REAPER | JS Programming

back to main reference page

JS Programming Reference

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.

JS File Structure

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:

- 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.

- slider**1**:**5**<**0**,**10**,**1**>**slider description**
  You can specify multiple of these lines (from 1-16 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 **1** is 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
  In the above example, the **/some_path** specifies a subdirectory of the REAPER\Data path, which will be scanned for .wav, .txt, .ogg, or .raw files. **default_value** defines a default filename. If this is used, the script will generally use file_open(slider1) in the @serialize code section to read the contents of the selected file.

- in_pin:**name_1**
  in_pin:**name_2**
  out_pin:**none**
  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.

- 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(). -- *REAPER 2.018+*

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:

- **@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 samplerate change, playback start/stop/etc).

- **@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.

- **@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).

- **@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.

- **@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.

- **@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.

  Note: the @gfx section should never call any file_* functions.

- **@import** *filename*
  (requires REAPER v4.25+)

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.

Basic Code Reference

The code in the code sections is a simple language that has similarities to C. Some basic features of this language are:

- Parenthesis "(" and ")" are used to clarify precidence, contain parameters for functions, or collect blocks of code into one statement.
- A semicolon ";" is used to separate statements from eachother (even within parenthesis, though it is not required for the last statement within parenthesis).
- Variables are not declared. Simply using a variable is enough to create it in the global scope. The JS editor has a list of variables with reference counts, so you can see if any variables have a reference count of 1 (which would often imply a typo).
- Variables have a global scope (meaning you can access them from any code section), however user functions can declare private variables via the local directive.
- All variables represent real values (i.e. 0, 1, 3.14159, -3, 3000000000000000.0 are all values that they can represent).
- In addition to variables, JS effects also have approximately 8 million slots of local RAM, and 1 million slots of shared (global across all effects) RAM. Note however that variables are quite a bit faster than RAM, so you should keep things that will be accessed a lot in variables whenever possible.
- Numbers are in normal decimal, however if you prefix an '$x' to them, they will be hexadecimal (i.e. $x90, $xDEADBEEF, etc). -- *(REAPER v4.25+ can also take traditional syntax, i.e. 0x90)*
- 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. -- *(REAPER 4.25+).*
- Comments can be specified using:
  - // comments to end of line

- /* comments block of code that span lines or be part of a line */

Basic operators:

- **=**
  Example: y = z;
  Assigns the value of 'z' to 'y'. 'z' can be a variable or an expression.
- *
  Example: y * z
  Multiplies two values and returns the product.
- /
  Example: y / z
  Divides two values and returns the quotient.
- %
  Example: y % z
  Divides two values and returns the remainder.
- **^**
  Example: y ^ z
  Returns the first parameter raised to the second parameter-th power.
- **+**
  Example: y + z
  Adds two values and returns the sum.
- **-**
  Example: y - z
  Subtracts two values and returns the difference.
- |
  Example: y | z
  Converts both values to integer, and returns bitwise OR of values.
- **&**
  Example: y & z
  Converts both values to integer, and returns bitwise AND of values.
- ***=**
  Example: y *= z
  Multiplies two values and stores the product back into 'y'.
- **/=**
  Example: y /= z
  Divides two values and stores the quotient back into 'y'.
- **%=**
  Example: y %= z

Divides two values and stores the remainder back into 'y'.

- **^=**
  Example: y ^= z
  Raises first parameter to the second parameter-th power, saves back to 'y'.
- **+=**
  Example: y += z
  Adds two values and stores the sum back into 'y'.
- **-=**
  Example: y -= z
  Subtracts two values and stores the difference back into 'y'.
- **|=**
  Example: y |= z
  Converts both values to integer, and stores the bitwise OR into 'y'
- **&=**
  Example: y &= z
  Converts both values to integer, and stores the bitwise AND into 'y'
- **||**
  Example: y || z Returns logical OR of values. If y is nonzero, 'z' is not evaluated.
- **&&**
  Example: y && z
  Returns logical AND of values. If y is zero, 'z' is not evaluated.
- **==**
  Example: y == z
  Compares two values, returns 1 if difference is less than 0.00001, 0 if not.
- **!=**
  Example: y != z
  Compares two values, returns 0 if difference is less than 0.00001, 1 if not.
- **<**
  Example: y < z
  Compares two values, returns 1 if first parameter is less than second.
- **>**
  Example: y > z
  Compares two values, returns 1 if first parameter is greater than second.
- **<=**
  Example: y <= z
  Compares two values, returns 1 if first is less than or equal to second.
- **>=**
  Example: y >= z

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

- **>>** *-- REAPER 4.111+*
  Example: z = y >> x
  Converts both values to 32 bit integers, bitwise right shifts the first value by the second. Note that shifts by more than 32 or less than 0 produce undefined results.
- **<<** *-- REAPER 4.111+*
  Example: z = y << x
  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 4.25+*
  Example: z = y ~ x
  Converts both values to 32 bit integers, bitwise xor the values.
- **~=** *-- REAPER 4.25+*
  Example: y ~= x
  Convert both values to 32 bit integers, store bitwire xor back into y.
- **!**
  Example: !z
  Returns the logical NOT of the parameter (if the parameter is 0.0, returns 1.0, otherwise returns 0.0).
- **?** *-- this is how branching is done -- like C's if()*
  ?:
  Example: y ? z;
  Example: y ? z : x;
  Evaluates the first parameter 'y', and if nonzero, evaluates and returns the second parameter 'z'. If a third parameter 'x' is specified (after the **:**), and the first parameter is zero, evaluates and returns 'x'. The latter two expressions may often contain multiple statements seperated by semicolons, i.e.

```
x % 5 ? (
  f += 1;
  x *= 1.5;
) : (
  f=max(3,f);
  x=0;
);
```

- **[ ]**
  Example: z=x[y]; or x[y]=z;

Example: z=gmem[y]; or gmem[y]=z;
In the first form, 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]). If a value in the brackets is omitted then 0 is used instead.

If 'gmem' is specified (the second form), then instead of local effect memory, the buffer is the global storage buffer, which is approximately 1 million (1,048,576) slots that are shared across all effects.

Some key notes about the above:

- (and) are used to override default precedence, or to enclose multiple statements. For example:
  - x = (y+1)*5;
  - (a = 5; b = 3;);
- Branching is done using the ? operator, rather than if()/else. For example:
  - a < 5 ? b=1; // if a is less than 5 then set b to 1.
  - a < 5 ? b=1 : c=1; // if a is less than 5 set b to 1, otherwise set C to 1.
  - a < 5 ? (b=1; c=1;); // if a is less than 5 set b and c to 1.

Basic math functions:

- **sin(angle)**
  Example: s = sin(theta);
  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)**
  Example: s = cos(theta);
  Returns the Cosine of the angle specified (specified in radians).

- **tan(angle)**
  Example: s = tan(theta);
  Returns the Tangent of the angle specified (specified in radians).

- **asin(x)**
  Example: theta = asin(s);
  Returns the Arc Sine of the value specified (return value is in radians).

- **acos(x)**
  Example: theta = acos(s);
  Returns the Arc Cosine of the value specified (return value is in radians).

- **atan(x)**
  Example: theta = atan(s);
  Returns the Arc Tangent of the value specified (return value is in radians).

- **atan2(x,y)**
  Example: theta = atan2(s,sd);
  Returns the Arc Tangent of s divided by sd (return value is in radians). This uses the signs of the inputs to determine the correct quadrant.

- **sqr(x)**
  Example: ss = sqr(s);
  Returns the square of the parameter (i.e. same as ss=s*s;).

- **sqrt(x)**
  Example: ss = sqrt(s);
  Returns the square root of the parameter.
  The parameter should be greater than or equal to zero.

- **pow(x,y)**
  Example: p = pow(s,y);
  Returns the first parameter raised to the second parameter-th power.
  Note: This is exactly the same as the operator ^, and is provided for compatibility.

- **exp(x)**
  Example: ss = exp(s);
  Returns the number e (approx 2.718) raised to the parameter-th power.

- **log(x)**
  Example: ss = log(s);

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

- **log10(x)**
  Example: ss = log10(s);
  Returns the logarithm (base 10) of the parameter.

- **abs(x)**
  Example: ss = abs(s);
  Returns the absolute value of the parameter.

- **min(x,y)**
  Example: s = min(x,y); Returns the minimum value of the two parameters.

- **max(x,y)**
  Example: s = max(x,y);
  Returns the maximum value of the two parameters.

- **sign(x)**
  Example: s = sign(x);
  Returns the sign of the parameter (-1, 0, or 1).

- **rand(x)**
  Example: s = rand(x);
  Returns a psuedorandom whole number between 0 and the parameter.

- **floor(x)**
  Example: s = floor(x);
  Rounds the value to the lowest integer possible (i.e. 3.9 becomes 3).

- **ceil(x)**
  Example: s = ceil(x);
  Rounds the value to the highest integer possible (i.e. 3.1 becomes 4).

- **invsqrt(x)**
  Example: s = invsqrt(x);
  Returns a fast inverse square root (1/sqrt(x)) approximation of the parameter.

Advanced Functions

In addition to the basic functions available, REAPER/JS also provides numerous advanced functions, organized into the following categories:

Looping, FFT/MDCT, Memory Utility, Slider, File, MIDI, and Graphics.

Accessing Samples

- **spl(channelindex)** -- *REAPER 2.018+*
  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);
```

Looping

- **loop(count,code)**
  Example:

```
loop(32,
   r += b;
   b = var * 1.5;
);
```

  Evaluates the second parameter a finite number of times, specified by the first parameter. If the first parameter is less than 1, the second parameter is not evaluated. Be careful with specifying large values for the first parameter -- it is possible to hang

your effect for long periods of time doing so. The maximum for the first parameter is approximately 1 million.

The first parameter is only evaluated once (so modifying it within the code will have no effect on the number of loops). For a loop of indeterminate length, see while() below.

- **while(code)**
  Example:

```
while(
  a += b;
  b *= 1.5;
  a < 1000; // as long as a is below 1000, we go again.
);
```

Evaluates the first parameter a finite number of times, until the last statement in the code block is zero. There is an artificial limit for number of executions of about 1 million (If you need more than 1 million iterations you can nest while()s, but this isn't recommended as you will slow things down and likely break things).

FFT/MDCT Functions

- **mdct(start_index, size), imdct(start_index, size)**
  Example:

```
mdct(0, 512);
```

Performs a modified DCT (or inverse in the case of imdct()) on the data in the local memory buffer at the offset specified by the first parameter. The second parameter controls the size of the MDCT, and it MUST be one of the following: 64, 128, 256, 512, 1024, 2048, or 4096. The MDCT takes the number of inputs provided, and replaces the first half of them with the results. The IMDCT takes size/2 inputs, and gives size results.

Note that the MDCT must NOT cross a 65,536 item boundary, so be sure to specify

the offset accordingly.

The MDCT/IMDCT provided also provide windowing, so your code is not required to window the overlapped results, but simply add them. See the example effects for more information.

- **fft(start_index, size), ifft(start_index, size)**
  fft_permute(index,size), fft_ipermute(index,size)
  Example:

```
buffer=0;
fft(buffer, 512);
fft_permute(buffer, 512);
buffer[32]=0;
fft_ipermute(buffer, 512);
ifft(buffer, 512);
// need to scale output by 1/512.0, too.
```

  Performs a FFT (or inverse in the case of ifft()) on the data in the local memory buffer at the offset specified by the first parameter. The size of the FFT is specified by the second parameter, which must be 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, or 32768. The outputs are permuted, so if you plan to use them in-order, call **fft_permute(idx, size)** before and **fft_ipermute(idx,size)** after your in-order use. Your inputs or outputs will need to be scaled down by 1/size, if used.

  Note that the FFT/IFFT require real/imaginary input pairs (so a 256 point FFT actually works with 512 items), while the real FFT/IFFT (rfft() and irfft(), below) provide a real-FFT.

  Note that the FFT/IFFT must NOT cross a 65,536 item boundary, so be sure to specify the offset accordingly.

- **convolve_c(dest,src,size)**
  Used to convolve two buffers, typically after FFTing them. convolve_c works with complex numbers. The sizes specify number of items (the number of complex number pairs).

Note that the convolution must NOT cross a 65,536 item boundary, so be sure to specify the offset accordingly.

Memory Utility

- **freembuf(top)**
  The freembuf() function provides a facility for you to notify the memory manager that you are no longer using a portion of the local memory buffer.

  For example, if the user changed a parameter on your effect halving your memory requirements, you should use the lowest indices possible, and call this function with the highest index you are using plus 1, i.e. if you are using 128,000 items, you should call freembuf(128001); If you are no longer using any memory, you should call freembuf(0);

  Note that calling this does not guarantee that the memory is freed or cleared, it just provides a hint that it is OK to free it.

- **memcpy(dest,source,length)**
  The memcpy() function provides the ability to quickly copy regions of the local memory buffer. The regions may overlap, but neither region may cross a 65,536 item boundary (they may be on different pages, however).

- **memset(dest,value,length)**
  The memset() function provides the ability to quickly set a region of the local memory buffer to a particular value. Unlike memcpy(), this region may be of any length and cross any boundaries.

Slider Functions

For these functions, the parameter can be the variables slider1-sliderN, in which case that slider is refreshed. Otherwise, it can be a bitmask of which sliders have changed, where 1 would be the first slider, 2 would be the second, 4 would be the third, 32768 being the 16th slider, and so on.

- **sliderchange(mask or sliderX)**
  Example:

```
sliderchange(slider4);
```

or

```
sliderchange(2 ^ sliderindex);
```

The sliderchange() function provides a facility for you to notify REAPER/JS that you have changed a sliderX variable so that it can update the display. This function is not necessary to call from the @slider code section, it is provided so that other code sections can update the sliders. Note that this function only updates the display, it does not send an automation message to the host.

- **slider_automate(mask or sliderX)**
  Example:

```
slider_automate(slider4);
```

or

```
slider_automate(2 ^ sliderindex);
```

The slider_automate() function provides a facility for you to notify REAPER/JS that you have changed a sliderX variable so that it can update the display, and record the move as automation. This function is not necessary to call from the @slider code section, it is provided so that other code sections can write programmatic automation messages.

File Functions

The following functions can be used in the @serialize section or in other sections (for reading of files specified with file: or from an extended file slider.

In a @serialize section, they can be for read or for write, but on all others they are for reading only.

For @serialize, file_open() and file_close() should not be used, and the file handle should be 0.

These functions should NOT under any circumstances be used in the @gfx section.

- **file_open(index or slider)**
  Example:

```
filename:0,myfile.wav
handle = file_open(0);
```

  Example:

```
slider1:/mydata:mydef.wav:WAV File
handle = file_open(slider1);
```

  Opens a file from either the effect filename list or from a file slider. Once open, you may use all of the file functions available. Be sure to close the file handle when done with it, using file_close().

  If file_open() fails, it will return < 0 (usually -1).

- **file_close(handle)**
  Example:

```
file_close(handle);
```

  Closes a file opened with file_open().

- **file_rewind(handle)**
  Example:

```
file_rewind(handle);
```

Use this to rewind the current file to the beginning, to re-read the file etc.

- **file_var(handle,variable)**
  Example:

```
file_var(handle,myVar);
```

  This reads (or writes if in a @serialize write) the variable from(to) the current file.

- **file_mem(handle,offset, length)**
  Example:

```
amt=file_mem(handle,offset,len);
```

  This reads (or writes) the block of local memory from(to) the current file. Returns the actual number of items read (or written).

- **file_avail(handle)**
  Example:

```
len=file_avail(handle);
```

  Returns the number of items remaining in the file, if it is in read mode. Returns < 0 if in write mode. If the file is in text mode (file_text(handle) returns TRUE), then the return value is simply 0 if EOF, 1 if not EOF.

- **file_riff(handle,nch,samplrate)**
  Example:

```
file_riff(handle,nch,samplrate);
nch ? file_mem(handle,0,file_avail(0));
```

  If the file was a RIFF WAV file, or a valid .OGG Vorbis file, this will set the first para-

meter to the number of channels, and the second to the samplerate.

file_text(handle,istext)
Example:

```
istext=file_text(handle);
istext ? use_diff_avail syntax;
```

If the file was a text file (and ended in .txt), this will return 1. If you need to use different file_avail() logic for text files (you often will), you can query it this way.

Text file notes
Note that if in a extended file-slider code section, and the extension of the file is .txt, it will read one line at a time, ignoring non-number lines. Note that file_avail() should be called to check for EOF after each read, and if it returns 0, the last file_var() should be ignored.

You can also use file_mem(offs,bignum) and it will read the maximum available.

The format of each line in the text file can be either a floating point number, a binary number beginning with 'b', i.e. b0101010111, or a hexadecimal number beginning with 'x', i.e. xDEADF000.

Additionally text files can create their own symbolic constants (using =), and combine them using basic +, -, |, & etc operators.

MIDI Functions
The following functions can be used in @block or @sample sections to send and receive MIDI.

Note that if a JS effect calls midirecv(), MIDI will not be passed through unless you also midisend() the received messages.

Offset is offset from current block, in samples. msg1 is status byte, msg23 is second (and third if available) data bytes, second byte is low 8 bits (msg23&$xff), third byte is next 8 bits (msg23/256)&$xff.

- **midisend(offset,msg1,msg23)**
  Example:

```
midisend(0,$x90 + 0,69|(127*256)); // send note 69 to
channel 0 at velocity 127 (max)
```

- **midirecv(offset,msg1,msg23)**
  Example:

```
midirecv(offset,msg1,msg23);
(msg1&$xF0)==$x90 ? ( /* note on! */ )
```

  Note: midirecv returns msg1 as a return value, as well.

  So you can do things like:

```
while( midirecv(offs,msg1,msg23) ? (
midisend(offs,msg1,msg23); ); );
```

- **midisyx(offset,msgptr,len)**
  Example:

```
buf[0] = $xAA|0;
buf[1] = $xBB|0;
midisyx(offset,buf,2);  // send sysex: F0 AA BB F7
```

Graphics Functions
Effects can specify a @gfx code section, from which the effect can draw its own custom UI and/or analysis display.

These functions must only be called from the @gfx section.

There are numerous Graphics State Variables that should be used as well.

- **gfx_lineto(x,y,aa)**
  Draws a line from gfx_x,gfx_y to x,y. if aa is > 0.5, then antialiasing is used. Updates gfx_x and gfx_y to x,y.

- **gfx_rectto(x,y)**
  Fills a rectangle from gfx_x,gfx_y to x,y. Updates gfx_x,gfx_y to x,y.

- **gfx_setpixel(r,g,b)**
  Writes a pixel of r,g,b to gfx_x,gfx_y.

- **gfx_getpixel(r,g,b)**
  Gets the value of the pixel at gfx_x,gfx_y into r,g,b.

- **gfx_drawnumber(n,ndigits)**
  Draws the number "n" with "ndigits" of precision to gfx_x, gfx_y, and updates gfx_x to the right side of the drawing. The text height is gfx_texth

- **gfx_drawchar($'c')**
  Draws the character 'c' (can be a numeric ASCII code as well), to gfx_x, gfx_y, and moves gfx_x over by the size of the character.

- **gfx_blurto(x,y)** -- *REAPER 2.018+*
  Blurs the region of the screen between gfx_x,gfx_y and x,y, and updates gfx_x,gfx_y to x,y.

- **gfx_blit(source, scale, rotation)** -- *REAPER 2.018+*
  Copies the entirity of the source bitmap to gfx_x,gfx_y using current opacity and copy mode (set with gfx_a, gfx_mode). You can specify scale (1.0 is unscaled) and rotation (0.0 is not rotated, angles are in radians).

  For the "source" parameter specify -1 to use the main framebuffer as source, or 0..n to use the PNG specified in a filename: line.

- **gfx_blitext(source, coordinatelist, rotation)** -- *REAPER 2.018+*
  This is much more flexible version of gfx_blit.

  For the "source" parameter specify -1 to use the main framebuffer as source, or 0..n to use the PNG specified in a filename: line.

coordinatelist should be an index to memory where a list of 10 parameters are stored. i.e.

```
  coordinatelist=1000; // use memory slots 1000-1009
  coordinatelist[0]=source_x;
  coordinatelist[1]=source_y;
  coordinatelist[2]=source_w;
  coordinatelist[3]=source_h;
  coordinatelist[4]=dest_x;
  coordinatelist[5]=dest_y;
  coordinatelist[6]=dest_w;
  coordinatelist[7]=dest_h;
  coordinatelist[8]=rotation_x_offset; // only used if rotation
is set, represents offset from center of image
  coordinatelist[9]=rotation_y_offset; // only used if rotation
is set, represents offset from center of image
  gfx_blitext(img,coordinatelist,angle);
```

- **gfx_getimgdim(image, w, h)** -- *REAPER 2.018+*
  Retreives the dimensions of image (representing a filename: index number) into w and h. Sets these values to 0 if an image failed loading (or if the filename index is invalid).

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() (though spl0 etc are slightly faster than spl()).

- **slider1, slider2, ... slider16**
  Context: available everywhere
  Usage: read/write
  The variables slider1, slider2, ... slider16 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.

- **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.

Audio and transport state:

- **srate**
  Context: available everywhere
  Usage: read-only
  The srate variable is set by the system to whatever the current sampling frequency is set to (usually 44100 to 192000). Generally speaking your @init code section will be called when this changes, though it's probably a good idea not to depend too much on that.

- **num_ch**
  Context: most contexts (see comments)
  Usage: read-only
  Specifies the number of channels available (usually 2). Note however splXX are still available even if this count is less, their inputs/outputs are just ignored.

- **samplesblock**
  Context: most contexts (see comments)
  Usage: read-only
  The samplesblock variable can be used within @block code to see how many samples will come before the next @block call. It may also be valid in other contexts (though your code should handle invalid values in other contexts with grace).

- **tempo**
  Context: @block, @sample
  Usage: read-only
  The current project tempo, in "bpm". An example value would be 120.0.

- **play_state**
  Context: @block, @sample
  Usage: read-only
  The current playback state of REAPER (0=stopped, <0=error, 1=playing, 2=paused, 5=recording, 6=record paused).

- **play_position**
  Context: @block, @sample
  Usage: read-only
  The current playback position in REAPER (as of last @block), in seconds.

- **beat_position**
  Context: @block, @sample
  Usage: read-only The current playback position (as of last @block) in REAPER, in beats (beats = quarternotes in /4 time signatures).

Extended Functionality:

- **ext_noinit**
  Context: @init only
  Set this variable to 1.0 in your @init section if you do not wish for @init to be called (and variables/RAM to be possibly cleared) on every transport start.

- **ext_nodenorm**
  Context: @init only
  Set this variable to 1.0 in your @init section if you do not wish to have anti-denormal noise added to input.

- **reg00-reg99**
  Context: available everywhere
  Usage: read/write
  The 100 variables reg00, reg01, reg02, .. reg99 are shared across all effects and can be used for inter-effect communication. Their use should be documented in the effect descriptions to avoid collisions with other effects.

Delay Compensation (PDC):

- **pdc_delay**

Context: @block, @slider
Usage: read-write
The current delay added by the plug-in, in samples. Note that you shouldnt change this too often. This specifies the amount of the delay that should be compensated, however you need to set the pdc_bot_ch and pdc_top_ch below to tell JS which channels should be compensated.

- **pdc_bot_ch, pdc_top_ch**
  Context: @block, @slider
  Usage: read-write
  The channels that are delayed by pdc_delay. For example:

```
    pdc_bot_ch=0; pdc_top_ch=2; // delays the first two channels (spl0/spl1).
    pdc_bot_ch=2; pdc_top_ch=5; // delays channels spl2,spl3, and spl4.
```

  (this is provided so that channels you dont delay can be properly synchronized by the host).

- **pdc_midi**
  Context: @block, @slider
  Usage: read-write
  If set to 1.0, this will delay compensate MIDI as well as any specified audio channels.

Graphics and Mouse:
Note: these should all only be accessed from within the @gfx code section:

- **gfx_r, gfx_g, gfx_b, gfx_a**
  These represent the current red, green, blue, and alpha components used by drawing operations (0.0..1.0).

- **gfx_w, gfx_h**
  These are set to the current width and height of the framebuffer you will be drawing to.

- **gfx_x, gfx_y**

These set the "current" graphics position in x,y. You can set these yourselves, and many of the drawing functions update them as well.

- **gfx_mode**
  Set to 0 for default options. Add 1.0 for additive blend mode (if you wish to do subtractive, set gfx_a to negative and use gfx_mode as additive). Add 2.0 to disable source alpha for gfx_blit(). Add 4.0 to disable filtering for gfx_blit().

- **gfx_clear**
  If set to a value of >= 0.0, this will result in the framebuffer being cleared to that color. the color for this one is packed RGB (0..255), i.e. red+green*256+blue*65536. The default is 0. Or you could just clear manually using lice_rectto.

- **gfx_texth**
  Read-only, set to the height of text rendering. Eventually this may be read/write but for now it's readonly, and usually 8.

- **mouse_x, mouse_y, mouse_cap**
  mouse_x and mouse_y are set to the coordinates of the mouse. If the user is clicking and dragging the mouse within the UI/graphics area, mouse_cap is set to nonzero (1 if left mouse down, 2 if right - 4,8,16 will be set for control,shift, alt).

User defined functions, psuedo-objects

Note: this section only applies to REAPER 4.25+

JS now supports user defined functions, as well as some basic object style data access.

Functions can be defined anywhere in top level code (i.e. not within an existing () block, but before or after existing code), and in any section, although functions defined in @init can be used from other sections (whereas functions defined in other sections are local to those sections). Functions are not able to be called recursively -- this is enforced by functions only being able to call functions that are declared before the current function, and functions not being able to call themselves. Functions may have 0 to 40 parameters. To define a function, use the following syntax:

```
function getSampleRate()
(
  srate; // return srate
);

function mySine(x)
(
  // taylor approximation
  x - (x^3)/(3*2) + (x^5)/(5*4*3*2) - (x^7)/(7*6*5*4*3*2) +
(x^9)/(9*8*7*6*5*4*3*2);
);

function calculateSomething(x y)
(
  x += mySine(y);
  x/y;
);
```

Which would then be callable from other code, such as:

```
y = mySine($pi * 18000 / getSampleRate());
z = calculateSomething(1,2);
```

Note that the parameters for functions are private to the function, and will not affect global variables. If you need more private variables for a function, you can declare additional variables using a local() statement between the function declaration and the body of the function. Variables declared in the local() statement will be local to that function, and persist across calls of the function (though calls to a function from two different sections (such as @init and @sample) will have two different local states. Example:

```
function mySine(x) local(lastreq lastvalue)
(
  lastreq != x ? (
    lastreq = x; // save last input
    // taylor approximation
    lastvalue = x - (x^3)/(3*2) + (x^5)/(5*4*3*2) -
(x^7)/(7*6*5*4*3*2) + (x^9)/(9*8*7*6*5*4*3*2);
  );
```

```
      lastvalue; // result of function is cached value
    );
```

In the above example, mySine() will cache the last value used and not perform the calculation if the cached value is available. Note that the local variables are initialized to 0, which happens to work for this demonstration but if it was myCosine(), additional logic would be needed.

JS also supports relative namespaces on global variables, allowing for psuedo object style programming. Accessing the relative namespace is accomplished either by using a "this." prefix for variable/function names, or by using the instance() declaration in the function definition for variable names:

```
function set_foo(x) instance(foo)
(
  foo = x;
);
// or
function set_foo(x)
(
  this.foo = x;
);

whatever.set_foo(32); // whatever.foo = 32;
set_foo(32); // set_foo.foo = 32;

function test2()
(
  this.set_foo(32);
);
whatever.test2(); // whatever.foo = 32
```

Additionally functions can use the "this.." prefix for navigating up the namespace hierarchy, such as:

```
function set_par_foo(x)
(
  this..foo = x;
);
```

```
a.set_par_foo(1); // sets foo (global) to 1
a.b.set_par_foo(1); // sets  a.foo to 1
```