after cutting. This is also used in reconstructing the original (now-treated) soundfile.

However, the main difference between ISOLATE and PARTITION is how the disjunct pieces are identified. In PARTITION there are automatic processes for this, but the number of output files is user-defined by the *outcnt* parameter, and the durations are controlled by the *groupcnt* or *dur* parameters. In ISOLATE the user specifies *start end* times for the pieces in a textfile: the *cutsfile* (Modes **1-2**) or specifies a list of (increasing) times in *splicefile* Modes **4-5**, or they are picked up by level thresholds (*dBon* and *dBoff* (Mode **3**).

You are advised to familiarise yourself with the way the different Modes create different numbers of output soundfiles containing different numbers of segments.

Musical Applications

The main purpose of ISOLATE is to be able to treat the different segment-streams in some way before they are reassembled. Any processes that do not alter the segment lengths could be considered suitable.

ALSO SEE: PARTITION

End of ISOLATE

SFEDIT JOIN – Join files together, one after another

Usage

sfedit join *infile1 infile2* [*infile3 ...*] *outfile* [**-w***splicelen*] [**-b**] [**-e**]

Parameters

infile1 – first soundfile to splice
infile2 infile3 ... – additional soundfiles to splice
outfile – resultant soundfile output
-w*splicelen* – duration of splice in milliseconds (Default: 15ms)
-b – splice slope at start of first file
-e – splice slope at end of last file

Understanding the SFEDIT JOIN Process

Splicing is joining soundfiles together. The joins can be 'butt' or sloped. A butt join means that the sounds are butted up against each other just as they are, with no overlap and no slope other than what they may already possess. This can be done by specifying 0 for *splice*. Unless your sounds are already spliced at the start and end, this will almost always produce a click at the edit point.

Sloped joins either use the default overlap of 15 ms or specify another *splicelen* time. Longer times mean more overlap and more gradual changes in relative amplitude, the preceding sound getting softer while the following sound gets louder. Splice times are not restricted by the software, but if both **-b** and **-e** are used, *splice* cannot be greater than half the length of the sound.

[Thanks to Gustav Ciamaga for pointing out the following.]
SFEDIT JOIN requires at least 2 infiles, usually different but they can also be the same. In this instance, one could also taper (fade-in/fade-out) the beginning and/or end of the resultant repeated sound. To be more specific: you would use the same soundfile as both *infile1* and *infile2*. In this instance the beginning and end of the resultant repeated sound can both be tapered (fade-in / fade-out). Use the **-b** flag to set the fade-in at the beginning and the **-e** flag to set the fade-out at the end (of the 2nd, repeated, sound). You can confirm this with a splice window of 1000 ms (**-w**1000).

Thus, when the same sound is used several times: as *infile1*, *infile2* or more times, this is one way to achieve repetitions or pulsations, depending on the nature of the sound.

Musical Applications

Splicing is one of the basic assembly procedures used in electroacoustic music. A butt join achieves maximum contrast and/or a join with no loss of time (no overlap), but there is a danger of clicks. Sometimes a portion of soundfile is removed, processed and replaced. In this case, it is best to use VIEWSF in single sample view mode in order to edit with sample accuracy as close to zero as possible.

Overlaps smooth the joins and reduce the possibility of clicks or other unwanted 'bumps'. Long splice times achieve a smooth flow of one sound into the other without going as far as a full-length crossfade or morph.

ALSO SEE: **SUBMIX CROSSFADE**, and the various **MORPH** functions.

End of SFEDIT JOIN

SFEDIT JOINDYN – Join soundfiles in loudness-patterned sequence

Usage

sfedit joindyn *infile* [*infile2 ...*] *outfile pattern* [**-w***splicelen*] [**-b**] [**-e**]

Parameters

infile – first input soundfile to join
infile2... – optional second or more soundfiles to join
outfile – resultant soundfile output
pattern – text file containing a pattern of *soundfile level* pairs. The soundfiles are identified by numbers, with the numbering following the order in which they are listed, starting with the number 1. *Level* range is 0 to 1. Example (repeating and fading):

```
[sfile level]
1        0.50
2        0.75
3        1.00
3        0.90
2        0.85
2        0.75
2        0.65
1        0.70
1        0.50
1        0.40
1        0.30
```

-w*splicelen* – duration of splice window in milliseconds. Default: 15ms (optional)
-b – option to apply splice to the start of the first sound
-e – option to apply splice to the end of the last sound

Understanding the SFEDIT JOINDYN Process

As with **SFEDIT JOINSEQ**, this function splices together soundfiles in the order in which they are listed. In addition, it specifies the relative loudness of each soundfile in the sequence. Note that in *Sound Loom* similar patterns can be created as *mixfiles*, giving you the possibility to change the entry times of the sounds. This involves an advanced use of the TABLE EDITOR.

Also note that **EXTEND SEQUENCE2** allows you to put several sounds into a patterned sequence using any timing sequence (and patterns of levels).

Musical Applications

SFEDIT JOINDYN provides a quick and direct way to create a group of soundfiles in any order, with patterns of dynamic level.

End of SFEDFIT JOINDYN

SFEDIT JOINSEQ – Join soundfiles in patterned sequence

Usage

sfedit joinseq *infile* [*infile2 ...*] *outfile pattern* [**-w***splicelen*] [**-m***maxlen*] [**-b**] [**-e**]

Parameters

infile – first input soundfile to join
infile2... – optional second or more soundfiles to join
outfile – resultant soundfile output
pattern – text file containing a pattern of numbers specifying the sequence (ordering) of soundfiles to use. The soundfiles are identified by numbers, with the numbering following the order in which they are listed, starting with the number 1. Example (a permutation):

```
1 2 3  2 3 1  3 1 2  3 2 1  2 1 3  1 3 2
```

-w*splicelen* – duration of splice window in milliseconds. Default: 15ms (optional)
-m*maxlen* – maximum number of items in pattern to use
-b – option to apply splice to the start of the first sound
-e – option to apply splice to the end of the last sound

Understanding the SFEDIT JOINSEQ Process

SFEDIT JOINSEQ allows a set of sounds to be joined, end to end, in a pattern. Any sound in the sequence can be repeated any number of times.

The *pattern* is specified in a text file, with the sound pattern value separated by spaces or newlines. The soundfiles are identified by numbers, with the numbering following the order in which they are listed, starting with the number 1. Thus only the numbers are needed in the *pattern* file, as illustrated above.

Maxlen enables you to use an existing *pattern* with fewer active components (i.e., apply data reduction).

Musical Applications

This would be a way to generating patterns of events, such as sequences of vocal syllables, with a melodic flavour – given that the source material has sufficiently different pitch levels and that you are going the use the whole length of each sound before the next sound begins(this is only a splice operation).

End of SFEDFIT JOINSEQ

MANYSIL – Insert many silences into a soundfile

Usage

manysil manysil *infile outfile silencedata splicelen*

Example command line to insert several silences into a sound:

```
    manysil manysil flocalc707 flosils manysils.txt 30
```

Parameters

infile – input soundfile
outfile – output soundfile
silencedata – a textfile with *time duration* pairs: *time* is the insertion time of the silence, *duration* is the duration of the inserted silence.
splicelen – length in milliseconds of the splices used

Understanding the MANYSIL Process

MANYSIL extends idea of inserting silence, to many silences. Using MANYSIL is straightforward, focusing on setting the times and durations of the silences in the data file.

Here is a sample *silencedata* file:

Start-time	Duration
0.5	0.5
1.0	1.0
1.5	1.5
2.0	2.0
2.5	1.5
3.0	1.0
3.5	0.5
4.0	3.0

Note that on the *Sound Loom* the **Sound View** Window allows you to mark the time and duration of each insertion. This is done by drawing out a box of the required duration. The time sets the start point for each silent section. There is one box for each inserted silence. **Sound View** then **automatically transfers this data** to the textfile format required.

Musical Applications

MANYSIL enables us to set up a number of silences all at once. This might be simply to separate data, or one may want to introduce a duration pattern of silences into the sound.

End of MANYSIL MANYSIL

SFEDIT MASKS – Mask specified chunks of a sound, with silence

Usage

sfedit masks mode *infile outfile excisefile* [**-w***splice*]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel-count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – cut segment saved as a new soundfile
excisefile – a textfile with (paired) start and end times of chunks to be masked. These must be in increasing time order.
-w*splice* – splice window in milliseconds (Default: 15)

Understanding the SFEDIT MASKS Process

The *excisefile* takes into consideration the overall length of the input soundfile. With paired *start* and *end* times on separate lines, it specifies the start and end of silences. These are inserted into the *infile* replacing what was there previously. The resultant soundfile is therefore the same length as the original input. It is convenient to be able to create several silences at once.

The *splice* parameter enables you to smooth the edges of these silences to varying degrees.

Musical Applications

Some ideas:

- create pulses of sound and silence
- design excise times for two different soundfiles such that they overlap or interlock exactly. Then you can mix (**SUBMIX MIX**) or interleave (**SUBMIX INTERLEAVE**) the two soundfiles so that the two sounds alternate. The latter merges mono files into a multichannel file, so the sound and silence of each input would end up on different channels.

ALSO SEE: **SFEDIT INSIL** above and **COMBINE INTERLEAVE** in the spectral dimension.

End of SFEDIT MASKS

SFEDIT NOISECUT – Suppress noise in a (mono) sound file, replacing with silence

Usage

sfedit noisecut *infile outfile splicelen noisfrq maxnoise mintone* [**-n**]

Parameters

infile – input soundfile
outfile – output soundfile containing either only the **non-noise** or the **noise** components (**-n** flag) of the original sound
splicelen – duration of splice slopes, in milliseconds
noisfrq – frequency above which the signal is regarded as noise (try 6000 Hz)
maxnoise – the **maximum** duration in milliseconds of any noise segments permitted to remain, i.e., NOT replaced. Range: 1000 to 22050ms
mintone – the **minimum** duration in milliseconds of any non-noise segments to be retained. Range: 0 to 50ms
-n – option to retain noise rather than non-noise

Understanding the SFEDIT NOISECUT Process

This process was developed after many attempts to automatically separate the noise constituents (sibilants etc.) from speech whilst trying to track the pitch of the other material. It uses a filter to recognise the presence of sibilants in the speech and allows vowels (and strongly pitched iteratives) to be separated – in place (i.e., remaining at their original time) – from the speech stream.

Alternatively, the sibilants (i.e., noise) can be similarly extracted, in place. This is the **-n** option.

Musical Applications

One way to apply this function is to use it to treat the pitched and unpitched components in a stream of events in different ways. You could first separate the pitched and noise elements (using this program in its two different senses). Then you might add tremolo to the pitched elements and after this reintroduce the unmodified noise elements by mixing them with the undulating tones.

End of SFEDFIT NOISECUT

PACKET – Isolate or generate a sound packet

Usage

packet packet 1 *insndfile outsndfile times mindur narrowing centring* [**-n | -f**] [**-s**]
packet packet 2 *insndfile outsndfile times dur narrowing centring* [**-n | -f**] [**-s**]

Example command line to create a packet:

```
    packet packet 1 in.wav out.wav 0.1 100 1 0
```

Modes

1 Found packet: looks for signal minima to determine the edges of the wave-packet
2 Forced packet: creates a packet at a specified time

Parameters

insndfile – input soundfile
outsndfile – output soundfile
times – a single time or a textfile of times at which the packet or packets is/are extracted or created
mindur – Mode **1**: the minimum duration in milliseconds of the (found) packet; it must be less than half the source duration, OR
dur – Mode **2**: the duration in milliseconds of the (forced) packet; it must be less than half the source duration
narrowing – narrows the packet envelope (Range: 0 to 1000):

- Values below 1.0 broaden the packet
- Values very close to zero may produce clicks (square wave envelope)
- Very high values with very short packets may produce click-impulses or silence

centring – centres the peak of the packet envelope. (If the packet content has varying levels, the true peak position may not correspond to the envelope peak position, unless the **-f** flag is used.)

- **0** – the peak is at the centre
- **-1** – the peak is at the start
- **1** – the peak is at the end

-n – normalise the packet level
-f – the packet wave maxima and minima are forced up or down to the packet contour. Default: the packet envelope is simply imposed on the existing signal. The **-n** normalisation flag is ignored if the **-f** flag is used.
-s – shave off a leading or trailing silence

Understanding the PACKET Process

In effect, PACKET cuts out a portion of soundfile, envelopes it, and places the amplitude peak at the beginning, centre or end of that envelope as specified by the user.

Musical Applications

This process streamlines the task of creating a usable snippet of soundfile. Trevor Wishart suggests that the main use of such snippets will be in the **TEXTURE** set of programs. Recall that these programs always begin each sound unit from the beginning of the input soundfile, with the option to then use the whole of the input soundfile for each iteration (the **-w** flag). PACKET makes it easier to prepare a specific short snippet that can then be proliferated into a texture.

End of PACKET

PARTITION – Partition a mono soundfile into disjunct files in blocks defined by groups of wavesets

Usage

partition partition 1 *infile outname outcnt groupcnt*
partition partition 2 *infile outname outcnt groupcnt* [**-r***rand*] [**-s***splice*]

Example command line to create separate files from a single input:

```
partition partition 1 insound outname 3 4
```

Modes

1 block durations are determined by number of **wavesets**
2 block durations are specified by the user

Parameters

insndfile – input soundfile to be partitioned (mono)
outname – generic root name for the output soundfiles. Numerals starting at 0 are appended to identify the different streams.
outcnt – the number of output soundfiles
groupcnt – Mode **1**: the number of wavesets per block, OR
dur – Mode **2**: the duration in seconds per block, where the duration of the blocks must be short enough to allow at least one block to be sent to every output file (i.e., minimum_dur = insndfilelen ÷ *outcnt*). If too large a *dur* is specified, *dur* will be reduced appropriately.
-r*rand* – the randomisation of durations (only) (Range: 0 to 1)
-s*splice* – the splice length in milliseconds (Default: 3mS)

Understanding the PARTITION Process

The purpose of PARTITION is to facilitate your sound design activities. The program cuts ALL of the sound in a source soundfile into disjunct pieces, assigning each of them in turn to *N* output soundfiles and producing *N* streams of disjunct segments. For example, if 8 pieces are cut and 2 output files are specified, each output soundfile will get 4 pieces.

The cut pieces, furthermore, are time-positioned in the output soundfiles *at the same times that they occurred in the source*. This is achieved by (automatically) inserting appropriate silences between the cut pieces in the output soundfiles.

The result of this process is that the different segment-streams can then be treated differently. Having done so, the resulting sounds can be remixed, synchronised at time zero, and all the original segments (now treated) will be returned to their original locations in the source. Implied here is that treatments which alter time-location within the soundfile should be avoided.

There are two modes.

- the blocks are defined by the number of **wavesets**. This means that the duration of the resulting blocks will be set by the length of those wavesets.
  .
- the blocks are defined by specifying a duration. This means that you have some control over the size of the blocks.

Suppose there is an input soundfile which contains a series of sounds-blocks (determined by wavesets – zero crossings). This series of *successive* blocks can be labelled "**abcdefghijklmno...**". We decide to create **3** output soundfiles (streams) with these sounds. **What happens is that every third block is assigned to a different stream** (the "-" represents a silence replacing a missing block):

- Outfile 1 contains: "a--d--g--j--m--"...
- Outfile 2 contains: "-b--e--h--k--n-"...
- Outfile 3 contains: "--c--f--i--l--o"...

ALSO SEE: **ISOLATE** for a detailed discussion of the similarities and differences between ISOLATE and PARTITION.
For the definition of a "waveset", or pseudo-wavecycle, see the **DISTORT group**.

End of PARTITION

PREFIX SILENCE – Add silence to the beginning of a soundfile

Usage

prefix silence *infile outfile dur*

Example command line to add silence:

```
    prefix silence flute flutedel 1.7
```

Parameters

infile – input soundfile
outfile – output soundfile with silence prefixed
dur – duration of silence to add

Understanding the PREFIX SILENCE Process

PREFIX SILENCE simply places the specified duration of silence at the beginning of the *infile*.

Musical Applications

This can be a straightforward way to create a gap between soundfiles, or time an entry. The utility might come in handy when running batch files.

ALSO SEE: SFEDIT INSIL and **SILEND**

End of PREFIX SILENCE

SFEDIT RANDCHUNKS – Cut chunks from a soundfile, randomly

Usage

sfedit randchunks *infile chunkcnt minchunk* [**-m***maxchunk*] [**-l**] [**-s**]

Parameters

infile – input soundfile

> **There is no outfile name.** The cut sections will be saved as new soundfiles, named *infile* truncated by one character, with a number added, starting from zero.

chunkcnt – the number of chunks to cut
minchunk – the minimum length of the chunks, in seconds
-m*maxchunk* – the maximum lengths of the chunks, in seconds
-l – chunks chosen are evenly distributed over the file (Default: random distribution)
-s – all chunks start at the beginning of the file

Understanding the SFEDIT RANDCHUNKS Process

SFEDIT RANDCHUNKS is like **SFEDIT RANDCUTS** but enables you to be more specific about the number of chunks (*chunkcnt*) and their length (*minchunk* and the optional *maxchunk*). Each chunk is saved as a new soundfile, with a name derived from the name of the *infile*.

Musical Applications

The number and length controls make it possible to make a controlled number of chunks of random length within a specified range. The ability to focus on the start of the soundfile enables you to explore the qualities of the attack portion of the sound.

End of SFEDIT RANDCHUNKS

SFEDIT RANDCUTS – Cut soundfile into pieces with cuts at random times

Usage

sfedit randcuts *infile average-chunklen scattering*

Parameters

infile – input soundfile

There is no outfile name. The the cut sections will be saved as new soundfiles, named *infile* with an underscore and a number added, starting from zero.
(The program usage may still say that a character is truncated, but this has not happened since an alteration was made in 2010.)

average-chunklen – the average length of the chunks to cut
scattering – controls the amount of variation in the length of the cuts (Range: 0 to 8)

Understanding the SFEDIT RANDCUTS Process

SFEDIT RANDCUTS provides a way to cut up a soundfile into several portions of a specified average length, saving each as a separate soundfile. The amount of difference in the lengths can be adjusted with the *scattering* parameter: the regularity of the lengths gets less and less as *scatter* increases.

Musical Applications

This could be a way of multiplying source material when a given soundfile has sufficient variation in its contents to justify the procedure.

ALSO SEE: SFEDIT RANDCHUNKS.

End of SFEDIT RANDCUTS

SFEDIT REPLACE – Insert a 2nd sound into an existing sound, replacing part of the original sound

Usage

sfedit replace *infile insert outfile time endtime* [**-w***splice*] [**-l***level*]

Parameters

infile – input soundfile
insert – 2nd soundfile to insert
outfile – completed output soundfile
time – the *time* at which the 2nd soundfile is to be inserted into the 1st soundfile
endtime – the *endtime* of the segment in the original soundfile to be replaced
-w*splice* – splice window in milliseconds. Default: 15 ms
-l*level* – gain multiplier on the inserted soundfile. Default: 1.0

Understanding the SFEDIT REPLACE Process

There is already a process "insert sound" (**SFEDIT INSERT**) which allows you to insert a 2nd sound into an existing sound, either by overwriting the original sound at the point of insertion, or cutting the original sound at the point of insertion, inserting the 2nd sound, and then continuing with the first sound from the place where it was cut.

This process allows you to overwrite a SPECIFIED SEGMENT of the original sound with the new sound, even where this is not the same length as the inserted sound. Note however, that the 2nd sound must be AT LEAST AS LONG as the gap created in the original sound .

Musical Applications

This function provides a overwrite facility.

End of SFEDIT REPLACE

RETIME – Rearrange and retime events within a soundfile

Usage

retime retime 1 *infile outfile refpoints tempo*
retime retime 2 *infile outfile resyncdata tempo peakwidth splicelen* (This mode is not available on the Command Line. It is available in *Sound Loom*.)
retime retime 3 *infile outfile minsil inpkwidth outpkwidth splicelen*
retime retime 4 *infile outfile tempo minsil pregain*
retime retime 5 *infile outfile factor minsil* [**-s***start* **-e***end* **-a***sync*]
retime retime 6 *infile outfile retempodata tempo offset minsil pregain*
retime retime 7 *infile outfile retempodata offset minsil pregain*
retime retime 8 *infile outfile tempo eventtime beats repeats minsil*
retime retime 9 *infile outfile maskdata minsil*
retime retime 10 *infile outfile minsil equalise* [**-m***meter* **-p***pregain*]
retime retime 11 *infile minsil*
retime retime 12 *infile outfile.txt*
retime retime 13 *infile outfile goalpeaktime*
retime retime 14 *infile outfile goalpeaktime peaktime*

On the Command Line, type **retime retime mode_number** to get the detailed Usage for a Mode.

Example command line to to adjust tempo (Mode 1) :

```
retime retime 1 infile outfile refpoints.txt 180
```

Modes

1 Specify the times of peaks in the input. Output these at a regular pulse in the given tempo
2 Synchronise specified peaks to specified times at a specified tempo (Also see **Mode 2** below. This mode is not available on the Command Line. It is available in *Sound Loom*.)
3 Shorten existing events in the input sound. This process shortens the events in the infile. It assumes these events are separated by silences, however short.
4 Find existing events in the input sound and output them at a regular tempo (**MM**). This process assumes that these events are separated by silences, however short.
5 Find events in the input sound and change their speed by a factor. This process assumes these events are separated by silences, however short.
6 Position events in the input sound at specified **beats** in the output. This process assumes these events are separated by silences, however short.
7 Position events in the input sound at specified **times** in the output. This process assumes these events are separated by silences, however short.
8 Specify an event within the input sound and repeat it at a specified tempo. This process assumes these events are separated by silences, however short.

9 Replace some events in the input sound by silence, using a specified pattern of 'deletions'. This process can be used to change the rhythmic pattern of the input events. It assumes these events are separated by silences, however short.
10 Adjust the levels of events in the input sound, changing the pattern of accents. This process can be used to change the accenting pattern of the input events. It assumes these events are separated by silences, however short.
11 Finds durations of shortest and longest **events** in the input sound (not silences). This process assumes these events are separated by silences, however short. Output is to the Console.

12 Find the start of the sound in the file (the 1st non-zero sample). The *starttime* of the event is written to a new, **or an existing** datafile. The latter allows several sounds to be run through the process to the same datafile, accumulating a list of the *starttimes* of each sound processed. – On the *Sound Loom* a list of sounds on the **Chosen Files** can be processed via the **Bulk Process** mechanism, to produce a single output datafile.
13 Find the peak in the input sound and move all the data so the peak is at a specified time. This process allows the peak within a soundfile to be placed at a specific time after the file start. To achieve this, the entire sound is moved by inserting or removing silence.
14 Specify an event in the input sound and move all the data so the event is at a specified time. This process allows the event you specify within the input soundfile to be placed at a specific time after the file start. To achieve this, the entire sound is moved by inserting or removing silence.

Parameters

infile – input soundfile
outfile – output soundfile
refpoints – (Mode **1**) times of peaks in the input sound file which will become on-the-beat events in the output
tempo – (Modes **1-2-4-6**) tempo of the output soundfile, as a **Metronome Mark (MM)** for values > 20, OR as a **beat duration** for values less than 1. This parameter can be used to change the tempo of the input sound from its original MM (specified in *resyncdata*) to the tempo defined here.
resyncdata – (Mode **2**) a textfile consisting of:

- The (approximate) MM of the input file sound
- The time of the **first accented event**. This must be one of the times in column 2, below.
- Two columns of timing data:
  - The 1st column represents the actual time of events in the input sound.
  - The 2nd column represents the required times of those events *if at the MM of the **input** sound* (specified earlier).
  - Times in both columns must increase.

  Note that, in the *Sound Loom*, this data can be automatically generated from a **property file** in which the ***rcode*** property has been specified for a sound. (The *rcode* property allows you to map the idealised rhythm of a musical phrase).

peakwidth – (Mode **2**) The duration (in milliseconds) of the events in the output sound. This parameter can vary over time, using a *time peakwidth* breakpoint file.
splicelen – (Modes **2-3**) the duration of splices (in milliseconds) which cut the events to be placed in the output sound

minsil – (Modes **3** through **11**) duration in milliseconds of the minimum silence between events. (Range: 0.045351 to 10000.0). This should be large enough to avoid fleeting silences *within* events being noticed by the process. If too large, the event duration shown may be that of the entire soundfile. Listening to the sound or viewing it in an editor may help to indicate the right value for a given sound if the displayed results don't seem reasonable.
inpkwidth – (Mode **3**) (minimum) width (in milliseconds) of events in the **input** file.
outpkwidth – (Mode **3**) required width (in milliseconds) of events in the **output** file.
pregain – (Mode **4-6-7** and **10**) gain on the input signal (Range > 0 to 1). *Pregain* may need to be set less than one if events in the output overlap one another. Note that **-p***pregain* is optional in Mode **10**.
factor – (Mode **5**) speed-change factor (which can vary over time).
-s*start* – (Mode **5**) time at which the speed changing becomes effective
-e*end* – (Mode **5**) time at which the speed changing ceases to be effective

> *Start* and *end* allow you to (for example) shorten or lengthen specific events within the sound.

-a*sync* – (Mode **2**) approximate time of any *infile*-event which synchronises with its copy in the output. This parameter can be used to generate an output sound which will synchronise in a particular way with the input sound when the two are mixed together. (Also see below.)
retempodata – (Modes **6-7**) textfile contains positions of events in the output.

- Mode **6:** Events are counted **in beats** at the defined **Tempo**, and are **assumed to start at beat zero**.
- Mode **7:** Events positions are given **as times in seconds**, and are **assumed to start at time zero**.
- The *offset* parameter can be used to make the event sequence start at a later time.

offset – (Modes **6-7**) time of the first sounding event in the output file
eventtime – (Mode **8**) start time (roughly) of event to repeat (time specified must be **inside** the event).
beats (Mode **8**) number of beats (at the specified tempo) within the event-to-repeat – effectively the duration of the event-to-repeat.
repeats – (Mode **8**) number of times to repeat the event (1 or more)
maskdata – (Mode **9**) A textfile with a sequence of **zeros** and **ones**:

- **0** means masks an event (replace it by silence).
- **1** leaves an event unmasked.
- The pattern of masking is repeated once its end is reached.

equalise – (Mode **10**) Range 0 - 1.

> If the **meter** is set to zero
>
> - *equalise* = **0**: has no effect
> - *equalise* = **1**: all events are forced to equal loudness.
> - Intermediate values have intermediate effects.
>
> If the **meter** is set to a non-zero value
>
> - the value of *equalise* determines the level balance between accented and unaccented beats.

-m*meter* – (Mode **10**) specifies the pattern of accented beats

- *meter* = **0**, no events emphasized:
  - The relative loudness of the events is adjusted by the value of **equalise**.
  - The loudest event remains at its original level.
  - Other events are boosted towards this maximum. First, the difference between the amplitude of this event and the maximum is calculated. If the event is 0.2 of the maximum, this difference is 0.8. With an equalisation level of 0.5, the event is boosted half way (0.5) towards the maximum: i.e., in this example, by half of 0.8 (0.4), bringing it to 0.6 of the maximum.
- *meter* = **any non-zero integer (*N*)**, e.g. 3:
  - All events are first forced to the level of the loudest.
  - Then every N^{th} event remains at that maximum level, and all the others are reduced to **equalise** times the maximum level.

outfile.txt – output datafile for Mode **12**

- The output datafile **must have '.txt' extension**.
- If the file already exists, the output is **appended** to that file.

goalpeaktime – (Modes **13-14**) Time to which the peak found in the input file is to be moved

- Note that, if the peak is moved to an earlier time, and this causes the initial sound in the file to be placed 'before zero', the process will fail.
- Moving the peak **backwards** in time should only be attempted if there is sufficient silence at the start of the sound.

peaktime – (Mode **14**) Time of the event to be moved, within the input soundfile

- Note that, if the *goalpeaktime* is **before** the *peaktime*, the event will be moved backwards in time.
- If this causes the inital sound in the input file to be placed 'before zero',the process will fail.
- Moving the peak **backwards** in time should only be attempted if there is sufficient silence at the start of the sound.
- Modes **13** and **14** allow **specific events in two (or more) different files** to be **synchronised**, simply by aligning the start-times of the files themselves.

Understanding the RETIME Process

RETIME is really a whole suite of programs for retiming the events within a sound (file): to some pre-specified set of times, to a tempo, or to a pattern etc. Some of these processes only apply to sounds which contain events **separated by silence**, while others can be applied to any sound (for which you must specify the location of its peaks).

NB: These processes work by editing, rather than time-stretching. They insert small silences or, conversely, cut out small segments from the original source. They are therefore probably more appropriate for small-detail adjustments to material (especially if the time between events is being shortened) than for grand redesigns.

None of these processes alter the (consecutive) order of the events in the source.

A **metronome mark** indicates the number of beats per minute in a musical phrase, e.g.,

- MM 60 : 60 beats per minute, 1 beat every second.
- MM 180 : 180 beats per minute, 3 beats every second.
- MM 30 : 30 beats per minute, 1 beat every 2 seconds.

The maths: 60 ÷ MM = duration. E.g., 60 ÷ 120 = 0.5, and 1 (second) ÷ 0.5 = 2 beats per second.

The sound input to RETIME must have definable peaks. Some modes of this process only work with events (within the soundfile) separated by silences.

- Modes **1** and **2** of RETIME work on any soundfile. In these processes you are asked to mark the peaks in the source.
- Mode **1** takes the peak events that you have marked and places them at regular time-intervals at a **metronome mark** (MM) that you specify.
- Mode **2** takes the peak events that you have marked and places them at new times which you specify.

These processes works by directly editing the source, and the repositioned events in the output are separated by (usually tiny) silences. For this reason the output may sound slightly 'clipped' (in the sense used about speech – *not* digitally distorted) relative to the input.

These processes may also be used to generate output sounds appropriate for use in Modes **3** to **11**.

- Modes **3** to **11** allow you to change the tempo, rhythm or accentuation pattern of the events in the input sound.
- They assume that the events in the input are *separated by silences*, however short.
- Mode **11** provides information about the duration of (silence-separated) events in the sound. This is a typical display (*minsil* was 0.1):

```
INFO: Counting silences between events.
INFO: Marking silence-separated Events.
INFO:
INFO: Shortest event = 0.000023 secs :: = 0.022676 mS
INFO: Longest event = 0.960045 secs
```

- Modes **12** & **13** allow an event within the input soundfile to be positioned at a particular time relative to the start of the file.

The process works by inserting or removing silence at the start of the file. Modifying the soundfile in this way ensures that, when several of these soundfiles are used together (e.g., in a mix), specific events in the files can be rhythmically aligned, by aligning the start times of the soundfiles themselves.

Mode 2:

- The process cuts the input sound at each specified peak in the source and reassembles the pieces at the specified output times.
- This process can be applied to any sound-data (with definable peaks).

- It generates a file of silence-separated peaks (*unless the output events overlap one another*).
- The process can be used to force an event series with an approximate or indefinite rhythm and tempo to lie on a precisly defined rhythm in some (possibly different) tempo.

"SYNC" what's this about?

For rhythmic work, you may well want a particular event in the time-modified sound to occur at a specific time relative to the original. For example, say the original event is a 2 quaver anacrusis to a strong beat at 0.65 seconds. If you want the strong beat to be at the same TIME in the output event, you can use the "SYNC" option to specify where this beat is. Note that NOT all sync-specs are possible. Particularly, if you timestretch the entire file (using a factor greater than 1), then events in the output cannot be made to sync with events in the input (except at time zero).

Musical Applications

The overall application of RETIME is to (re)rhythmicise material. This suite of processes can be used to change the time-pattern within an input sound, such as the rhythm of a melodic phrase or the prosody of speech. They were originally developed to slightly shift the true rhythm of spoken phrases of natural speech onto an idealised rhythmic frame, permitting different vocal phrases to be rhythmically locked to a particular tempo, without radically altering the speech prosody.

End of RETIME

SILEND – Add silence to the end of a soundfile

Usage

silend silend 1 *insndfile outsndfile sildur* **silend silend 2** *insndfile outsndfile outdur*

Example command line to add silence at the end:

```
silend silend 1 insnd.wav outsnd.wav 3.5
```

Modes

1 specify duration of the silence
2 specify total output duration

Parameters

insndfile – input soundfile
outsndfile – output soundfile with silence at ende
sildur – duration of silence to add
outdur – total duration of output sound after the silence is added

Understanding the SILEND Process

Other CDP programs can add silence at the start (**PREFIX SILENCE**) or in the middle (**SFEDIT INSIL**) of a soundfile, but not at the end. SILEND completes the facilities by adding silence at the end of the soundfile.

Musical Applications

One application is to add silence at the end of each soundfile when **assembling** a list of soundfiles to be played as one sequence. This puts pauses between the sounds, pauses that are more appropriately placed at the end rather than the beginning. Using the blockedit facilities in *Sound Loom* or a batch file will do the job quickly.

ALSO SEE: PREFIX SILENCE and **SFEDIT INSIL**

End of SILEND

SFEDIT SPHINX – Switch between several files, with different switch times, to make a new sound

Usage

sfedit sphinx 1 *infile1 infile2 ... outfile switch-times splicelen* [**-w***weight*] [**-r**]
sfedit sphinx 2-3 *infile1 infile2 ... outfile switch-times splicelen segcnt* [**-w***weight*] [**-r**]

Example command line to create a soundfile from several inputs:

```
sfedit sphinx 1 infile1 infile2 switchtimes.txt 12
sfedit sphinx 2 infile1 infile2 switchtimes.txt 12 48
sfedit sphinx 3 infile1 infile2 switchtimes.txt 12 48 -r
```

SYNOPSIS
The switch times are in rows and columns. The rows have sets of times and the columns are like parallel tracks. The program moves down through the rows, selecting one time from one of the rows at each row position. The times in each column can be independent, but with no time gaps within a column. The columns are read in order, from left to right, unless the **-r** (*randomisation*) flag is set. (Imagine all files are running in parallel on a multitrack. Switch from one 'track' to another at switch times, where the Nth switch-time in one file corresponds to the Nth in another file, *but these are not necessarily the same absolute time*.) See our example file.

Modes

1 In Sequence – Move down the rows in sequence, selecting one time from each column.
2 Permutated – Randomly permutate the order of the rows of times, selecting one time from each column.
3 Random Choice – Select any row at random, and a time from each column.

Parameters

infile1 infile2 ... – 2 or more input soundfiles
outfile – resulting output soundfile
switch-times – text file containing the times in seconds at which the input sound(s) are divided into segments

- Include time zero if you want to use a segment at the start of the file
- The 1st time values *for each file* are listed on the 1st line of the file
- The 2nd time values *for each file* are listed on the 2nd line.
- You need one more set of times than number of segments, because the last set of times forms the end-times for the start-times in the penultimate row.

- **The program cycles through the rows and times (in various ways) until it comes to the end of either the (shortest) input soundfile or the number of segments defined.** There is a soundfile for each column of data, and in all Modes you will hear the segments cycling repeatedly through the soundfiles in order: e.g., 1-2-3, 1-2-3, etc., unless **-r** is set. (See **example file** below.)
splicelen – the duration of the splices, in milliseconds (Range: 2 to 15 milliseconds)
segcnt – Modes **2-3** only: the number of segments to use in the output
-w*weight* – when set, *infile1* occurs *weight* times more often than the other infiles
-r – when set, the order of files used is randomly permutated, otherwise the files retain the order in which they were invoked. In other words, it randomises from which column the time is taken.

Understanding the SFEDIT SPHINX Process

Imagine that we are using 3 input soundfiles (A, B,& C) with SPHINX, and we cut 4 segments from each soundfile.

We need to provide information on how these soundfiles are to be segmented. For each soundfile we must provide a sequence of times at which the soundfile will be cut, i.e., a column of times in the *switch-times* data file for each soundfile.

For soundfile **A**, we can call these 4 segments **a1**, **a2**, **a3** and **a4**, and
for soundfile **B**, we can call these 4 segments **b1**, **b2**, **b3** and **b4**
and so on.

To make these 4 segments we will need to specify 5 times. Thus the times for soundfile **A** might be:

0.1 0.35 1.0 7.0 7.1

and these will cut the input soundfile into these 4 segments (and only these – the output will play these 4 segments and then stop):

segment 1: 0.1 to 0.35, duration 0.35 seconds
segment 2: 0.35 to 1.0, duration 0.65 seconds
segment 3: 1.0 to 7.0, duration 6.00 seconds
segment 4: 7.0 to 7.1, duration 0.1 seconds

NB: Segments are cut from the soundfile **sequentially** (end-to-end) starting at the first time (in a given column) and ending at the last. *You therefore cannot skip over any part of the input soundfile – i.e., leave time-gaps – apart from:*

- the start of the file (by making the first time greater than 0.0)
- the end of the file (by making the last time less than the file's duration)

The information on how to segment each file is provided in the *switch-times* text datafile containing ***columns of time values***:

- Each column in the data corresponds to one of the input soundfiles.
- ***The same number of segments must be cut from each file*** – this implies that each column of the datafile must have the same number of rows.
- The segments in the different files ***do not have to correspond in length*** – **a1** does not need to be the same length as **b1** or **c1**.
- The segments ***do not have to be at the same times in the 3 input soundfiles*** – the start times (or end times) in each column do not need to be the same – but there are no time gaps between segments.

For example, suppose we have a datafile of 5 times (schematically):

A*time1* **B***time1* **C***time1*
A*time2* **B***time2* **C***time2*
A*time3* **B***time3* **C***time3*
A*time4* **B***time4* **C***time4*
A*time5* **B***time5* **C***time5*

Producing 4 segments for each column:

a1 b1 c1
a2 b2 c2
a3 b3 c3
a4 b4 c4

Here is a typical *switch-times* file: 3 columns, one for each of 3 input soundfiles, **A**, **B** & **C**:

```
  A    B    C
 2.0  4.5  7.0
 4.0  5.0  7.2
 6.0  5.5  7.4
 8.0  6.0  7.6
10.0  6.5  7.8
```

Here we have 15 times and 12 *possible* segments (4 from each of 3 soundfiles), which are:

- **A1:** 2.0 -> 4.0 **B1:**4.5 -> 5.0 **C1:** 7.0 -> 7.2
- **A2:** 4.0 -> 6.0 **B2:**5.0 -> 5.5 **C2:** 7.2 -> 7.4
- **A3:** 6.0 -> 8.0 **B3:**5.5 -> 6.0 **C3:** 7.4 -> 7.6
- **A4:** 8.0 -> 10.0 **B4:**6.0 -> 6.5 **C4:** 7.6 -> 7.8

But note that still there are only 4 segments defined. In the basic operation of SPHINX in Mode **1**, the program will select the first segment from row 1, column 1 (A1), the second from row 2, column 2 (B2), the third from row 3, column 3 – now there are no more columns so it goes back to column 1 (in row 4) for the fourth segment (A4), and it continues to cycle round in this way. In this set of times, the length of segment that will come from each input sound is controlled by having progressively shorter lengths for soundfiles 2 & 3. The next section discusses how SPHINX reorders these segments in various ways.

Mode 1: strict sequential order

In Mode **1**, the program choses each of the 3 input **soundfiles** *in turn*, and selects a time from each **row** of segments *in turn*. In this case, the output will therefore be (with the order shown in red):

```
;this file will produce 12 segments
0.0     1.0     3.0
0.6     1.4     3.5
1.2     1.8     4.0
1.8     2.2     4.5
2.4     2.6     5.0
3.0     3.0     5.5
3.6     3.4     6.0
4.2     3.8     6.3
4.8     4.2     6.6
```

```
5.4       4.6       6.9
6.0       5.0       7.2
6.6       5.4       7.5
7.2       5.8       7.8
```

Put together into one line, the order of segments in the output soundfile will be like this:

```
Soundfiles: 1    2    3  / 1    2    3  / 1    2    3  / 1     2     3
Segments:   A1 - B2 - C3 / A4 - B5 - C6 / A7 - B8 - C9 / A10 - B11 - C12
```

In this example, we should hear a repeating sequence of long (sound **A**), medium-long (sound **B**), and short (sound **C**). This gives an indication of how the relationship of times and soundfiles can be constructed to achieve specific effects. Otherwise, more varied times can be used, but the cycling through the input soundfiles in order will remain the same unless the **-r** flag is set.

Note that soundfiles and columns of data always match up:

- At the 3rd entry (**C3**) we have used all the files, so the program goes back to the first soundfile (**A**) for the next segment (**A4**).
- It proceeds across the columns and down the rows until it reaches the end of the shortest soundfile or runs out of segments.

Mode 2: randomly permutated row-order

In **Mode 2** the order of rows is set by a random *permutation*. The program then selects a time from each row and produces a segment accordingly, until all rows have been used. Then a new row-order is chosen, etc. All the rows are used once before a new permutation of the row-order begins.

Note that (unless the **-r** flag is set) the ***input soundfiles*** are still chosen strictly in order.

Mode 3: entirely randomised row-order

In Mode **3** the rows are selected *entirely at random* – i.e., not even a reordering by permutation. The program thus selects its time from any row and produces the corresponding segment.

The input soundfiles are still chosen in strict order (the order in which the user supplied them) unless the **-r** flag is set. In this case, the time in each row (and therefore the order of the input soundfiles) is randomly permutated. This reordering of the soundfiles repeats each time all the input soundfiles have been used, similar to the way the rows are changed round in Mode **2**.

Musical Applications

SPHINX provides a way to mix up the contents of several soundfiles in a semi-controlled way. An example from Gustav Ciamaga provides a useful illustration. He has determined the start-time of each syllable in soundfiles containing spoken text. He then used SPHINX to rearrange these syllables in various ways, maintaining their integrity. If segment start-times were to be chosen at random, then the input material would be divided up with no regard for internal shapes, perhaps picking out silences. The effect of this would vary depending on the nature of the sonic material.

Note that the modes and options can be thought of as progressive degrees of randomisation:

- In Mode **1** both soundfiles and times remain in **sequential** order – though the *weight* flag (**-w**) could be used to make the first soundfile predominate, and the *soundfile-randomisation* flag (**-r**) could be used to mix up the order of the soundfiles. For the purposes of this list of progressive randomisations, let us say that it the latter flag is saved until last.
- Mode **2** takes the randomisation process a step further by **randomly permutating** the order of the rows on each pass through the rows. We are keeping the order of the soundfiles the same, although the option not to do so remains available.
- Mode **3** makes selection of rows **entirely random**.
- We can intensify the randomisation process by not using the *weight* flag (**-w**) and by invoking the *soundfile-randomisation* flag (**-r**).
- Segment start-times chosen by the user completely at random mix up the material of the input soundfiles in an even more unpredictable manner.
- Finally, it should be mentioned that smaller segments will mix up the sounds to a greater degree than larger segments.

Gustav Ciamga has also provided this advice. "Creating lists of switch points can be facilitated with the function **Gated Onsets** (*Soundshaper* INFO > (SOUND) FILES > Gated Onsets), with *gate_level* .035 and *min_length* .05, and all other parameters set at zero. (The program being used here is HOUSEKEEP EXTRACT, Mode **6** [AE].) I prefer this procedure as it guarantees that switch points will coincide with the onsets or attack-points in a sound file. Mind you, arbitrary lists of switch points can provide surprising outcomes.

"*Soundshaper* users will find the TWIXT/SPHINX functions integrated on one parameter page, i.e., **Edit/Mix > Edit > Switch**. The first, third and fifth modes found on this parameter page are **TWIXT** routines; the second, fourth and sixth modes are SPHINX routines."

I did this gating with *count.wav* (female voice counting 1 to 10) and the following file was produced (*countonsets.txt*):

```
0.052245
0.899773
1.828571
2.612245
3.517823
4.376961
4.812336
5.282540
5.654059
6.089433
6.489977
6.896327
7.598730
```

Then I ran **Edit/Mix > Switch** in *Soundshaper*, using the first mode listed: **Single-switch Sequence**. *Count.wav* was the first sound, and *frogs3cdt.wav* was the second sounds (chirping frogs), and *countonsets.txt* (as above) was the *timesfile*. The result was a fairly tidy alternation between the counting voice and the frog's chirps, showing that the switch was taking place pretty much at the onsets of the counts. (AE)

In *Sound Loom*, the TWIXT and SPHINX functions are under the EDIT button. Please note:

- TWIXT is listed as **switch between files**
- SPHINX is listed as **make a sphinx**
- 2 or more input soundfiles need to be in the left panel (CHOSEN FILES).
- Do **not** put the *timesfile* onto the left panel (CHOSEN FILES) – it should be on the WORKSPACE.
- You **GET** the *timesfile* when you are on the parameters dialogue page for either function.

ALSO SEE: SFEDIT TWIXT.

End of SFEDIT SPHINX

SUBTRACT – Subtract one file from another

Usage

subtract subtract *infile1 infile2 outfile* [**-c***chans*]

Example command line to perform a soundfile subtraction:

```
subtract subtract inf1 inf2 outf -c3
```

Parameters

insndfile1 – mono or 2 or more channel input soundfile from which to subtract another soundfile (which must be mono)
insndfile2 – input mono-only soundfile to subtract from *insndfile1*
outsndfile – resultant mono or multi-channel soundfile
-c*chan* – which channel of a multi-channel *insndfile1* (includes a stereo soundfile) to use. If *insndfile1* is mono, you will need to put `-c1` – the parameter is not optional.

Understanding the SUBTRACT Process

This program simply subtracts *infile2* (which must be mono) from *infile1*, which may be multi-channel. The *chan* parameter enables you to specify which channel of an input multi-channel soundfile to use. The output soundfile will be mono or multi-channel depending on the inputs.

End of SUBTRACT

SFEDIT SYLLABLES – Separate out vocal syllables

Usage

sfedit syllables mode *infile outfile cuttimes dovetail splicelen* [**-p**]

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel-count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input soundfile
outfile – generic name of output soundfiles; the filenames are formed by adding numbers starting at '1' to the generic name
cuttimes – text file containing the *start end* time pairs for each syllable to be cut
dovetail – the time in milliseconds to allow for syllable overlap. Range: 1 to 20 ms
splicelen – the duration of the splice window in milliseconds (cannot be shorter than the time between any two times)
-p – forces the process to cut PAIRS of syllables

Understanding the SFEDIT SYLLABLES Process

The syllables in speech are difficult to separate one from the other simply by editing. By their very nature, the sounds of speech flow naturally one into another, and there is no 'natural' cutting point between them. This process compensates for this problem by shaving a little bit from the end of the previous syllable and a little bit from the start of the following syllable, thereby, for every syllable, giving separated syllables that are more convincing.

The *cuttimes* file needs to be carefully constructed by noting in an appropriate sound editor the *start* and *end* times of the syllables you want to extract . They are given with each *start end* time pair given on separate lines:

```
0.0058 0.215
0.319  0.720
1.01   1.56
1.72   2.00
2.1    2.467
2.7    3.04
```

You could then excise the syllables right away, but the advantage of using SFEDIT SYLLABLES is ability to allow for syllable overlap, thus capture endings and beginnings of syllables that overlap and would otherwise be lost.

Graphic sound editors should work for this purpose, such as *Audition*, *Sound Forge* or the new graphic display and editing facilities on the *Sound Loom*, as you can block out and hear a portion of soundfile. In *Sound Loom*, you ALT Mouse Click on a soundfile on the Workspace to access these facilities.

Using ears only, *Soundshaper's* 'Play using Markers' facility (i.e., Play FROM .. TO) is also a straightforward way to find the edit points, and gives a time display accurate to three decimal places.

Gaps between times need to be large enough to accommodate the *dovetail* (overlap) and the *splice* (join slope). The maximum *dovetail* is 20 ms, i.e., 0.02 seconds, so this time distance should be regarded as the minimum. This applies to the length of the syllables as well as the time between syllables. Syllables that are too short, it should be added, will not be useful as soundfiles because the sound will disappear in the *splice*. As a rule of thumb, regard 200 ms as the shortest practical length of a syllable.

A series of soundfiles using the generic name as the base are created. If the generic name is 'speechsyl', the separate soundfiles containing each syllable will be 'speechsy1', 'speechsy2' etc. The soundfile extension is appended by the program, as usual. A variant of SFEDIT CUTMANY, all the soundfiles are created in one pass.

Musical Applications

SFEDIT SYLLABLES can be used to separate

- the syllables of speech
- the individual note events from a melody performed on an instrument with strong transitional characteristics where it passes from one note to another

End of SFEDIT SYLLABLES

SFEDIT TWIXT – Switch between several files, to make a new sound

Usage

sfedit twixt 1 *infile1 infile2 ... outfile switch-times splicelen* [**-w***weight*] [**-r**]
sfedit twixt 2-3 *infile1 infile2 ... outfile switch-times splicelen segcnt* [**-w***weight*] [**-r**]
sfedit twixt 4 *infile1 infile2 ... outfile switch-times splicelen*

Example command line to create a soundfile from several inputs:

```
sfedit twixt 1 infile1 infile2 switchtimes.txt 12
sfedit twixt 2 infile1 infile2 switchtimes.txt 12 23 -r
sfedit twixt 4 infile1 infile2 switchtimes.txt 12
```

SYNOPSIS
The switch times in TWIXT are in strict ascending order, whether written in a row (times separated by 'white space'), in a column or in several columns. If several columns, the times are read from left to right in each row and must constantly ascend. A given segmentation is applied to any of the input soundfiles, depending on the parameter settings. The **-r** (*randomisation*) flag mixes up the soundfiles rather than just cycling round the inputs. The Modes change how the times are read. See our example file.

Modes

1 In Sequence – Imagine all soundfiles are running in parallel on a multitrack. Switch from one sound to another at switch times, where the Nth switch-time in one file corresponds to the Nth in another file, *and the times are the same on all the tracks*.
2 Permuted – Similar to Mode **1** but the time-segment order is randomly permutated.
3 Random Choice – Similar to Mode **1** but choose any time-segment at random as the next segment.
2 Edit only – Cut *infile1* (only) to chunks defined by switch-times, and output the chunks as separate soundfiles. Note that at least two infiles need to be listed as input soundfiles for this mode to work, even though the cuts are only made on the first of the listed soundfiles.

Parameters

infile1 infile2 ... – two or more input soundfiles
outfile – resulting output soundfile; Mode **4** outputs several numbered output soundfiles.
switch-times – text file containing a single column of **ascending times** in seconds at which the output soundfile switches between the input soundfile(s):

- Include time zero if you want to use a segment at the start of the file
- The 1st time values *for each file* are listed on the 1st line of the file
- The 2nd time values *for each file* are listed on the 2nd line.

- You need one more time (or row of times) than number of segments, because the last time (in each column) forms the end-time for the previous start-time.
  splicelen – the duration of the splices, in milliseconds
  segcnt – Modes **2-3** only: the number of segments to use in the output
  -w*weight* – when set, infile1 occurs *weight* times more often than the other infiles
  -r – when set, the order of files used is randomly permutated

Understanding the SFEDIT TWIXT Process

Imagine we use the same 3 input soundfiles (A, B, C) cut into 4 segments, as described in the notes about **SPHINX**.

With TWIXT, input soundfiles **A**, **B** and **C** are cut into segments (**a1**, **a2**, **a3**, **a34**.....**b1**, **b2**, **b3**, **b4**.....**c1**, **c2**, **c3**, **c4**), but corresponding segments in the 3 files are

- at the **same time**
- of the **same length**

i.e., (if you put the data in columns):

- **a1** = **b1** = **c1** ..... **a2** = **b2** = **c2** etc.
- **A***cut1* = **B***cut1* = **C** *cut1*
  A*cut2* = **B***cut2* = **C***cut2*
  etc.

Because of this, we don�t in fact need to specify separate cutpoints for each of the 3 files. One set of cutpoints specifies what happens in EVERY file, so the data can just as well be written in a single line or column. In the following *switch-times* file, the times:

```
0.1
0.3
1.0
7.0
7.1
```

cut ***every*** input soundfile into the 4 segments:

> **segment 1**: 0.1 to 0.35, duration 0.35 seconds
> **segment 2**: 0.35 to 1.0, duration 0.65 seconds
> **segment 3**: 1.0 to 7.0, duration 6.00 seconds
> **segment 4**: 7.0 to 7.1, duration 0.1 seconds

Twixt then proceeds exactly as SPHINX, reading through the times (in various ways) and cutting out the given segment from one of the soundfiles (except for Mode **4** which adds an 'Edit' option). The program cycles round the times in their input order, unless the use of the **-r** flag randomises the soundfile order.

It is possible to enter the *switch-times* for TWIXT in columns, one column for each soundfile. You might want to do this if you wanted to plan out how each soundfile was to be used. However, note that TWIXT reads the data differently than SPHINX. With TWIXT, the data is read from left to right, row by row, **and therefore all times must be in ascending order, row by row**. We could re-work the *switch-times* for a SPHINX example to be in ascending order:

```
2.0 3.0 3.5 3.7 4.7 5.2 5.4 6.4 6.9 7.1 7.5 7.8 8.0
```

In columns (note comment later about connecting these columns with the soundfiles – can be confusing because the segments are *between* the times):

```
2.0 3.0 3.5
3.7 4.7 5.2
5.4 6.4 6.9
7.1 7.5 7.8
8.0
```

Note the 13th time to provide an end point for the final (12th) segment. The order of the output sequence, cyling round the soundfiles in order, will be like this:

Order of Segments in a TWIXT Output Soundfile

[1] A1 (2.0 -> 3.0)	[2] B1 (3.0 -> 3.5)	[3] C1 (3.5 -> 3.7)
[4] A2 (3.7 -> 4.7)	[5] B2 (4.7 -> 5.2)	[6] C2 (5.2 -> 5.4)
[7] A3 (5.4 -> 6.4)	[8] B3 (6.4 -> 6.9)	[9] C3 (6.9 -> 7.1)
[10] A4 (7.1 -> 7.5)	[11] B4 (7.5 -> 7.8)	[12] C4 (7.8 -> 8.0)

Note that there are no gaps in the times. We still get a repeating long, medium-long, short sequence cycling through the input soundfiles, but the movement through each soundfile will be forward, moving from the beginning (or near the beginning) to the end (or near the end).

The following single column *switch-times* file uses the use of the equal times with three input soundfiles, all of which are at least 9 seconds long. This is designed to help you hear the cycling around the soundfiles. Note that there are 9 segments and 10 times. Also note that comments are used in the text file to help clarify where the segments are. Comments are preceded by a semi-colon and go *after* the data if on the same line.

```
0.0     ;sounds1-2-3, set1 (0-1, 1-2, 2-3)
1.0
2.0
3.0     ;sounds1-2-3, set2 (3-4, 4-5, 5-6)
4.0
5.0
6.0     ;sounds1-2-3, set3 (6-7, 7-8, 8-9)
7.0
8.0
9.0     ;time at which to end the last segment
```

I find it easier to use this type of *switch-times* file format (rather than columns) with TWIXT, because it is easier to keep clear where the segments are in relationship to the input soundfiles if you are creating some kind of duration pattern. But remember that the **-r** (*randomisation*) flag will mix up the order of the soundfiles. [AE]

Musical Applications

The applications for TWIXT are the same as for **SPHINX**, but with the difference that the same times and lengths apply to all the input soundfiles – only the soundfile to which they apply changes (cycles round the inputs in various ways by changing the order of the times), and the soundfile order itself can be randomised.

Note that while **HOUSEKEEP EXTRACT** Mode **6** results in a list of onset times, as noted above, SFEDIT TWIXT Mode **4** is one way to use these times to create separate soundfiles that start with these times and end with the next time in the list – or the end of the input soundfile. On this point, note the advice provided by Gustav Ciamaga.

End of SFEDIT TWIXT

SFEDIT ZCUT – Cut out and keep a segment of a MONO soundfile, cutting at zero crossings (no splices)

Usage

sfedit zcut mode *infile outfile start end*

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel-count)
3 Time as grouped sample count (e.g., 3 = 3 stereo pairs)

Parameters

infile – input MONO soundfile
outfile – cut segment saved as a new soundfile
start – (approximate) time in the *infile* where the segment to keep begins
end – (approximate) time in the *infile* where the segment to keep ends

Understanding the SFEDIT ZCUT Process

This process uses an alternative method to splice the sound segment, cutting it at the nearest zero-crossings in the signal, rather than making a splice. Nevertheless the cut should be clickless.

Musical Applications

This is a different way of cutting out a segment of sound. The start and end times you give are approximate because the nearest zero points will probably not be precisely at those times.

End of SFEDIT ZCUT

SFEDIT ZCUTS – Cut out and keep segments of a MONO soundfile, cutting at zero crossings (no splices)

Usage

sfedit zcuts mode *infile outfile_generic_name cutttimes*

Modes

1 Time in seconds
2 Time as sample count (rounded to multiples of channel-count)

Parameters

infile – input MONO soundfile
outfile_generic_name – generic name for cut segments saved as new soundfiles (the numbers '1', '2' etc. are added to this generic name to name the various output soundfiles)
cuttimes – a textfile of time-pairs for the *start* and *end* of each segment to cut

Understanding the SFEDIT ZCUTS Process

This process uses an alternative method to 'splice' the sound segments, cutting them at the nearest zero-crossings in the signal, rather than making a splice slope of default or specified duration . Nevertheless the cuts should be clickless.

Musical Applications

This is a different way of cutting out a segment of sound. The *start* and *end* times you give are approximate because the nearest zero points will probably not be precisely at those times.

The *cuttimes* data file enables you to specify the *start* and *end* times for several cuts. As many new soundfiles are made as there are data pairs in the file, with a number added to your generic name to create the output filenames. For example, if your generic name is **pop**, the various cuts will be named **pop1**, **pop2** etc.).

End of SFEDIT ZCUTS

ON RETIMING – An Overview of Rhythm Facilities

Overview of CDP programs that affect rhythm and timing

- **PREFIX SILENCE** and **MANYSIL** complement existing silence insertion processes **INSIL** and **SILEND**.
- **ENVNU EXPDECAY** produces a technically exact exponential decay to zero.
- **PEAKFIND** generates a list of the times of the peaks in a source sound.
- **CONSTRICT** shortens a soundfile by shortening any silences it finds in the source.
- **GRAINEX EXTEND** is a method for timestretching iterative sounds, such as a rolled-"rr" sound.

Review of the facilities in RETIME

- The existing rhythm programs in the CDP program set, **EXTEND SEQUENCE** and **EXTEND SEQUENCE2** have been complemented by a new suite of processes, collectively known as **RETIME**, which has 14 Modes.
- The sequence programs can be used to organise individual soundfiles as events in a rhythmic pattern.
- In contrast RETIME can be used to rearrange events **within** a soundfile, and to synchronise particular events in one soundfile with particular events in another.
- On the *Sound Loom* all these rhythmic processes have now been placed on a new **RHYTHM** menu, on the Process Page.
- In *Soundshaper*, the **RETIME** processes are found in the menu **SOUNDFILES > RHYTHM**, along with **MANYSIL**, **CONSTRICT** and **ENVNU PEAKCHOP** [R.F.].

End of RETIMING OVERVIEW

Last Updated 1 Apr 2017
Documentation: Archer Endrich, revised R. Fraser