REAPER

JSFX Language Reference

September 6, 2017

version 5+

Introduction

This is a reference guide to programming audio-oriented effects for REAPER using JS. JS is a scripting language which is compiled on the fly and allows you to modify and/or generate audio and MIDI, as well as draw custom vector based UI and analysis displays. JS effects are simple text files, which when loaded in REAPER become full featured plug-ins. You can try loading existing JS effects and since they are distributed in source form, you can also edit existing effects to suit your needs (we recommend if editing an existing effect, you save it as something with a new name–if you do not you may lose your changes when upgrading REAPER). This guide will offer an outline of the structure of the text file used by JS, the syntax for writing code, as well as a list of all functions and special variables available for use.

Chapter 1

JSFX File Structure

1.1 Description Lines

JS Effects are text files that are composed of some description lines followed by one or more code sections.
The description lines that can be specified are:

1.1.1 `desc:Effect Description`

This line should be specified once and only once, and defines the name of the effect which will be displayed to the user. Ideally this line should be the first line of the file, so that it can be quickly identified as a JS file.

1.1.2 `slider1:5<0,10,1>Slider Description`

You can specify multiple of these lines (from 1-64 currently) to specify parameters that the user can control using standard UI controls (typically a fader and text input, but this can vary, see below). These parameters are also automatable from REAPER.

In the above example, the first 1 specifies the first parameter, 5 is the default value of the parameter, 0 is the minimum value, 10 is the maximum value, and 1is the change increment. slider description is what is displayed to the user.

There are additional extended slider syntaxes.
One is: `slider1:0<0,5,1{zerolabel,onelabel,twolabel,threelabel,fourlabel,fivelabel}>some setting`

This will show this parameter with a list of options from "zerolabel" to "fivelabel". Note that these parameters should be set to start at 0 and have a change increment of 1, as shown above.
Another extended syntax is: `slider1:/some_path:default_value:slider description`

You can also hide sliders by prefixing their names with "-": `slider1:0<0,127,1>-Hidden parameter`

Such parameters will not be visible in the plug-in UI but still be active, automatable, etc.

1.1.3 `in_pin:Name / out_pin:Name`

These optional lines export names for each of the JS pins (effect channels), for display in REAPER's plug-in pin connector dialog. If the only named `in_pin` or `out_pin` is labeled "none", REAPER

will know that the effect has no audio inputs and/or outputs, which enables some processing optimizations. MIDI-only FX should specify `in_pin:none` and `out_pin:none`.

1.1.4 `filename:0,filename.wav`

These lines can be used to specify filenames which can be used by code later. These definitions include 0 (the index) and a filename. The indices must be listed in order without gaps – i.e. the first should always be 0, the second (if any) always should be 1, and so on.

To use for generic data files, the files should be located in the REAPER\Data directory, and these can be opened with file_open(), passing the filename index.

You may also specify a PNG file. If you specify a file ending in .png, it will be opened from the same directory as the effect, and you can use the filename index as a parameter to gfx_blit().

1.1.5 `options:option_dependent_syntax`

This line can be used to specify JSFX options:

- `options:gmem=someUniquelyNamedSpace`
  This option allows plugins to allocate their own global shared buffer, see gmem[].
- `options:want_all_kb`

1.1.6 `import filename`

You can specify a filename to import (this filename will be searched within the JS effect directory). Importing files via this directive will have any functions defined in their @init sections available to the local effect. Additionally, if the imported file implements other sections (such as @sample, etc), and the importing file does not implement those sections, the imported version of those sections will be used.

Note that files that are designed for import only (such as function libraries) should ideally be named xyz.jsfx-inc, as these will be ignored in the user FX list in REAPER.

1.2 Code Sections

Following the description lines, there should be code sections. All of the code sections are optional (though an effect without any would likely have limited use). Code sections are declared by a single line, then followed by as much code as needed until the end of the file, or until the next code section. Each code section can only be defined once. The following code sections are currently used:

1.2.1 `@init`

The code in the @init section gets executed on effect load, on samplerate changes, and on start of playback. If you wish this code to not execute on start of playback, you can set ext_noinit to 1.0, and it will only execute on load or samplerate change (and not on playback start/stop). All memory and variables are zero on load, and if no @serialize code section is defined, then all memory and variables will be re-zeroed before calling @init (on sample rate change, playback start/stop/etc).

1.2.2 `@slider`

The code in the @slider section gets executed following an @init, or when a parameter (slider) changes. Ideally code in here should detect when a slider has changed, and adapt to the new parameters (ideally avoiding clicks or glitches). The parameters defined with sliderX: can be read using the variables sliderX.

1.2.3 `@block`

The code in the @block section is executed before processing each sample block. Typically, a block is whatever length as defined by the audio hardware, or anywhere from 128-2048 samples. In this code section, the samplesblock variable will be valid (and set to the size of the upcoming block).

1.2.4 `@sample`

The code in the @sample section is executed for every PCM audio sample. This code can analyze, process, or synthesize, by reading, modifying, or writing to the variables spl0, spl1, ... spl63.

1.2.5 `@serialize`

The code in the @serialize section is executed when the plug-in needs to load or save some extended state. The sliderX parameters are saved automatically, but if there are internal state variables or memory that should be saved, they should be saved/restored here using file_var() or file_mem() (passing an argument of 0 for the file handle). (If the code needs to detect whether it is saving or loading, it can do so with file_avail() (file_avail(0) will return <0 if it is writing).

Note when saving the state of variables or memory, they are stored in a more compact 32-bit representation, so a slight precision loss is possible. Note also that you should not clear any variables saved/loaded by @serialize in @init, as sometimes @init will be called following @serialize.

1.2.6 `@gfx [width] [height]`

The @gfx section gets executed around 30 times a second when the plug-ins GUI is open. You can do whatever processing you like in this (Typically using gfx_*()). Note that this code runs in a separate thread from the audio processing, so you may have both running simultaneously which could leave certain variables/RAM in an unpredictable state.

The @gfx section has two optional parameters, which can specify the desired width/height of the graphics area. Set either of these to 0 (or omit them) to specify that the code doesn't care what size it gets. Note that these are simply hints to request this size – you may not always get the specified size. Your code in this section should use the gfx_w, gfx_h variables to actually determine drawing dimensions.

Chapter 2

Basic Code Reference

2.1 Introduction to EEL2

The core of JSFX is custom code written in a simple language (called EEL2), which has many similarities to C. Code is written in one or more of the numerous code sections. Some basic features of this language are:

- Variables do not need to be declared, are by default global to the effect, and are all double-precision floating point.
- Basic operations including addition (+), subtraction (-), multiplication (*), division (/), and exponential (^)
- Bitwise operations including OR (|), AND (&), XOR (~), shift-left (<<), and shift-right-sign-extend (>>). These all convert to integer for calculation.
- Parentheses "(" and ")" can be used to clarify precedence, contain parameters for functions, and collect multiple statements into a single statement.
- A semicolon ";" is used to separate statements from each other (including within parentheses).
- A virtual local address space of about 8 million slots, which can be accessed via brackets "[" and "]".
- A shared global address space of about 1 million slots, accessed via gmem[]. These words are shared between all JSFX plug-in instances.
- Shared global named variables, accessible via the "_global." prefix. These variables are shared between all JSFX plug-in instances.
- User definable functions, which can define private variables, parameters, and also can, optionally, access namespaced instance variables.
- Numbers are in normal decimal, however if you prefix an '$x' to them, they will be hexadecimal (i.e. $x90, $xDEADBEEF, etc).

- You may specify the ASCII value of a character using $'c' (where c is the character).
- If you wish to generate a mask of 1 bits in integer, you can use $~X, for example $~7 is 127, $~8 is 255, $~16 is 65535, etc.
- Comments can be specified using:
  // comments to end of line
  /* comments block of code that spans lines or is part of a line. */

2.2 Operator Reference

2.2.1 []

```
z=x[y];
x[y]=z;
```

You may use brackets to index into memory that is local to your effect. Your effect has approximately 8 million (8,388,608) slots of memory and you may access them either with fixed offsets (i.e. 16811[0]) or with variables (myBuffer[5]). The sum of the value to the left of the brackets and the value within the brackets is used to index memory. If a value in the brackets is omitted then only the value to the left of the brackets is used.

```
z=gmem[y];
gmem[y]=z;
```

If 'gmem' is specified as the left parameter to the brackets, then the global shared buffer is used, which is approximately 1 million (1,048,576) slots that are shared across all instances of all JSFX effects.

The plug-in can also specify a line (before code sections): options:gmem=someUniquelyNamedSpace which will make gmem[] refer to a larger shared buffer, accessible by any plugin that uses options:gmem=<the same name>. So, if you have a single plug-in, or a few plug-ins that access the shared namespace, they can communicate without having to worry about other plug-ins. This option also increases the size of gmem[] to be 8 million entries (from the default 1 million). – REAPER 4.6+

2.2.2 !x

returns the logical NOT of the parameter (if the parameter is 0.0, returns 1.0, otherwise returns 0.0).

2.2.3 -x

returns value with a reversed sign (-1 * value). • +value – returns value unmodified.

2.2.4 x ^ y

returns the first parameter raised to the power of the second parameter. This is also available the function pow(x,y)

2.2.5 `x % y`

divides two values as integers and returns the remainder.

2.2.6 `x << y`

converts both values to 32-bit integers, bitwise left shifts the first value by the second. Note that shifts by more than 32 or less than 0 produce undefined results. – REAPER 4.111+

2.2.7 `x >> y`

converts both values to 32-bit integers, bitwise right shifts the first value by the second, with sign-extension (negative values of y produce non-positive results). Note that shifts by more than 32 or less than 0 produce undefined results. – REAPER 4.111+

2.2.8 `x / y`

divides two values and returns the quotient.

2.2.9 `x * y`

multiplies two values and returns the product.

2.2.10 `x - y`

subtracts two values and returns the difference.

2.2.11 `x + y`

adds two values and returns the sum.

2.2.12 `x | y`

converts both values to integer, and returns bitwise OR of values.

2.2.13 `x & y`

converts both values to integer, and returns bitwise AND of values.

2.2.14 `x ~ y`

converts both values to 32 bit integers, bitwise XOR the values. – REAPER 4.25+

2.2.15 `x == y`

compares two values, returns 1 if difference is less than 0.00001, 0 if not.

2.2.16 `x === y`

compares two values, returns 1 if exactly equal, 0 if not. – REAPER 4.53+

2.2.17 `x != y`

compares two values, returns 0 if difference is less than 0.00001, 1 if not.

2.2.18 `x !== y`

compares two values, returns 0 if exactly equal, 1 if not. – REAPER 4.53+

2.2.19 `x < y`

compares two values, returns 1 if first parameter is less than second.

2.2.20 `x > y`

compares two values, returns 1 if first parameter is greater than second.

2.2.21 `x <= y`

compares two values, returns 1 if first is less than or equal to second.

2.2.22 `x >= y`

compares two values, returns 1 if first is greater than or equal to second.

2.2.23 `x || y`

returns logical OR of values. If 'x' is nonzero, 'y' is not evaluated.

2.2.24 `x && y`

returns logical AND of values. If 'x' is zero, 'y' is not evaluated.

2.2.25 `x ? y`

how conditional branching is done – similar to C's if/else

2.2.26 `x ? y : z`

If x is non-zero, executes and returns y, otherwise executes and returns z (or 0.0 if ': z' is not specified).

Note that the expressions used can contain multiple statements within parentheses, such as: x % 5 ? (f += 1; x *= 1.5;) : (f=max(3,f); x=0;);

2.2.27 `x = y`

assigns the value of 'y' to 'x'. 'y' can be a variable or an expression.

2.2.28 `x *= y`

multiplies two values and stores the product back into 'x'.

2.2.29 `x /= y`

divides two values and stores the quotient back into 'x'.

2.2.30 `x %= y`

divides two values as integers and stores the remainder back into 'x'.

2.2.31 `x ^= y`

raises x to the y-th power, saves back to 'x'

2.2.32 `x += y`

adds two values and stores the sum back into 'x'.

2.2.33 `x -= y`

subtracts 'y' from 'x' and stores the difference into 'x'.

2.2.34 `x |= y`

converts both values to integer, and stores the bitwise OR into 'x'

2.2.35 `x &= y`

converts both values to integer, and stores the bitwise AND into 'x'

2.2.36 `x ~= y`

converts both values to integer, and stores the bitwise XOR into 'x' – REAPER 4.25+

2.2.37 Notes for C programmers

- (and) (vs { }) – enclose multiple statements, and the value of that expression is the last statement within the block:

```
z = (
a = 5;
b = 3;
```

```
a+b;
); // z will be set to 8, for example
```

- Conditional branching is done using the ? or ?: operator, rather than if()/else.
```
a < 5 ?  b = 6; // if a is less than 5, set b to 6
a < 5 ?  b = 6 :  c = 7; //if a is less than 5, set b to 6, otherwise set c to 7
a < 5 ?  ( // if a is less than 5, set b to 6 and c to 7
b = 6;
c = 7;     );
```

- The ? and ?: operators can also be used as the lvalue of expressions:
```
(a < 5 ?  b :  c) = 8; // if a is less than 5, set b to 8, otherwise set c to 8
```

Simple Math Functions

sin(angle)

returns the Sine of the angle specified (specified in radians – to convert from degrees to radians, multiply by $pi/180, or 0.017453)

cos(angle)

returns the Cosine of the angle specified (specified in radians).

tan(angle)

returns the Tangent of the angle specified (specified in radians).

asin(x)

returns the Arc Sine of the value specified (return value is in radians).

acos(x)

returns the Arc Cosine of the value specified (return value is in radians).

atan(x)

returns the Arc Tangent of the value specified (return value is in radians).

atan2(x,y)

returns the Arc Tangent of x divided by y (return value is in radians).

sqr(x)

returns the square of the parameter (similar to x*x, though only evaluating x once).

`sqrt(x)`

returns the square root of the parameter.

`pow(x,y)`

returns the first parameter raised to the second parameter-th power. Identical in behavior and performance to the ^ operator.

`exp(x)`

returns the number e (approx 2.718) raised to the parameter-th power. This function is significantly faster than pow() or the ^ operator

`log(x)`

returns the natural logarithm (base e) of the parameter.

`log10(x)`

returns the logarithm (base 10) of the parameter.

`abs(x)`

returns the absolute value of the parameter.

`min(x,y)`

returns the minimum value of the two parameters.

`max(x,y)`

returns the maximum value of the two parameters.

`sign(x)`

returns the sign of the parameter (-1, 0, or 1).

`rand(x)`

returns a psuedorandom number between 0 and the parameter.

`floor(x)`

rounds the value to the lowest integer possible (floor(3.9)==3, floor(-3.1)==-4).

`ceil(x)`

rounds the value to the highest integer possible (ceil(3.1)==4, ceil(-3.9)==-3).

`invsqrt(x)`

returns a fast inverse square root (1/sqrt(x)) approximation of the parameter.

Time Functions

`time([v])`

REAPER 4.60+ Returns the current time as seconds since January 1, 1970. 1 second granularity. If a parameter is specified, it will be set to the timestamp.

`time_precise([v])`

REAPER 4.60+ Returns a system-specific timestamp in seconds. Granularity is system-defined, but generally much less than 1 millisecond. Useful for benchmarking. If a parameter is specified, it will be set to the timestamp.

Special Variables

Basic Functionality

spl0, spl1 ... spl63

Context: @sample only

Usage: read/write
The variables spl0 and spl1 represent the current left and right samples in @sample code. The normal +0dB range is -1.0 .. 1.0, but overs are allowed (and will eventually be clipped if not reduced by a later effect). On a very basic level, these values represent the speaker position at the point in time, but if you need more information you should do more research on PCM audio. If the effect is operating on a track that has more than 2 channels, then spl2..splN will be set with those channels values as well. If you do not modify a splX variable, it will be passed through unmodified. See also spl(x) below, though splX is generally slightly faster than spl(X).

spl(channel_index)

Context: @sample only
If you wish to programmatically choose which sample to access, use this function (rather than splX). This is slightly slower than splX, however has the advantage that you can do spl(variable) (enabling easily configurable channel mappings). Valid syntaxes include:

```
spl(channelindex)=somevalue;
spl(5)+=spl(3);
```

slider1, slider2 ... slider64

Context: available everywhere

Usage: read/write
The variables slider1, slider2, ... slider64 allow interaction between the user and the effect, allowing the effects parameters to be adjusted by the user and likewise allow the effect to modify the parameters shown to the user (if you modify sliderX in a context other than @slider then you should call sliderchange(sliderX) to notify JS to refresh the control). The values of these sliders are purely effect-defined, and will be shown to the user, as well as tweaked by the user.

slider(slider_index)

Context: available everywhere
If you wish to programmatically choose which slider to access, use this function (rather than sliderX). Valid syntaxes include:

```
val = slider(sliderindex);
slider(i) = 1;
```

slider_next_chg(slider_index,nextval)

Context: @block, @sample
Used for sample-accurate automation. Each call will return a sample offset, and set nextval to the value at that sample offset. Returns a non-positive value if no changes (or no more changes) are available.

trigger

Context: @block, @sample

Usage: read/write
The trigger variable provides a facility for triggering effects. If this variable is used in an effect, the UI will show 10 trigger buttons, which when checked will result in the appropriate bit being set in this variable.

For example, to check for trigger 5 (triggered also by the key '5' on the keyboard):

```
isourtrig = trigger & (2^5);
```

Conversely, to set trigger 5:

```
trigger |= 2^5;
```

Or, to clear trigger 5:

```
trigger & (2^5) ?  trigger -= 2^5;
```

It is recommended that you use this variable in @block, but only sparingly in @sample.

Index