**Easy Concept**

Easy is intended to be a script-driven way to build specialized audio signals. While this is general concept is similar to some other utilities, it emphasizes building an audio signal based on a timeline.

What is good for:

* Building volume and frequency effects at given times.
* Repetitive audio effects.
* Making interesting sounds using interference patterns.
* Sampling existing signals and reusing portions.

What it is not intended for:

* Music generation
* Audio editing
* Sophisticated filters

However, there are lots of other good tools for these things and Easy can be used in conjunction with these other tools.

**Shortcomings**

As the project evolved, it was a clear that there was a tradeoff between complexity and simplicity and functionality vs. this-project-taking-forever. Audio is complicated. Nethertheless, here are a few things I’d like to improve:

- A proper lex/yacc based parser: this would allow better error catching and warnings. As it is, it isn’t easy to catch every unanticipated error in the command file.

- Math: It would be handy to simple formulas to play with frequencies and timing.

- Sounds: Although it doesn’t really much, it might be nice to make sounds more complex. More like midi instruments maybe, perhaps with ADSR envelopes.

**Sound Generation Organization**

EASY Sound

Generation

Generation

sound

sound

mix

overwrite

modulate

**commands**

defsound

deflfo

1.0 command

2.0-5. command

15.3 mix ...

etc.

Output Signal (Right)

completed our journey of renovations with only the ensuite bathroom left to do. It is now 4+2 bedrooms or 4+1 depending on whether you count the basement office. With minor changes the basement could be an apartment.e

Output Signal (Left)

completed our journey of renovations with only the ensuite bathroom left to do. It is now 4+2 bedrooms or 4+1 depending on whether you count the basement office. With minor changes the basement could be an apartment.e

LFO

sound

**SOUNDS**

Sounds are reusable basic building blocks to make output signals. These are monophonic signal shapes that can be used multiple times and merged into a final audio signals on either channel. Sounds can be short or long duration. They have the following core attributes:

**defsound NAME form:sine freq:900-1100 vol:.6-.9 fadein:2.5 fadeout:.5 length:60**

name – the name by which the sound can be later referenced. You don’t have to use capitals but it helps clarity if you do.

form – the base shape of the sound; defaults to ‘sine’. Others are:

1. sine – a pure sinusoid tone
2. comp – a compressed sinusoid with a quicker rise and longer peak
3. poly1 – a major combination chord with a tonic, fifth, and octave. This effect can be built by mixing Sounds but is provided for convenience.
4. poly2 – a minor combination chord with a tonic, minor third, and diminished seventh
5. saw – a sawtooth sound shape
6. square – a square sound
7. tens – a pulse train of 30uS pulses intended to mimic a TENS signal. These pulses are biphasic (positive and negative) and timed to support triphase effects. The overall density of the pulses will change with frequency and volume will correspond to amplitude (n.b. on some TENS units volume may drive pulse density so there are differences).

freq – frequency. This can be a single value or a sliding (glissando) frequency. In addition to a frequency specified in Hz, you can use notes. frequency starting with A-G like “F#5” will be converted to the appropriate midi note frequency.

vol – volume from 0 to 1, this can also slide.

fadein – an optional linear fade-in time in seconds.

fadeout – an optional linear fade-out time in seconds.

endsound – the length of the sound in seconds

Notes:

* The defsound command always sets the *current time reference back to zero*. This is to make use of the ‘modify’ command simpler to use.
* While defsound defines a sound, the sound is not actually created, or rendered in memory, until it is needed through a mix or module command.
* Limitation: when changing frequency, the synthesizer tries to keep the waveforms generated to be continuous without gaps. This imposes some tracking limitations on the rate in which *frequencies* and *phase shifts* can happen. Strange behaviour is possible.
* Special sounds created by **load** or **resample** commands: The ‘load’ command can also be used to fetch sounds out of existing audio [.wav formal - 16 bit 44.1kHz]. In this case a stereo signal is stored under the Sound name but everything else works the same. The left or right channel is selected by using SOUNDNAME.left and SOUNDNAME.right.

**Sound Modifiers**

Sound modifiers are ways of changing a sound signal *before* it is made and put into the output signal.

The format is fairly simple:

**12.3 modify LEFTSIG phase 0.3**

**12.3 modify LEFTSIG varyphase 0.3 5**

where:

‘12.3’ is the timestamp. Note that +/- in front of the timestamp makes it relative to the last timestamp.

‘LEFTSIG’ is the name of the Sound to be altered.

‘phase’ or ‘varyphase’ is the modify command. Most commands take a single number but a few take two.

‘0.3’ is the amount of modification (in this case, 30% ahead of phase)

‘5’ is the second argument for vary phase (5 seconds)

Let’s talk about the modifiers in four major categories: volume, frequency, phase, and oscillator-based.

Volume modifiers are:

Volume modifications are the most obvious in their effects. They vary the strength of signals.

**vol** – increase or decrease the volume of the signal. This is done immediately. The value is in +/- volume where 1. is full volume.

‘**bendvol**’ -

**‘fade’** - does a volume transition up or down. First argument is the amount (relative) to the current volume. The second argument is the length of time.

**‘bendvol’** - bends or fades the volume downward than restores it. The first number is the relative amount. The second number is the length of time. ‘e.g. **6 modify bendvol .1 1**’ Alters the volume by +.1 and then sets it back after 1 second.

**‘volabs’** - does an absolute immediate change in volume. The new volume is the argument.

Additionally: see warbtremelo below.

Frequency modifiers are:

Changing frequency induces interference patterns with other signals with both volume or phase shift effects.

**‘freq’** – shift the frequency of the signal up or down. The shift in frequency is applied after completion of the current sinusoid cycle to avoid audio clicks. The value is in +/- Hz from current frequency.

‘**bendfreq**’ - ‘e.g. **5 modify bendphase .4 2**’ Shifts the phase forward by 40% and then shifts it back after 2 seconds [It is unlikely that the phase will be preserved with respect other signals.]

Additionally: see warbvibrato below.

Phase modifiers are:

Phase changes typically have powerful triphase effects. By shifting the sine wave in relation to another, triphase effects are induced between the left and right channels.

**‘phase’** – shift the phase of the signal forward or back. For ‘back’, this is done by skipping updates until the sound is shifted. For ‘forward’, this is done by skipping values until the sound is shifted. The shift must be done gradually to supress audio clicks. The value is in +/- phase as a fraction [-.5:.5]

**‘bendphase’** - does a single change of phase and then shifts it back. The arguments are the amount of shift [-.5:.5] and the time [seconds].

**‘varyphase’** - does the bendphase command over and over. It has the same arguments as bendphase. It can be stopped with “varyphase 0 0”.

Oscillator effects (Warble):

There are 4 commands that trigger an built-in oscillator for minor alterations in the signal. To avoid confusion, I’ve called this “warble”.

‘**warb**’ – this configures a warble (it’s just a low frequency sine oscillator) in the sound generation and specifies the frequency used for vibrato or tremolo effects. The value is in Hz for the oscillator: this is normally a very low value, for example: 0.05Hz to 4Hz. *This frequency drives how fast the other (warbvibrato, warbtremelo) effects work.*

‘**warbvibrato**’ – this specifies the amount of vibrato (frequency variance) driven by the oscillator. The value is in Hz. All frequency effects come with limitations on the tracking speed of frequency changes.

‘**warbtremelo’** – this specifies the amount of tremolo (volume variance) driven by the oscillator. The value is in difference in volume scaled 0-1.

Examples:

**6 modify warb .1**

**6 modify warbvibrato 2**

... (starting at time 6), this will vary the frequency up and down every 10 seconds by a value of +/- 2Hz. So the if the wave you are generating is base frequency of 1000Hz, the frequency will be varying from 998Hz to 1002Hz in a 10 second cycle. There are tracking limitations on frequency changes, so strange things may happen if you vary the frequency too fast.

**6 modify warb .4**

**6 modify warbtremelo .3**

These commands will vary the volume by +/- 30% every 2.5 seconds.

Note - you can have vibrato and tremelo going at the same time.

‘**warbstop**’ – turns off the warble effects. A value is needed but is ignored at the moment.

Note that these types of warbling effects can be done in other ways through mixing and modulation and LFOs but there are some conveniences to providing other commands to do this.

You would normally add modifiers after defining the sound. However, if you mix the sound into the output signal, then add a modifier, the sound will be invalidated and rerendered the next time you use it. You probably don’t want to do this: it just makes the software grind through all the math again.

**Load**

The load command will create a Sound out of a pre-existing sound file. This uses the scipy library. At this time, 16-bit stereo audio files are supported. This is the most common format. The underlying library can’t do 24-bit integer and 32-bit float audio files: these are generally only used in specialized audio production applications. The software cannot open or translate .mp3 files as there is no reliable library in Python to do so (the convert command is suggested instead).

The command format is:

**load KICK “samples/kickdrumbeat.wav” start:20.0 end:25.0**

In this example, the software will go open the filename given. A new Sound named “KICK” will be created using the audio from the 20 second mark to the 25 second mark.

These samples can be used like any other Sound signal: in mixes or modulations, with one major difference. These Sounds will be stereo. To use them in a mix use the ‘.left’ and ‘.right’ specifiers.

For example:

**load SONG “samples/strictmachine.wav”**

**balance left**

**0 mix SONG.left**

**balance right**

**0.05 mix SONG.right**

This will load an entire music file and write it into the output signal. The right track has a 50ms delay which will produce an “expanded” stereo effect. I don’t know how useful this is but it is kind of cool.

To load a shorter sample use the ‘start’ and ‘end’ tags:

**load SONG “samples/strictmachine.wav” start:20 end:25**

**Resample**

The ‘resample’ command will create a Sound out of a section of the output signal. You can use this to apply further effects and as a copy-paste type mechanism.

Here is an example use:

**20-25 resample GOODBIT**

**30 mix GOODBIT fadein:.5**

**40 mix GOODBIT fadein:.5**

This creates a stereo sound. The ‘fadein’ and ‘fadeout’ options can be used.

**LFO**

LFO, or Low Frequency Oscillators, are signals used to modify output signals. These are so-named because they mimic similar functions in synthesizers. Internally, these are normally modelled one soundlength of a signal that repeats, although optionally a longer signal can be generated. LFOs contain only positive values.

LFOs are created with the deflfo command:

**deflfo NAME form:square freq:12.3-20.0 minvol:.1 maxvol:.9 duty:.6 length:8**

where arguments are:

name … the name which the LFO. Once again, caps are recommended.

form … the sound shape of the LFO. Options are ‘sine’, ‘square’, and ‘saw’. Defaults to ‘square’.

freq… the frequency of the LFO. Frequency may be a single value or a start-finish value as shown.

minvol… the minimum value of the signal at “off”

maxvol… the maximum value of the signal when “on”

duty… the duty cycle of the LFO (0-1) for square sounds only. This is the percentage of time that the signal is at the maxvol level.

LFOs can be used in several ways:

1. The ‘**amod**’ (Amplitude Modification) command: this will vary the volume of the output signal according to the values in the LFO. For example, if the output is a 1000Hz sine sound and the LFO is a 1Hz square sound the output signal will be a sign sound that turns on and off every second.

Note: AMOD is influenced by the balance setting for right/both/left in an all-or-nothing fashion.

1. The mix command has two LFO options:
   1. ‘**lfo**’: This does an amplitude modulation on a soundform as it is being mixed into the output signal. The argument is LFO name. To put it another way, the value of the LFO becomes the volume in both channels.
   2. ‘**lfobal**’: Using this option, the LFO values drive the distribution of signal from left to right in the mix. The argument is LFO name. Note that this command treats the middle position as “50%” in each channel. The right channel is “reverse” volume from left channel.

**Output Signal Commands**

The output signal is what is actually written to the final file. Unlike Sounds or LFOs, this is a stereo signal (by default).

**Balance**

Balance between left and right is controlled by the ‘balance’ command.

**balance left # sets the balance to left**

**balance 1 # also sets the balance to left**

**balance right # sets the balance to right**

**balance .75 # sets the balance mostly left**

**balance 0 # also sets the balance to right**

**balance .5 # equal/full volume in both channels**

**balance centre # equal/full volume in both channels**

**balance center # equal/full volume in both channels**

**balance middle # equal/full volume in both channels**

**balance both # equal/full volume in both channels**

Conceptually, this tries to emulate a balance knob on an amplifier.

Some commands may ignore the balance where it doesn’t make sense. For example, modulate, it doesn’t really make sense as modulate involves multiplication. Balance, therefore, is treated as on or off.

**Mix**

Mix copies a sound into the output signal starting at the time given. The sound is added to whatever signal is already there. This is done at the full strength of the sound (which is likely not full volume) if not specified or can be reduced or amplified in volume.

**12.3 mix PULSE1**

... is the simplest form of the mix command. This adds the contents of sound ‘PULSE1’ at 12.3 seconds into the output signal without modifying it’s volume. The mix command is applied to which every channel(s) are specified by previous ‘balance’ commands.

**+3 mix PULSE1 vol:.5**

… and this command following would repeat the mix operation 3 seconds later at time 15.3 seconds and 50% volume. Note that if the first sound was longer than 3 seconds it would be added to itself (like an echo).

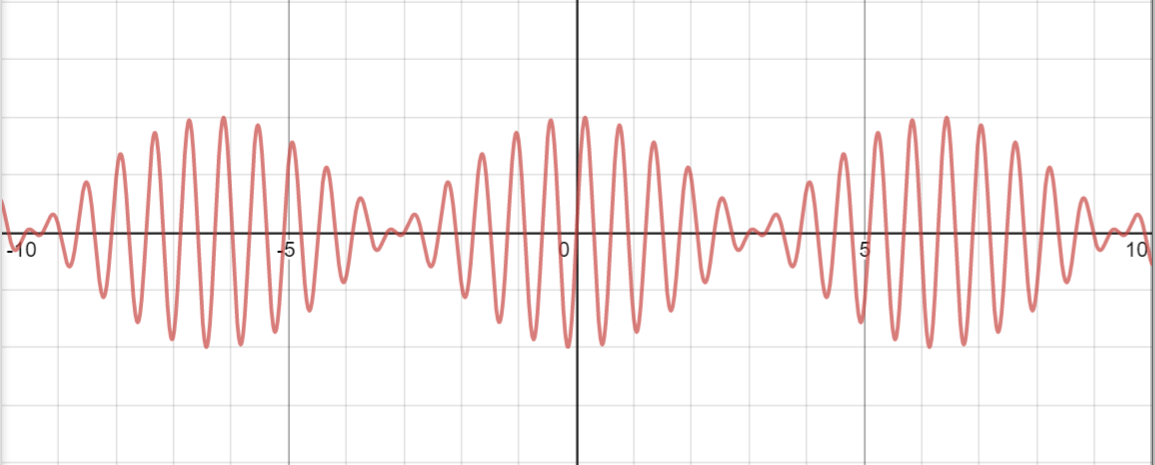
You can see that ‘mix’ commands can *easily cause clipping* – this is where the signal saturates at 100% volume. Sometimes this may be desirable.

As noted in the LFO section, there are two modifiers that can be used to alter volume during the mixing operation.

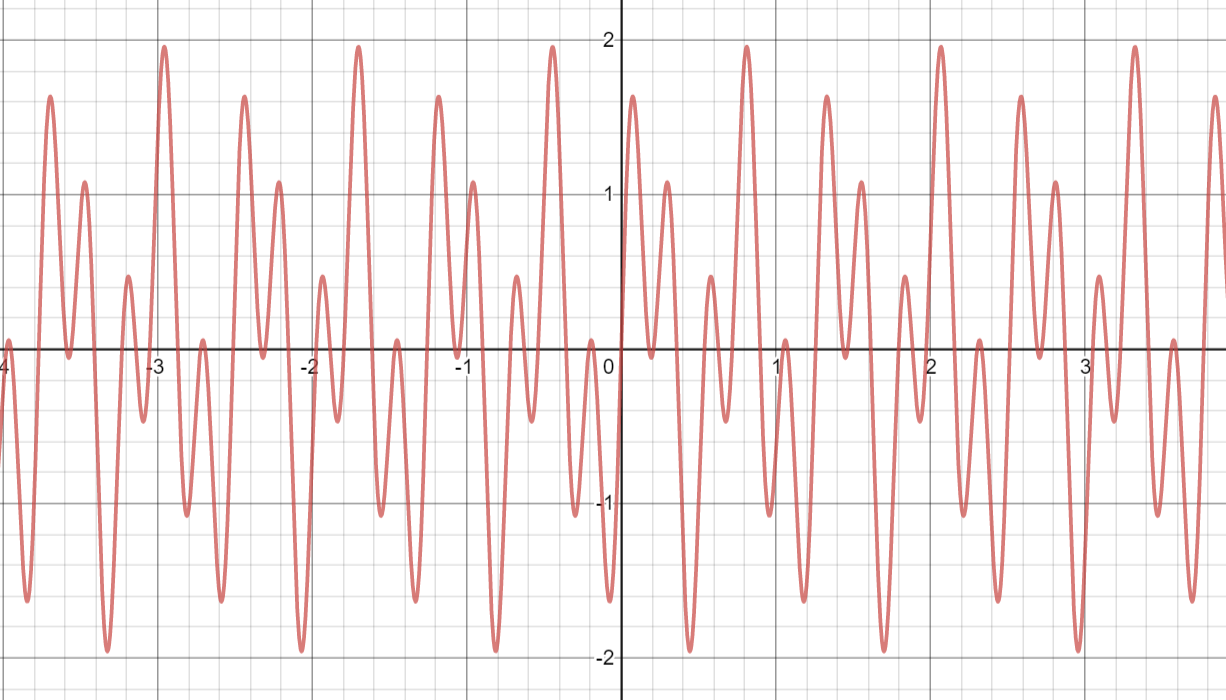
**0 mix BASE lfo:THROB**

… the ‘lfo’ subcommand will cause the volume to be varied according to the LFO values. At low values, this is another way to create a tremolo.

Here is what mixing a 10Hz and an 11Hz sine sound looks like:



And for 10Hz and 25Hz:



Note on the vertical scale that that ‘mix’ easily drives the signal over 100%. This signal would be clipped at +/- 1 if applied to audio and would be quite harsh.

Another variation on the lfo option exists…

**0 mix BASE lfobal:THROB**

… using the ‘lfobal’ command causes the volume to swing back and forth from left to right according to the LFO signal.

The mix command can also take a time range:

**20-25 mix BASE**

…this limits the amount of a Sound that will be transferred to the output signal. In this case, only the first 5 seconds of BASE will be transferred starting at 20 seconds into the output. The default of full volume will be used.

Note that the time parameter is precise. This can be used to create phase effects. For example:

**defsound MYSOUND freq:1000 form:sine vol:.8 length:10**

**balance left**

**0 mix MYSOUND # wavelength is 1ms**

**balance right**

**0.0005 mix MYSOUND # 0.5ms out of phase**

...will create LEFT and RIGHT channels 50% out of phase with strong triphase effects.

**Modulate**

Modulate is the other major way to copy a sound into an output signal. While ‘mix’ works by addition**,** ‘modulate’ works by **multiplication.** Consider the following commands:

**0 overwrite BASE**

**5 modulate FIVEHZ .8**

…the first command copies some base signal into the output sound. The second command does a frequency modulation of that sound with a sound named FIVEHZ at strength .8 or 80%.

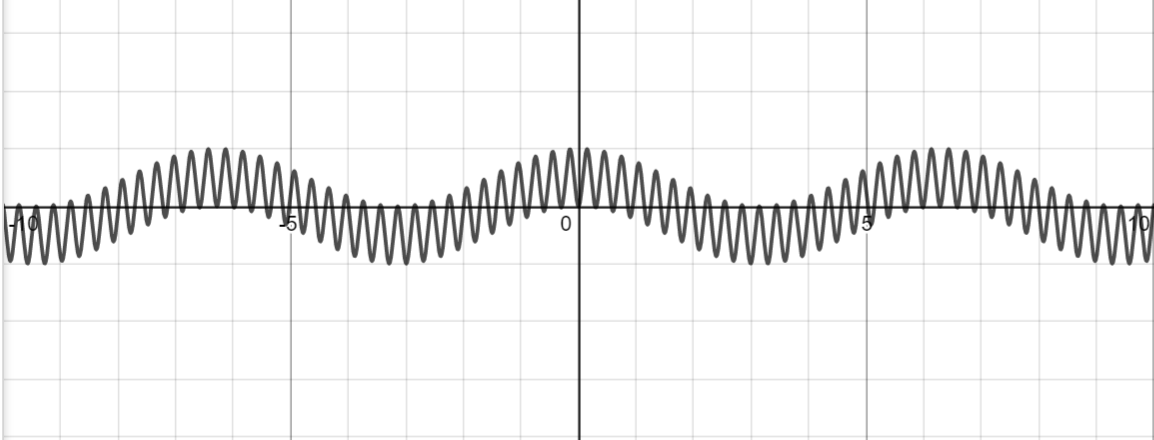
Now, this is a pure modulation, and at time of writing there is no wet/dry signal mix command yet but this could be done through a subsequent ‘mix’ of adding pure BASE signal back in.

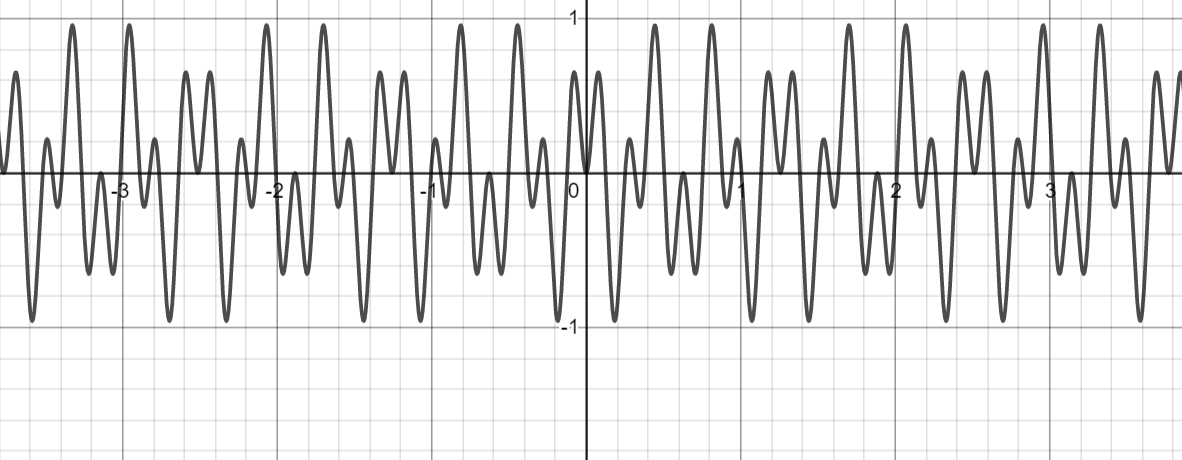
Frequency modulation can produce some very interesting effects. The following very important trig identity applies:

productSines.gif

This is widely used in music and engineering. Combining two frequencies will get you a mix of the **sum** and **difference** of the frequencies.

Here is the effect of modulating a 10Hz and 11Hz sine sound:



The effect is more dramatic as the frequencies differ more (this is 10Hz and 25Hz):

Note that modulation produces a very different signal with frequency effects. The signal is not overdriven above 100% so it is not clipped.

**Overwrite**

The overwrite command is just like the mix command except whatever is already in the output signal is erased.

**20 overwrite SWOOP**

This can be quite useful to intercut one signal with another. If the balance command is >.5 on left it will overwrite at full strength. At <.5 it will be at reduced strength. The opposite is true for right channel. Setting balance at .5 exactly makes the copying of the signal at full-strength mono.

**Silence**

Silence will simply zero out the output signal.

**15-20 silence**

…will create 5 seconds of silence from 15 seconds to 20 seconds on the timeline.

Repeated silence commands can create a crude amplitude modulation effect, but there are better ways to do it.

**Amplify**

Simply amplifies a time range. The volume range can be a single value or can be two values separated by a dash.

**15-20 amplify 0.2-1.3**

Note: the values are like the equivalent Audacity function. So, ‘1.0’ is keep the volume the same. During testing, I discovered negative values will make the software crash. Apologies, but it is really hard to catch every syntax problem.

**Amod**

Amod is short for amplitude modulation. It applies an LFO to the output signal, using the changing values in the output signal to vary the volume. Consider the following examples:

**lfo SQUARE40 square freq:40**

**0 amod SQUARE40 $(END)**

…will apply a 40Hz square sound volume filter to the entire output signal created so far. This could be used to create a dirty bass sound or talking-into-fan effect.

**lfo SINSLIDE sine freq:0.1-2 len:10 volmin:0.5 volmax:0.9**

**0 amod SINSLIDE 30**

... would create a sine function tremolo that gets faster as it goes repeating itself 3 times. There would always be some original signal coming through.

The order in which these effects are applied will do very different things, much like creating a layered drawing with different opacities.

**Echo**

The echo command creates a simple echo within one channel. The command format is:

**10-20 echo lvol:.2 rvol:.1 delay:.5**

Where:

… lvol is left channel echo strength

…rvol is right channel echo strength

…delay is the delay time in seconds

Echoes can echo repeatedly. Multiple echoes can be applied if a muddier reverb-like effect is desired. At the present time, there is no cross-channel echo capability. The echo command uses the volumes specified and ignores the balance command.

Caution: if volume is too high, echoes can drive a signal into clipping.

**Compression**

Compression increases the volume of all signals by some amount achieving an overall increase in loudness. The amount is proportional to the reference level specified, the strength factor for the effect, and the difference between the signal value and the reference level. The compressor model used is simplistic.

For example, if the reference level is (+/-) .8, and the value is at .4, and the strength is at .5: the value at that point might be increased to .6. The negative value for the same would be “increased” to -.6. To pick another figure, a value of .7 would be increased to .75.

It is possible to specify a target value above 1, forcing some degree of clipping, but this has very limited uses.

The command format looks like this:

**0-60 compress target:.8 strength:.25**

Note: compression uses the balance setting. To do both channels, set ‘balance both’.

**Soft Clipping**

Softclipping is applied to the output signal and is typically one of the last commands given. This is often desired when excess signal has been mixed into the output causing peak values to be driven into clipping (volumes above 1.).

In some ways, soft clipping is the opposite of compression. The goal is to remove harsh flatlines of soundforms as the audio system reaches its peak voltage. This distortion of sound shape leads to a spectrum of noise being produced at each clip. The alternative is to flatten each clip by rounding the curve. This also produces frequency distortions but they are more difficult to perceive.

The command looks like this:

**30-60 softclip strength:2.0 voladj:1.4**

This software uses an arctan function to smooth out the clips. There are two optional parameters, with reasonable defaults already programmed in. These are:

strength… the amount of the signal is modified

voladj… the amount of amplification applied to compensate for signal loss

Both of these values are typically set “low” and require some experimentation.

Note: compression uses the balance setting. To do both channels, set ‘balance both’.

**Pitch**

The pitch command can be used to shift the pitch of an output signal. The method used is by adding or deleting individual samples from the signal [in comparison, generating the original Sound at the desired frequency shift would be smoother]. This might be useful for a fully mixed signal to create interesting effects.

The pitch command will except varying pitch shifts. A range within +1.5 / .5 is recommended.

**10-20 pitch 1.0-1.2**

Note that strange interferences happen with pitch shifts. The software accomplishes pitch shifts by adding and deleting samples so that discontinuities don’t happen. Because audio files are 44.1KHz this is somewhat limiting: the amount of data is enormous but at the same time the resolution within each wavelength is not all that high. [A 1000Hz sine wave only has 44 samples in each wavelength.]

**Resample**

The resample command will copy a segment of the output signal back into a Sound signal which is created. Why would you want to do this? Possibly to reuse an interesting section of the output signal or somehow reuse it with further effects applied.

The format is:

30-32 resample GOODPART fadein:.5 fadeout:.5

**Time**

Example:

**20 time**

…sets the current reference time to 20 seconds. This is useful for using subsequent relative times inside repeats.

*Important! Whenever a ‘+’ is used in front of a time, the time is set forward by that many seconds. Only one current value for time is tracked. The current time is set to ‘0’ whenever a new sound is defined. There is more! So a ‘10’ second time in a Sound is different from ‘10’ second time in the Output signal. That Sound could be mixed into the Output at multiple different times.*

Why? This design +is intended to make it easy to make audio based on timelines.

**No Operation**

**+5 noop**

… does nothing 5 seconds after previous time stamp. Why have this? What is this command for? It can be used as an empty placeholder in macros or swaps.

**Building More Complex Audio Files**

Repeats, swaps, and macros are all tools that can be used to build more complicated files with the simple commands we’ve discussed.

**Repeat**

The repeat command very useful. The repeat command does the command that follows ‘n’ times.

For example:

**repeat 10 +5 mix BUZZ**

... mixes in the BUZZ sound every 5 seconds 10 times.

**Macro**

A macro is a multi-line command. The **‘define’** command is used to create one. For example:

**define SEQ**

**+.5 mix C**

**+1 mix B**

**+1 mix D**

**+.5 time**

**enddefine**

The **‘macro’** command is used to run these lines of code. For example:

**10 macro SEQ**

This makes more sense when the **repeat** command is also used:

**repeat 10 macro SEQ**

... in this case, 40 commands are run 3 different sounds being put into the output signal with spaces in between.

**Symbols / Swaps**

Symbols are simple swaps into the text of commands. There are different ways to use them. The syntax is:

**symbol=value**

*Symbols are defined with the ‘=’ sign. Don’t used spaces around the ‘=’.*

*Symbols are used with $(symbol). Once again, don’t use extra spaces.*

This could be for something simple, like setting frequencies:

freq1=1000

freq2=1000.5

defsound SOUND1 form:sine freq:$(freq1) vol:.4 length:30

defsound SOUND1 form:sine freq:$(freq2) vol:.4 length:30

A more complicated usage is found when using **erecorder.** When using erecorder, the template file created has a number of ‘event’ symbols made every time you press or release a key. These should be set to the command to be run at each time.

A good example is the cleo2.e file in the demo directory.

**20.4 $(event3)**

**20.4 $(event32)**

**20.5 $(event3R)**

**20.5 $(event3R\_2)**

You see that there are 2 events you can use for each key *pushed* and 2 events for each key *release*. These are defined as follows, with key down/up shifting phase and the other two being unused.

**event3=modify BASEL phase +.2**

**event3R=modify BASEL phase -.2**

**event3R\_2=noop**

**event32=noop**

This is used to insert sounds or modifiers on the timeline given. Obviously, this only works with certain effects and a fair amount of trial and error is needed to craft commands that work.

### Usage On Linux

Python3 should already be installed.

Install numpy

Install scipy

Install lame

Install id3v2

Build default examples with:

make demo

make test

make test2

this builds all the files in those directories.

to run, generally: python3 -f filename.e

find the .wav and .mp3 file in your current directories

### Usage on Windows

### (thanks to e-Stimson for this)

Installing Python and required libraries

1. Go here to get [Python](https://www.python.org/ftp/python/3.9.2/python-3.9.2-amd64.exe)3
2. Go to your Downloads folder and run the Python installer, making sure to **enable the option that adds Python to your PATH variable** - it just means you can run commands from anywhere.
3. Open a command prompt: Press the Windows key, and type in '***Command Prompt***' (without the single quotes and press Enter.
4. Now lets install the required Python libraries using pip (it's an installer that comes with Python)
5. In the Command Prompt type in ***pip install numpy***  
   It will install numpy
6. In the Command Prompt type in ***pip install scipy***  
   It will install scipy
7. Once that's all done, type '***Exit***' and press Enter, or just close the Command Prompt window.

Downloading and unzipping Easy

1. Download Easy from [Andrus' Mega Account](https://mega.nz/folder/b1NS1QAa" \l "9nzLdpeL1u75ATQmvxpQAg), Right click the 'Easy' folder and choose **Download** > **Standard Download**. It should download start downloading 'easy.zip'
2. Browse to your Downloads folder, right click easy.zip and choose Extract All accepting the default location. There should now be an 'easy' folder within your Downloads folder.

Running Easy

1. Open a new Command Prompt as per Step 3 above
2. type in '***cd Downloads\easy\easy***' press Enter
3. lets create a file using *eva.e* located in the *demo* folder  
   type in '***python.exe easy -f demo/eva.e***' press Enter
4. The debug / logging text will appear on screen letting you know what stage easy is at.
5. You'll now have a new eva.wav file in the Easy folder. (note, you'll probably get a couple of error lines talking about the mp3 and tagging - I didn't bother installing the lame encoder as I'm happy with the full fat wave files.)