A simple guide for using wAviCLe.

Introduction

wAviCLe's interface has been made as simple as possible to aid new users. We hope that this instructional page will allow people who have never tried using wAviCLe to edit audio files to understand how the program works and what it is capable of.

Details

We will be covering the following topics in discussing how to use wAviCLe:

- Requirements to use wAviCLe
- Loading and Running wAviCLe
- Creating lists of commands in .acf files
- Building filters in .flt files.

Requirements to use wAviCLe

In order to use wAviCLe, you will require the following software on your computer.

1. A recent version of the ACL2 interpreter (currently at ver 3.3)
2. A copy of PLT Scheme to run the program
3. A copy of drACuLa to allow PLT Scheme to interpret ACL2 code.

Loading and Running wAviCLe

Once the required setup is complete, actually running wAviCLe has been done on just a couple of commands. To load wAviCLe, you will need to open the main.lisp file in PLT Scheme and click the "Run" button. This will load all of the required functions needed to run wAviCLe and open an interface window in PLT Scheme so that you can get things started.

Now that everything has been loaded, just enter `(set-state-ok t)` into the interface window and PLT Scheme is ready to run wAviCLe.

Actually executing wAviCLe's code has been condensed down to a single command. The functions and sound files that you want to work with will be read from a .acf (audio command file) and the commands listed there will be run in order, so running wAviCLe requires you to use only the following command:

```
(main "acf" state)
```

where acf represents the location of the .acf file.

Creating lists of commands in .acf files

The .acf files are where most of the user's interaction with wAviCLe takes place. The program will automatically load .wav audio files, and run pre-made or user-made filters on them, combine multiple .wav files together and write the resulting soundwave to it's own audio file or draw it to a window on the screen.

wAviCLe works in three stages to apply changes to a .wav audio file, all of these stages are written into the .acf files that wAviCLe reads it's commands from. To output the results of your filters, wAviCLe uses these two commands at the beginning of the .acf file:

- `(put-signal output.wav wav-structure)` - Takes wav-structure, the output from filtered .wav files and other functions, then writes that to "output.wav" to playback your results in other audio programs.
- `(display-signal output.cvs wav-structure)` - Takes a wav-structure that has been filtered and displays it as a waveform in a window on the screen. Also writes the amplitudes in the output to a comma separated values file which can be opened by Excel or other spreadsheet applications. This allows users to visualize how the results of their filtering will look when written to a file.

Now that wAviCLe knows where to send the result, you need some sound data for those functions to output. The lambda function handles all changes to the wav-structures and provides it's result as another wav-structure for the output functions to use. To create a lambda function, you create a single function:

- `(lambda (var-list))` - Lambda creates a single function that will run several functions inside of itself. The var-list defines variables that will be used in the .acf file and can be as long as is needed, as long as each variable is defined later by an input function.

Inside of the lambda function, a user is free to perform any changes that they like to an audio file. After the var-list has been defined, the variables in it can be used with the functions of wAviCLe to create new wav-structures. The functions that can be used in lambda functions are:

- `(delay d wav-structure)` - This function changes the wav-structure by adding d seconds of silence to the beginning of it. The parameter d will need to be a number that represents the duration, in seconds, that you want to delay playback.
- `(cut t wav-structure)` - The cut function will remove the first and last t seconds from a wav-structure. The parameter t will need to be a number that represents the length of time, in seconds, that you want to remove from the beginning and end of the wav-structure.
- `(boost b wav-structure)` - Boost changes the sound data stored in the wav-structure by multiplying it's volume by b. Multiplying the volume can make the sound significantly louder or softer. The parameter b will need to be a positive number value.
- `(overdub wav-structure1 wav-structure2)` - Overdub creates a new wav-structure by playing one structure on top of another. The result of this function is like playing both of the wav-structures at the same time. This function can be used to combine changes to multiple sound files and merge them into one.
- `(fade-in b wav-structure)` - Fade-in gradually raises the volume of a wav-structure from 0 up to it's normal level over the first b seconds of that wav. The result is good way to begin a .wav sound file gradually. The parameter b needs to be a positive number that represents the duration in seconds that the user wants to fade the sound in.
- `(fade-out e wav-structure)` - Fade-out gradually lowers the volume of a wav-structure from it's normal level down to 0 over the last e seconds of that wav. The result is good way to let a .wav sound file end gradually and quietly. The parameter e needs to be a positive number that represents the duration in seconds that the user wants to fade the sound out.
- `(fuzz f wav-structure)` - Fuzz sets a maximum volume for a wav-structure. The parameter f sets a

value between 0 and 1 where the volume of the wav-structure will have a maximum value of f times the maximum value that the wav-structure can handle. Setting f to 0 would mute the wav-structure while setting f to 1 would leave the wav-structure unchanged.
- `(chipmunk c wav-structure)` - The chipmunk feature allows users to speed up or slow down the playback of of a sound file. This change also affects the pitch of the sound being played back so it can be used to make funny changes to your sound files. The parameter c represents a number to multiply the playback speed by.
- `(filter filter wav-structure)` - The filter function uses a filter read from a .flt file described below, to run a user-defined filter on a wav-structure. The filter will be read from a separate file and can be read defined however the user likes.
- `(echo d v wav-structure)` - The echo function creates an echo effect. The wav-structure is divided up by every d seconds of audio and every d seconds of audio is repeated with it's volume reduced by v. The parameter d represents the length of time in seconds of the audio that you would like to repeat and the parameter v is the amount each section with be boosted by on each repeat.
- `(reverse wav-structure)` - The reverse function takes a wav-structure and changes the audio so that it plays in reverse.

After a lambda function has been defined using the above functions, the variables that the lambda function uses need to be defined. In order to create the initial wav-structures and filters that the lambda function operations, the following functions are used:

- `(get-signal sample.wav)` - Read the file "sample.wav" from your hard disk and creates a wav-structure in order to allow wAviCLe to run it's filters on it.
- `(get-filter filter.flt)` - Read a user-made filter to be applied to the data in a wav-structure.

Note: File path may not contain any spaces when loading either wav files or filters.

Sample .acf file

Once the pieces of the lambda function have been brought together, defined, and sent to an output function, the result should be a single file that wAviCLe can read, interpret, and understand to perform the functions that it is being used for.

An example of a sample .acf file would be:

```
(put-signal output.wav
                ((lambda (f input1 input2)
                        (overdub (filter f input1)
                                 (chipmunk 2 input2)))
             (get-filter test.flt)
             (get-signal samples/voice1.wav)
             (get-signal samples/voice2.wav)))
```

The above .acf file is designed to run several of the functions available to it. The program starts at the bottom of the file and reads a filter (test.flt) as well as two wav files (samples/voice1.wav and samples/voice2.wav). The lambda function stores the filter as f and the wav files as a wav-structures input1 and input2 respectively.

Once the files have been read into structures in memory, the lambda function applies the filter to the first wav-structure and doubles the speed of the second wav-structure and reverses it's audio. The overdub function then takes the results of these two operations and dubs one over the other. The result of these changes is then sent to the put-signal operation and written out to a file called output.wav that can be opened in other audio processing programs.

Building filters in .flt files

The .flt files bring flexibility to wAviCLe by allowing users to create their own filters when none of the built-in functions perform the function that the user wants. Each .flt file consists of a list of numbers between -1 and 1, separated by commas. For each of these numbers, wAviCLe will take a sample of sound from the .wav file, multiply the filter value and the sample value together, then sum the products of each of those calculations together to create a new sample value for a new wav-structure.

A user may create as many filters as they like in an .flt file, for example:

```
0.33,0.33,0.33
```

will take the first three samples from a wav-structure, multiply them all by 1/3, then add them together and make them the first sample on a new wav-structure before moving up to perform the same function on samples 2, 3, and 4 of the old wav-structure and so-on until wAviCLe runs out of samples to filter. The resulting new wav-structure can then be written to a file by (put-signal) or displayed to the screen by (display-signal).