DreamMaker FX Cheatsheet

API Version: 1.5.1

Amplitude Modulator (aka tremelos)

`fx_amplitude_mod`

Amplitude modulators are the basic building blocks of tremelos and rhythmic effects. They essentially use an oscillator / waveform or an external control signal to vary the amplitude / volume of a signal.

Parameters

- *rate_hz* – *When using an internal oscillator, the "modulation" rate is oscillation (cycles per second). When in doubt, start with 1.0 (one cycle per second)*
- *depth* – *How much the volume is "modulated". A value of 0.0 is none at all and a value of 1.0 means full volume to zero volume.*
- *initial phase* – *The initial phase of the oscillator in degrees. When in doubt, use 0.0. This is useful when you want to have multiple oscillators running at different phases such as in harmonic tremelo where one may be at 0.0 and the other at 180.0.*
- *Modulation type* – *see Appendix for available waveforms (sine, square, triangle, random, pulse, etc.) as the modulation source.*
- *External modulator* – *Rather than using an internal modulator, you can also use an external audio source. Route audio to the .ext_mod_in audio to use it as the external modulator.*

Constructors

```
fx_amplitude_mod(float rate_hz,
                 float depth)
fx_amplitude_mod(float rate_hz,
                 float depth,
                 float initial_phase_deg,
                 OSC_TYPES modulation_type,
                 bool use_ext_modulator)
```

Modifiers

```
.set_depth(float depth)
.set_rate_hz(float rate_hz)
.set_lfo_type(OSC_TYPES) New!
.enable()
.bypass()
```

Audio nodes

```
.input
.output
.ext_mod_in
```

Control nodes

```
.depth
.rate_hz
```

Example

```
#include <dreammakerfx.h>

fx_amplitude_mod    mod1(1.0,     // Rate (Hz) is once per second
                         0.8,     // Depth (0.0->1.0)
                         0,       // Initial phase (degrees)
                         OSC_SINE,// Oscillator type is a sine wave
                         false);  // Don't use external LFO

void setup() {

  pedal.init();           // Initialize pedal

  // Route audio through effects
  pedal.route_audio(pedal.instr_in, mod1.input);
  pedal.route_audio(mod1.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();            // Run effects
}

void loop() {

  // Pot 0 changes the rate of the tremelo from 0 to 4Hz
  if (pedal.pot_0.has_changed()) {
    mod1.set_rate_hz(pedal.pot_0.val*4.0);
  }

  // Pot 1 changes the depth from 0.0 to 1.0
  if (pedal.pot_1.has_changed()) {
    mod1.set_depth(pedal.pot_1);
  }

   pedal.service(); // Run pedal service to take care of stuff
}
```

Fun things to try: use tap function to set rate, use a instrument input through a pitch shifter as the external modulator, use high modulation frequency like 440.0Hz, try a few in parallel running through filters with different initial phase values (to create harmonic tremelos).

DreamMaker FX Cheatsheet

Biquad filter (aka an audio filter)

Filters are a basic building block of so many audio effects. Filters allow certain frequencies to pass through and decrease the volume at other frequencies. A wah pedal is a filter that is "swept" across a range of frequencies based on foot position.

Parameters

- *frequency – This is the cutoff frequency or center frequency of the filter in Hertz.*
- *resonance – This is how quickly the filter "rolls off" – is it a gentle, wide filter or a tight narrow filter? A value of 1.0 is no resonance; > 1.0 is more resonant, < 1.0 is less resonant*
- *filter type – Filters come in lots of colors. Low-pass filters (LPF) cut higher frequencies. High-pass filters (HPF) cut lower frequencies. Band-pass filters (BPF) cut frequencies on both sides of the filter frequency. And notch filters cut the frequencies at the filter frequency and allow others to pass.*
- *Transition speed – When a new filter frequency or filter width is provided, the transition speed determines how quickly the filter will transition.*
- *filter gain (dB) – Some types of filters take a gain parameter as well. These are low and high shelves. The gain factor determines the gain at the "pass through" gains for shelving filters. A value of 0.0 (dB) is unity gain (no effect)*
- *Filter order – how many stages of the filter – higher is more extreme filtering effect.*

`fx_biquad_filter`

Constructors

```
fx_biquad_filter(float frequency,
                 float resonance,
                 BIQUAD_FILTER_TYPE filter_type)
```

New!

```
fx_biquad_filter(float frequency,
                 float resonance,
                 BIQUAD_FILTER_TYPE filter_type,
                 BIQUAD_FILTER_ORDER order)
fx_biquad_filter(float frequency,
                 float resonance,
                 float gain_db,
                 BIQUAD_FILTER_TYPE filter_type,
                 EFFECT_TRANSITION_SPEED trans_speed)
fx_biquad_filter(float frequency,
                 float resonance,
                 float gain_db,
                 BIQUAD_FILTER_TYPE filt_type,
                 EFFECT_TRANSITION_SPEED trans_speed,
                 BIQUAD_FILTER_ORDER order)
```

Modifiers

```
.set_freq(float freq)
.set_q(float q)
.set_gain(float gain)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
.ext_mod_in
```

Control nodes

```
.freq
.q
.gain
```

Example

```
/**
  * In this example, we'll create an auto-wah filter where we have an envelope tracker which tracks the volume
  * we're playing at and uses this to move the filter frequency.  This example uses both route_audio AND
  * route_control.  This is where the magic lies.
  */
#include <dreammakerfx.h>
fx_envelope_tracker   envy_tracky(10,     // 10 ms attack
                                  100,    // 100 ms release
                                  false); // not triggered

fx_biquad_filter  wah_filter(300.0,                 // 300 Hz starting frequency
                             FILTER_WIDTH_NARROW,   // Width of the filter is narrow
                             BIQUAD_TYPE_BPF);      // Type is bandpass

void setup() {
  pedal.init();           // Initialize pedal

  // Route audio through effects
  pedal.route_audio(pedal.instr_in, wah_filter.input);
  pedal.route_audio(wah_filter.output, pedal.amp_out);

  // Route audio to envelope tracker
  pedal.route_audio(pedal.instr_in, envy_tracky.input);

  // Route control from envelop tracker to filter frequency
  pedal.route_control(envy_tracky.envelope, wah_filter.freq, 1000.0, 300.0); // range 0->1 to 300->300+1000

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();            // Run effects
}

void loop() {
  pedal.service(); // Run pedal service to take care of stuff
}
```

Fun things to try: hook up a filter to the envelope tracker to create an auto-wah, run a clipper through a filter to get various tube sounds, hook up an oscillator to the filter frequency to create a rhythmic filter sweep, run filters through amplitude modulators to create harmonic modulators.

DreamMaker FX Cheatsheet

API Version: 1.5.1

Compressor / Sustainer

`fx_compressor`

Think of a compressor as a small robot that controls a volume knob based on how loud you're playing. When you strike a loud chord, the robot immediately turns the volume down and as the chord rings out, the robot turns the volume up progressively, so it sounds like you're just sustaining the chord. Instead of dying off, it sounds steady for a few seconds as the robot is turning up the volume. Compressors are used a lot with acoustic instruments and vocals but also with electric guitars too. A common in country music is running a Telecaster through a compressor.

Parameters

- *threshold* – *Where the robot starts turning down the volume. This value is in decibels so a good place to start is between -60.0 and -30.0*
- *ratio* – *How aggressively the robot will turn down the volume when the input exceeds the threshold. Values from 2-16 create a softer effect. A very high value of 100.0 creates a hard ceiling.*
- *attack* – *Time in milliseconds for robot to respond when a note exceeds the threshold. Setting this to 20-30 will allow a bit of a peak to sneak through.*
- *release* – *how long before the robot stops controlling volume after volume goes below threshold*
- *output gain* – *output volume (from 1.0 and up)*

Constructors

```
fx_compressor(float thresh,
              float ratio,
              float attack,
              float release,
              float gain_out)
```

Modifiers

```
.set_threshold(float threshold)
.set_ratio(float ratio)
.set_attack(float attack)
.set_release(float release)
.set_output_gain(float gain_out)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
```

Control nodes

```
.threshold
.ratio
.attack
.release
.out_gain
```

Example

```
#include <dreammakerfx.h>

fx_compressor compressor_1(-30.0,    // Initial threshold in dB
                           8,        // Initial ratio (1:8)
                           10.0,     // Attack (10ms)
                           100.0,    // Release (100ms)
                           2.0);     // Initial output gain (2x);

void setup() {
  pedal.init();  // Initialize pedal

  // Route audio through effects
  pedal.route_audio(pedal.instr_in, compressor_1.input);
  pedal.route_audio(compressor_1.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {
  // Run pedal service to take care of stuff
  pedal.service();

  if (pedal.pot_0.has_changed()) {.   // Pot 0 sets threshold from -20dB to -70dB
    compressor_1.set_threshold(-20 – (50.0 * pedal.pot_0.val);
  }
  if (pedal.pot_1.has_changed()) {    // Pot 1 sets compression ration from 1:1 to 40:1
    compressor_1.set_ratio(1.0+ (40.0 * pedal.pot_1.val));
  }
 if (pedal.pot_2.has_changed()) {    // Pot 2 sets output gain from 1.0 to 6.0
    compressor_1.set_output_gain(1.0 + pedal.pot_2.val*5.0);
  }
}
```

Fun things to try: Add a compressor on either side of a clipper to create more dynamics, run two compressors through a LPF and HPF to create a multi-band compressor (where low end and high end are compressed independently), vary compressor parameters with an LFO to get some wild sounds.

Delay (aka echoes!!)

A delay effect is basically an echo machine. Unlike other delay pedals, we have a massive amount of delay memory so you can create delays that are several seconds long. Also, this delay block allows you to add your own effects to the "feedback" path of the echo so each echo can run though an effects chain. Put a pitch shifter in here and each echo changes pitch. Add a phase shifter and each echo gets progressively "phasey". Put another echo effect in there and create effects like the movie Inception.

Parameters

- *delay length (millisecond) – the length of the echo in milliseconds (1000.0 milliseconds = 1 second). For the advanced constructor, the delay_len_max_ms determines the total memory allocated for this delay and will be the max length. In the basic constructor, the initial length is also the maximum delay length.*
- *feedback – how much of the output is feedback to the input. A value of 0.0 will product a single delay. A value of 1.0 will produce endless echoes. 0.5-0.7 is a nice decaying echo.*
- *dry mix – mix of the clean signal (0.0 to 1.0)*
- *wet mix – mix of the delayed/echo signal (0.0 to 1.0)*
- *Enable external fx send/receive – whether or not to enable the fx send / receive loop.*

fx_delay

Constructors

```
fx_delay(float delay_len_ms,
         float feedback)
fx_delay(float delay_len_ms,          Updated!
         float delay_len_max_ms,
         float feedback,
         float mix_dry,
         float mix_wet,
         bool enable_ext_fx)
```

Modifiers

```
.set_length_ms(float len_ms)
.set_feedback(float feedback)
.set_dry_mix(float dry_mix)  New!
.set_wet_mix(float wet_mix)  New!
.enable()
.bypass()
```

Audio nodes

```
.input
.output
.fx_send
.fx_receive
```

Control nodes

```
.length_ms
.feedback
.dry_mix  New!
.wet_mix  New!
```

Example

```
/**
  * this example creates a delay and places a low-pass dampening filter in the feedback loop so each echo
  * gets darker and darker.  Ran out of room for parameters but you get the idea by now ☺
  */
#include <dreammakerfx.h>

fx_delay        delay_1(1000.0, // Initial delay length of 1 second / 1000ms
                        5000.0, // Max delay of 5 seconds
                        0.7,    // Initial feedback value of 0.7
                        1.0,    // Clean mix
                        0.7,    // Delay / echo mix
                        true);  // Enable fx send/receive loop

fx_biquad_filter  fb_filt(1200.0,              // 1200 Hz starting frequency
                         FILTER_WIDTH_NORMAL,   // Width of the filter is narrow
                         BIQUAD_TYPE_LPF);      // Type is low-pass

void setup() {
  pedal.init();  // Initialize pedal

  // Route audio through effects
  pedal.route_audio(pedal.instr_in, delay_1.input);
  pedal.route_audio(delay_1.output, pedal.amp_out);

  // Route filter through delay fx send/receive loop
  pedal.route_audio(delay_1.fx_send, fb_filt.input);
  pedal.route_audio(fb_filt.output, delay_1.fx_receive);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();           // Run effects
}

void loop() {
  pedal.service(); // Run pedal service to take care of stuff
}
```

Fun things to try: Create a set of delays in parallel with lengths (1000ms, 750ms, 333ms) to create cool rhythmic echoes, create elaborate effects chains in the delay's feedback loop, add delays into the feedback fx send/receive loop of the delay, control a filter from a delayed version of a signal

Destructor (aka distortion!!)

fx_destructor

The destructor uses a variety of functions to destruct waveforms. These come in the form of warm, tube-like distortions to bit and sample rate crushers. Complex amplifier modelers use multiple stages of polynomial clippers and filters to create authentic replicas of tube amplifiers.

Parameters

- *type* – *there are currently a number of available destructors that are described in more detail on the next page.*
- *param_1* – *each destructor takes a different set of arguments which are outlined on the next page.*
- *param_2* – *each destructor takes a different set of arguments which are outlined on the next page.*
- *output_gain* – *a gain stage has been added to each destructor model to control gain of the output*

Constructors

```
fx_destructor(float param_1,
              DESTUCTOR_TYPE type)     New!
fx_destructor(float param_1,
              float param_2,
              DESTUCTOR_TYPE type)
fx_destructor(float param_1,
              float param_2,
              float output_gain,       New!
              DESTUCTOR_TYPE type)
```

Modifiers

```
.enable()
.bypass()
.set_param_1(float new_param_1)
.set_param_2(float new_param_2)
.set_output_gain(float new_gain)  New!

 // Aliases of set_param for clipping functions
.set_clipping_threshold(float new_clip) New!
.set_input_drive(float new_drive)       New!
```

Audio nodes

```
.input
.output
```

Control nodes

```
.param_1
.param_2
.output_gain
```

Example

```
#include <dreammakerfx.h>
fx_clipper    tube_distort(0.1,     // Poly smooth step param 1 is clipping threshold (0.0 -> 1.0)
                           8.0,     // Poly smooth step param 2 is gain before clipping function
                           SMOOTH_CLIP);  // clipping polynomial

fx_biquad_filter  warm_filter(900.0,               // 900 Hz starting frequency
                              FILTER_WIDTH_NORMAL,    // Width of the filter is normal (Q=0.7071)
                              BIQUAD_TYPE_LPF);       // Type is lowpass

void setup() {
  pedal.init();  // Initialize pedal

  // Route audio through effects
  pedal.route_audio(pedal.instr_in, distort.input);
  pedal.route_audio(distort.output, warm_filter.input);
  pedal.route_audio(warm_filter.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();           // Run effects
}

void loop() {
  pedal.service(); // Run pedal service to take care of stuff
  if (pedal.pot_0.has_changed()) {.   // Pot 0 sets drive from 1 to 64.0
    tube_distort.set_input_drive(1.0 + pedal.pot_0.val*63.0);
  }
  if (pedal.pot_1.has_changed()) {    // Pot 1 sets clipping from 0.2 to 0.01
    tube_distort.set_threshold(0.01 + (1.0-pedal.pot_1.val)*0.2);
  }
 if (pedal.pot_2.has_changed()) {    // Pot 2 sets tone from 400Hz to 1400Hz
    warm_filter.set_freq(400.0 + pedal.pot_2.val*1000.0);
  }
}
```

Fun things to try: Run a series of destructors in series to create more exotic sounding distortions, modulate the filters using oscillators, run several destructors through the slicer to create cool rhythmic distortions (for Tim), run destructors through a bank of band-pass filters at different freqs.

Envelope tracker

`fx_envelope_tracker`

An envelope tracker creates a control signal based on the amplitude of the signal running into it. Envelope trackers are the basis of auto-wah and envelope filter pedals in which the envelope control signal controls the center frequency for a band-pass filter. The envelope trackers can be used to control any other control node for other effects so it opens up lots of creative options. This envelope tracker also has a "trigger" mode which is useful if you want each pluck of an instrument to experience a full sweep.

Parameters

- *attack speed* – *How quickly/slowly the control signal increases when a new note event comes through. This is measured in milliseconds so a value of 50 is quick and 500 (half a second) is slow*
- *decay speed* – *How quickly/slowly the control signal decays back towards zero.*
- *triggered* – *When triggering is enabled, the control volume will rapidly decay to zero when a new event (like a string pluck) is detected. This creates a more pronounced effect when tracking instrument playing. This should be set to* ***true*** *or* ***false****.*
- *scale* - *Used to set the scale of the output envelope*
- *offset* – *Used to set the offset of the output envelope*

Constructors

```
fx_envelope_tracker(float attack_speed_ms,
                    float decay_speed_ms,
                    bool triggered)
fx_envelope_tracker(float attack_speed_ms,   New!
                    float decay_speed_ms,
                    bool triggered,
                    float ctrl_scale,
                    float ctrl_offset)
```

Modifiers

```
.set_attack_speed_ms(float attack_speed_ms)
.set_decay_speed_ms(float decay_speed_ms)
.set_env_scale(float scale)     New!
.set_env_offset(float offset)   New!
.enable()
.bypass()
```

Audio nodes

```
.input
```

Control nodes

```
.attach_speed_ms
.decay_speed_ms
.scale
.offset
.envelope   [output]
```

Example

```
/**
  * this example creates a swell effect where an envelope tracker with a slow attack rate is used to control
  * gain (volume).  The triggering is enabled such that each note gets its own "swell"
  */
#include <dreammakerfx.h>

fx_envelope_tracker      tracker_1(500.0,         // 500 ms attack
                                   800.0,         // 800 ms decay
                                   true,          // enable triggering
                                   2.0,           // scale of control signal
                                   0.0);          // offset of control signal

fx_gain                 vol_swell (1.0);          // Gain element with initial gain/volume of 1.0

void setup() {
  pedal.init();  // Initialize pedal

  // Route audio through volume control
  pedal.route_audio(pedal.instr_in, vol_swell.input);
  pedal.route_audio(vol_swell.output, pedal.amp_out);

  // Route audio to envelope tracker
  pedal.route_audio(pedal.instr_in, tracker_1.input);

  // Route control from envelop tracker to volume control
  pedal.route_control(envy_tracky.envelope, vol_swell.gain); // range 0->1 to 0->2

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {
  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Start attaching the envelope tracker to random parameters of other effects just to see what happens, create a wah pedal with lots of different types of band-pass filters that move in different directions.

Gain control (for controlling volume)

fx_gain

Gain control is volume control. This gain controller uses "linear" gain values so a value of 0.0 means the volume is all the way down (.e. "none shall pass!"). A value of 1.0 means it's just passing things through unaffected. Values greater than 1.0 will increase the volume and values below 1.0 will decrease the volume.

Parameters

- ***gain*** *– Linear volume adjustment as described above. Less than 1.0 is decreasing volume, greater than 1.0 is increasing volume, 0.0 is no signal at all, and 1.0 is passing things through*
- ***gain (dB)*** *- Logarithmic volume adjustment in decibels (dB). Google it to learn more.*

Constructors

`fx_gain(float gain_val)`

Modifiers

```
.set_gain(float new_gain)
.set_gain_db(float new_gain_db)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
```

Control nodes

```
.gain
```

Example

```
/**
  * this example creates a swell effect where an envelope tracker with a slow attack rate is used to control
  * gain (volume).  The triggering is enabled such that each note gets its own "swell"
  */
#include <dreammakerfx.h>

fx_envelope_tracker      tracker_1(500.0,         // 500 ms attack
                                   800.0,         // 800 ms decay
                                   true);         // enable triggering


fx_gain                  vol_swell(1.0);          // Gain element with initial gain/volume of 1.0

void setup() {
  pedal.init();  // Initialize pedal

  // Route audio through volume control
  pedal.route_audio(pedal.instr_in, vol_swell.input);
  pedal.route_audio(vol_swell.output, pedal.amp_out);

  // Route audio to envelope tracker
  pedal.route_audio(pedal.instr_in, tracker_1.input);

  // Route control from envelop tracker to volume control
  pedal.route_control(envy_tracky.envelope, vol_swell.gain, 2.0, 0.0); // range 0->1 to 0->2

  // Use left footswitch/LED to bypass effect
  pedal.add_bypass_button(FOOTSWITCH_LEFT);

  pedal.run();  // Run effects
}

void loop() {
  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Use gain to mix together lots of different signals to create really interesting layered effects, connect gain to LFOs to create effects that fade in and out

DreamMaker FX Cheatsheet

Polyphonic Instrument Synth New!

`fx_looper`

The instrument synth is capable of tracking multiple played strings on a guitar and generating synth notes that track the played notes. The instrument synth tracks both frequency and volume so it will track string bends and the natural volume decay of a plucked note. The instrument synth has an 8-note polyphone (can play 8 notes at once) and has a built in envelope filter and FM oscillator for each voice.

The instrument synth is hard-wired to detect notes at the instrument input so it can't be used in the middle of an effect chain. It doesn't have an input audio node, just an output. The instrument synth output only contains synthesized audio.

Parameters

- *oscillator type – The type of oscillator*
- *FM modulator oscillator type – Type of modulator used for FM synthesis (optional but cool way to create richer sounds)*
- *FM mod depth – amount of freq modulation to apply*
- *Freq ratio – the ratio of the note detected and the note played. 1.0 will play the same frequency detected, 2.0 will play an octave up, etc.*
- *FM mod freq ratio – the ratio of freq mod oscillator to oscillator*
- *Attack (ms) – the attack time (note swell) in ms*
- *Filter resonance – amount of resonance on envelope filter*
- *Filter response – how far the envelope filter moves on a new notes*

Constructors

```
fx_instrument_synth(OSC_TYPES osc_type,
                    float attack_ms,
                    float filter_resonance,
                    float filter_response)
fx_instrument_synth(OSC_TYPES osc_type,
                    OSC_TYPES fm_mod_osc_type,
                    float fm_mod_depth,
                    float freq_ratio,
                    float freq_ratio_fm_mod,
                    float attack_ms,
                    float filter_resonance,
                    float filter_response)
```

Modifiers

```
.set_freq_ratio(float ratio)
.set_fm_mod_ratio(float fm_mod_ratio)
.set_fm_mod_depth(float depth)
.set_attack_ms(float attack_ms)
.set_filter_resonance(float resonance)
.set_filter_response(float response)
.set_oscillator_type(OSC_TYPES new_type)
.set_oscillator_type_fm_mod(OSC_TYPES new_type)
.enable()
.bypass()
```

Audio nodes

```
.output
```

Control nodes

```
.attack_ms
.freq_ratio
.fm_mod_freq_ratio
.fm_mod_depth
.resonance
.response
```

Example

```
/**
  * this example creates a polyphonic instrument synth using an frequency modulated oscillator bank!
  */
#include <dreammakerfx.h>

fx_instrument_synth synth(OSC_TRIANGLE, // Primary oscillator
                          OSC_RAMP_NEG,  // FM mod oscillator
                          0.1,           // FM mod depth
                          1.0,           // Freq ratio (ratio of instrument frequency to synth frequency)
                          2.0,           // FM mod frequency ratio (1.0 = same frequency as primary oscillator)
                          250.0,         // Attack time in milliseconds
                          3.0,           // Filter resonance
                          0.6);          // Filter response (0.0 to 1.0)

void setup() {
  pedal.init();  // Initialize pedal
  pedal.route_audio(synth.output, pedal.amp_out);
  pedal.run();  // Run effects
}

void loop() {
  if (pedal.pot_left.has_changed()) {
     synth.set_fm_mod_depth(pedal.pot_left.val);    // Left pot sets mod depth
  }

  if (pedal.pot_center.has_changed()) {
     synth.set_attack_ms(25.0 + pedal.pot_center.val*975.0); // Center pot sets attack time  }

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: create multiple instrument synths with different oscillator configurations and run in parallel, run the output through various effects

Looper (for making loopers)

A looper is used to record a block of audio that is then looped (i.e. played back on a loop). It's a fun tool for laying down layers of looping audio. This looper can change the playback rate so a recorded loop can be played back slower or faster. The playback rate can even be negative so the loop that was just recorded can be played back in reverse. Like the delay effect, it also has an effects send/receive loop that can be optionally used before audio enters the recorded loop. So you could pitch shift the audio as it's being recorded in the loop for example and then the audio that will be played back on loop will be an octave lower. The looper has a loop length output control channel that can be used to synchronize other time-based effects like delays to the loop length

Parameters

- *playback rate – The rate at which the loop is played back. 1.0 is the same rate as it was recorded at. 0.5 would be half speed / one octave down. 2.0 would be double speed / one octave up. A negative value just means the loop will be played in reverse at that same rate.*
- *Loop length – One a loop has been recorded, its length can be used as a control parameter to control the length of a delay, for example. So you could run a looper through a delay effect that will be running at some 8 echoes per loop, for example.*

`fx_looper`

Constructors

```
fx_looper(float looper_dry_mix,
          float looper_loop_mix,
          float looper_max_length_seconds,
          bool looper_enable_loop_preprocessing)
```

Modifiers

```
.start_loop_recording()
.stop_loop_recording()
.set_playback_rate(float playback_rate)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
.preproc_send
.preproc_receive
```

Control nodes

```
.start
.stop
.playback_rate
.dry_mix
.loop_mix
.loop_length_seconds <- this one is an output!
```

Example

```
/**
  * This is a standard looper that
  */
#include <dreammakerfx.h>
fx_looper   loopy( 0.0,    // No dry mix
                   0.8,    // Looped audio mix
                   30,     // Max loop length in seconds
                   false); // Disable FX processing as audio enters the loop

void setup() {
  pedal.init();
  pedal.route_audio(pedal.instr_in, loopy.input);
  pedal.route_audio(loopy.output, pedal.amp_out);
  pedal.run();
}

bool looper_active = false;
void loop() {
  // When left foot switch held down, record a loop (if one not playing) or stop the loop that is currently
  // playing and get ready for another recording
  if (pedal.button_pressed(FOOTSWITCH_RIGHT, true)) {
    if (!looper_active) {
      loopy.start_loop_recording();
      looper_active = true;
    } else {
      loopy.stop_loop_playback();
      looper_active = false;
    }
  }
  if (pedal.button_released(FOOTSWITCH_RIGHT, true)) {
    if (looper_active) {
      loopy.stop_loop_recording();
    }
  }
  pedal.service();
}
```

Fun things to try: Create multiple loops in parallel and allow the user to layer several loops, connect a delay or an amplitude modulator to the loop length, run looper FX send/receive through pitch shifter so loop is an octave lower.

DreamMaker FX Cheatsheet

API Version: 1.5.1

Mixers

These are little utilities for mixing (aka adding) signals together.

Unlike other modules, the mixer constructors don't take any arguments so they're declared a bit differently (see example below).

Add fx_gain blocks to control the gain for individual channels.

And I guess that's all there is to say about the mixer. Not because the mixer isn't a great building block and stuff. I mean it is. But it's simplicity is the magic of it all.

Parameters

none, just none.

`fx_mixer`

Constructors

`fx_mixer_2`
`fx_mixer_3`
`fx_mixer_4`

Modifiers

`(none)`

Audio nodes

`.input_1`
`.input_2`
`.input_3` `(on 3 and 4 channel mixer)`
`.input_4` `(on 4 channel mixer)`
`.output`

Control nodes

`(none)`

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

fx_biquad_filter  filter_1(300.0,                 // 300 Hz starting frequency
                           FILTER_WIDTH_NARROW,   // Width of the filter is narrow
                           BIQUAD_TYPE_BPF);      // Type is bandpass
fx_biquad_filter  filter_2(900.0,                 // 300 Hz starting frequency
                           FILTER_WIDTH_NARROW,   // Width of the filter is narrow
                           BIQUAD_TYPE_BPF);      // Type is bandpass


fx_mixer_2 mixy;

void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, filter_1.input);
  pedal.route_audio(filter_1.output, mixy.input_1);

  pedal.route_audio(pedal.instr_in, filter_2.input);
  pedal.route_audio(filter_2.output, mixy.input_2);

  pedal.route_audio(mixy.output, pedal.amp_out);

  pedal.run();  // Run effects
}

void loop() {
  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Create a wah pedal with multiple band-pass filters, combine multiple effects chains in parallel, combine multiple pitch shifters

Oscillators (for sound or control)

`fx_looper`

These are basically waveform generators that can be used as audio sources (for creating tones, etc.) or control sources (for controlling other effects).

When using this as a control source (such as an LFO), use the .value control node.

Parameters

- *type – This is the type of waveform that is generated (sine, square wave, triangle wave, ramp, etc.). See the appendix for a full list of options.*
- *frequency – This frequency of the waveform in Hertz (Hz).*
- *amplitude– the (linear) volume of the waveform. Should be between 0.0 and 1.0 (full volume) when used to generate audio. And it can be any value when generating control signals.*
- *offset – When used in control applications, the offset. For example, if the offset is set to 1000 and the amplitude is set to 200, the waveform will oscillate between 800 and 1200. If used as an audio source, the offset should always be set to zero.*

Constructors

```
fx_oscillator(OSC_TYPES osc_type,
              float freq,
              float amplitude)
fx_oscillator(OSC_TYPES  osc_type,
              float freq,
              float amplitude,
              float offset)
```

Modifiers

```
.set_frequency(float freq)
.set_amplitude(float amplitude)
.set_oscillator_type(OSC_TYPES new_type)
```
New!
```
.enable()
.bypass()
```

Audio nodes

```
.output
```

Control nodes

```
.freq
.amplitude
.offset
.value [output]
```

Example

```
/**
  */
#include <dreammakerfx.h>


void setup() {
  pedal.init();  // Initialize pedal

  pedal.run();  // Run effects
}

void loop() {
  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Create multiple loops in parallel and allow the user to layer several loops, connect a delay or an amplitude modulator to the loop length, run looper FX send/receive through pitch shifter so loop is an octave lower.

Phase Shifter

`fx_phase_shifter`

These create a series of phase shifts connected to an LFO. Phase shifters were pioneered probably in the 1960s and 1970s and create a familiar effect. Search for phase shifter pedals on youtube for some examples of what they sound like.

Phase shifters create a particularly pronounced effect when the audio running through them has higher frequency components. They sound pretty interesting after a distortion or a synth sound.

Parameters

- depth - this is the depth of the effect and should vary from 0.0 (no effect) to 1.0 (full effect).
- rate (hz) – this is the rate at which the phase is shifted. A value of 1.0 is once per second (1 Hertz). Higher value produce more rapid oscillations and values between 0.0 and 1.0 produce slower oscillations
- feedback – this is the amount of feedback that is "fed back" into the effect itself. Feedback creates a more pronounced effect. Feedback should vary between -1.0 (full negative feedback) and 1.0 (full positive feedback). A value of 0.0 means there is no feedback.

Constructors

```
fx_phase_shifter(float rate_hz,
                 float depth,
                 float feedback)
```

Modifiers

```
.set_rate_hz(float rate_hz)
.set_depth(float depth)
.set_feedback(float feedback)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
```

Control nodes

```
.depth
.rate_hz
.feedback
```

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

fx_phase_shifter  shifter_1(1.0,                 // Initial rate of 1Hz
                            0.7,                 // Initial depth
                            0.3);                 // Initial feedback value


void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, shifter_1.input);
  pedal.route_audio(shifter_1, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect
  pedal.add_tap_interval_button(FOOTSWITCH_LEFT, true); // Use right footswitch to tap in rate

  pedal.run();  // Run effects
}

void loop() {

  // If a new tap interval has been tapped in, update the phase shifter
  if (pedal.new_tap_interval()) {
     shifter_1.set_rate_hz(pedal.get_tap_freq_hz());
   }

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Add a phase shifter after a distortion or a synth; add a phase shifter into the feedback send/receive loop of a delay; create multiple phase shifters in parallel set to different rates.

Pitch shifter

These shift the pitch of the audio by a relative amount. A value of 2.0 will shift the pitch up one octave. A value of 0.5 will shift the pitch down an octave. A value of 1.5 will shift the pitch up a fifth, and 0.75 will shift it down a fifth. And 1.189207 will shift it up a minor third.

Follow this link for a list of pitch shift values for other common intervals: https://en.wikipedia.org/wiki/Equal_temperament#Comparison_with_Just_Intonation

Note that pitch shifters are somewhat processor intensive. At the moment, the platform can support two pitch shifters.

Parameters

- pitch shift – the amount of relative pitch shift to apply as described above

fx_pitch_shift

Constructors

`fx_pitch_shift(float pitch_shift_freq)`

Modifiers

```
.set_freq_shift(float freq_shift)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
```

Control nodes

```
.freq_shift
```

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

fx_pitch_shift.   shifty(1.5);                 // Pitch shift up one fifth


void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, shifty.input);
  pedal.route_audio(shifty.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Add a pitch shifter in feedback path of a delay pedal; run a distortion through a pitch shifter (either before or after the effect) and mix it into a clean effect

Ring Modulator

`fx_ring_mod`

One of the weirder effects out there, a ring modulator multiplies the incoming audio signal by a sine wave at a certain carrier frequency (typically between 10.0 and 1000.0). This shifts the harmonics creating some very "interesting" sounds.

At full depth, the effect can be very bizarre, but many musicians have tastefully used these in more subtle ways.

Parameters

- frequency – the "carrier" frequency of the effect – best between 10.0 and 1000.0 but things get interesting beyond 1000.0 (as they always do).
- Depth – the depth of the audio effect (0.0 is no effect and 1.0 is full effect.

Constructors

```
fx_ring_mod(float ring_mod_freq,
            float ring_mod_depth)
```

Modifiers

```
.set_freq(float new_freq)
.set_depth(float new_depth)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
```

Control nodes

```
.freq
.depth
```

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

fx_ring_mod        ring_mod_1(200.0,     // Set carrier frequency to 200Hz
                              1.0);      // Set to full depth
fx_ring_mod        ring_mod_2(500.0,     // Set carrier frequency to 500Hz
                              1.0);      // Set to full depth

fx_mixer_2. mixy;
void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, ring_mod_1.input);
  pedal.route_audio(ring_mod_1.output, mixy.input_1);
  pedal.route_audio(pedal.instr_in, ring_mod_2.input);
  pedal.route_audio(ring_mod_2.output, mixy.input_2);

  pedal.route_audio(mixy.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {

 if (pedal.pot_0.has_changed()) {.   // Pot 0 sets threshold from -20dB to -70dB
     ring_mod_1.set_freq(250*pedal.pot_0.val);
     ring_mod_2.set_freq(500*pedal.pot_0.val);
 }

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Connect the frequency parameter to an envelope filter, add a ring modulator in the feedback path of a looper, run multiple ring modulators in series at frequency multiples (220, 440, etc.),

DreamMaker FX Cheatsheet

API Version: 1.5.1

Octave generator

`fx_octave`

This one emulates the Boss OC-3 and generates three synthesized signals at the current note, one octave down and two octaves down.

Each of the three synthesized signals have their own audio nodes so they can be independently processed. Alternatively, the .output node contains a mix of all signals.

Parameters

- Type – Type of oscillator – can be square, ramp or triangle. See Appendix for oscillator types.
- Clean mix – The mix of the clean signal (0.0 to 1.0)
- octave 0 mix – the mix of a synth signal at the current note being played (0.0 to 1.0)
- octave 1 mix – the mix of a synth signal one octave below the note being played (0.0 to 1.0)
- octave 2 mix – this mix of a synth signal two octaves below the note being played (0.0 to 1.0)

Constructors

```
fx_octave(OSC_TYPES type,
          float clean_mix,
          float oct_0_mix,
          float oct_1_mix,
          float oct_2_mix)
```

Modifiers

```
.set_clean_mix(float clean_mix)
.set_oct_0_mix(float oct_0_mix)
.set_oct_1_mix(float oct_1_mix)
.set_oct_2_mix(float oct_2_mix)
.enable()
.bypass()
```

Audio nodes

```
.input
.output // Everything mixed together
.oct_0_output
.oct_1_output
.oct_2_output
```

Control nodes

```
.clean_mix
.oct_0_mix
.oct_1_mix
.oct_2_mix
```

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

fx_octave         octave(OSC_TRI, // Triangle wave synth
                         0.7,    // Clean mix
                         0.2,    // Octave 0 mix
                         0.4,    // Octave 1 down mix
                         0.7,    // Octave 2 down mix


void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, octave.input);
  pedal.route_audio(octave.output, pedal.amp_out);

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: To Do

DreamMaker FX Cheatsheet

API Version: 1.5.1

Slicer

`fx_slicer`

Think of a slicer like a robot at a mixer that is send the incoming audio through multiple effects in a time sequenced manner. For example, if 2 channels are used and the period is set to 1000 ms (or 1 second), for the first half second, the audio is routed to output_1 and for the second half second, the audio is routed to output_2. These two outputs can run through different effects so the result is a cool, rhythmic effect.

Parameters

- Period (ms) – Type of oscillator – can be square, ramp or triangle. See Appendix for oscillator types.
- Channels – The total number of output channels to sequence through in the period amount of time

Constructors

```
 fx_slicer(float period_ms,
          int32_t channels)
```

Modifiers

```
.set_period_ms(float period)
.enable()
.bypass()
```

Audio nodes

```
.input
.output_1
.output_2
.output_3
.output_4
.output_5
.output_6
.output_7
.output_8
```

Control nodes

```
.clean_mix
.oct_0_mix
.oct_1_mix
.oct_2_mix
```

Example

```
/**
  * Using a mixer to mix audio running through a pair of band-pass filters together
*/
#include <dreammakerfx.h>

// To do


void setup() {
  pedal.init();  // Initialize pedal

 /// To do

  pedal.add_bypass_button(FOOTSWITCH_LEFT); // Use left footswitch/LED to bypass effect

  pedal.run();  // Run effects
}

void loop() {

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: To Do

DreamMaker FX Cheatsheet

Variable delay (aka flangers and choruses!)

A variable delay is a short delay line with a varying length where the delay is controlled by a low-frequency oscillator. Variable delays are used to create chorus and flanger effects but can also be used to create vibratos, rotating speaker effects and more.

Parameters

- Rate (hz) – The rate of the oscillator in Hertz (cycles per second)
- Depth – The depth of the shift in delay (resulting in a subtle increase / decrease in frequency) – should be from 0.0 to 1.0
- Feedback – Amount of output that is fed back to input – this increases the effect. Feedback can be from -1.0 to 1.0. Negative feedback creates interesting effets
- Mod type – the type of oscillator used to control the delay line. Flangers typically use triangle wave, choruses use the random wave.
- Buffer size ms – For advanced users, this is the size of the buffer in milliseconds.
- Ext mod – whether or not to use an external signal (fed in via ext_mod_in) rather than one of the default LFOs.

`fx_variable_delay`

Constructors

```
fx_variable_delay(float rate_hz,
                  float depth,
                  float feedback,
                  OSC_TYPES mod_type)
fx_variable_delay(float rate_hz,
                  float depth,
                  float feedback,
                  float buf_size_ms,
                  float mix_clean,
                  OSC_TYPES mod_type,
                  bool ext_mod )
```

Modifiers

```
.set_depth(float depth)
.set_rate_hz(float rate_hz)
.set_feedback(float feedback)
.enable()
.bypass()
```

Audio nodes

```
.input
.output
.ext_mod_in.     // external modulation signal
.modulated_out.  // Just modulated signal (no clean)
```

Control nodes

```
.depth
.rate_hz
.feedabck
```

Example

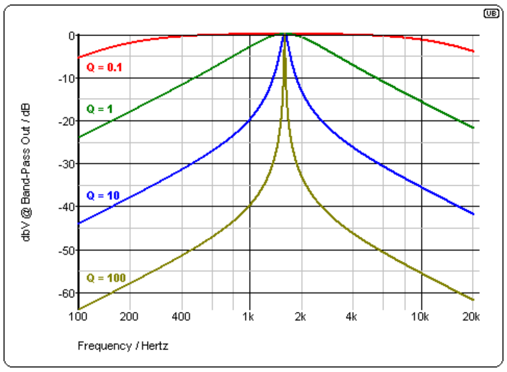
```
/**
*/
#include <dreammakerfx.h>

fx_variable_delay flangey(1.0,            // Initial oscillator rate of 1Hz (1 cycle / second)
                          0.5,            // Initial depth of 0.5
                          0.4,            // Initial feedback of 0.4
                          OSC_TRIANGLE);  // Use a triangle oscillator
void setup() {
  pedal.init();  // Initialize pedal

  pedal.route_audio(pedal.instr_in, flangey.input);
  pedal.route_audio(flangey.output, pedal.amp_out);

  // left footswitch is bypass
  pedal.add_bypass_button(FOOTSWITCH_LEFT);

  pedal.run();  // Run effects
}

void loop() {

  // Left pot controls depth of the effect
  if (pedal.pot_left.has_changed()) {
    flangey.set_depth(pedal.pot_left.val);
  }

  // Right pot controls the feedback (-1.0 to 1.0)
  if (pedal.pot_right.has_changed()) {
   flangey.set_feedback(1.0 - pedal.pot_right.val*2.0);
  }

  // Run pedal service to take care of stuff
  pedal.service();
}
```

Fun things to try: Variable delays can be used to create all sorts of pitch modulations; put one in the feedback path of a delay pedal, run several in parallel or series to create exotic effects.

Biquad filter (aka an audio filter) continued

`fx_biquad_filter :: resonance`

Resonance controls the resonance and width of the filter. A filter with higher resonance will filter out more of the frequencies outside of the center frequency of the filter whereas filters with less resonance have a slower "roll-off". A resonance value of 1.0 corresponds to a Q factor of 0.7071.



`fx_biquad_filter :: BIQUAD_FILTER_TYPE`

BIQUAD_TYPE_LPF
Low pass filter – cuts high frequencies, allows low frequencies to pass

BIQUAD_TYPE_HPF
High pass filter – cuts low frequencies, allows high frequencies to pass

BIQUAD_TYPE_BPF
Band pass filter – cuts low and high frequencies outside of center/cutoff frequency of filter (like wah pedal)

BIQUAD_TYPE_NOTCH
Notch filter – cuts a specific frequency and lets other frequencies pass (basically opposite of a band-pass filter)

BIQUAD_TYPE_PEAKING
Peaking filter – increases or decreases gain at a specific frequency (like those used in parametric filters)

BIQUAD_TYPE_L_SHELF
Low shelf filter – lets all frequencies pass but increases or reduces frequencies below the shelf frequency by specified amount

BIQUAD_TYPE_H_SHELF
High shelf filter – lets all frequencies pass but increases or reduces frequencies above the shelf frequency by specified amount

`fx_biquad_filter :: BIQUAD_FILTER_ORDER`

BIQUAD_ORDER_2
BIQUAD_ORDER_4
BIQUAD_ORDER_6
BIQUAD_ORDER_8

Higher "order" filters have a more extreme filtering effect. When in doubt, start with second order (BIQUAD_ORDER_2) and work your way up from there.

DreamMaker FX Cheatsheet

`fx_destructor :: DESTUCTOR_TYPEs`

SMOOTH_CLIP

A warm clipping function (based on polynomial expansion) that mimics the analog warmth of a tube amp.

Parameters

- *param_1* : **clipping threshold**
  The Clipping threshold is typically (0.01 to 1.0). Values closer to 0.0 are more aggressive. Start with 0.1.
- **param_2 : input drive**
  The input drive is a multiplier that determines how input signal is multiplied. Values between 2 and 6 will be warmer, softer clipping whereas values greater than 10 will heavier distortions.

SMOOTHER_CLIP

A warmer clipping function (based on polynomial expansion) that mimics the analog warmth of a tube amp.

Parameters

- *param_1* : **clipping threshold**
  The Clipping threshold is typically (0.01 to 1.0). Values closer to 0.0 are more aggressive. Start with 0.1.
- **param_2 : input drive**
  The input drive is a multiplier that determines how input signal is multiplied. Values between 2 and 6 will be warmer, softer clipping whereas values greater than 10 will heavier distortions.

SMOOTH_FUZZ

A warm clipping function (based on polynomial expansion) that mimics the analog warmth of a tube amp and also rectifies the signal creating an octave multiplying effect.

Parameters

- *param_1* : **clipping threshold**
  The Clipping threshold is typically (0.01 to 1.0). Values closer to 0.0 are more aggressive. Start with 0.1.
- **param_2 : input drive**
  The input drive is a multiplier that determines how input signal is multiplied. Values between 2 and 6 will be warmer, softer clipping whereas values greater than 10 will heavier distortions.

BIT_CRUSHER

A bit crusher simulates reduction in the number of bits used to store audio samples and creates quantization noise, a very "digital" sounding distortion.

Parameters

- *param_1* : **bit shift coefficient**
  Determines how extreme the effect is. 0.0 provides a gentle bit reduction while 1.0 is an aggressive bit reduction..
- **param_2 : (not used)**

SAMPLE_RATE_CRUSHER

A sample rate crusher reduces the sample rate without any anti-aliasing. The result is a remapping of upper harmonics into different audio bands.

Parameters

- *param_1* : **sample rate reduction coefficient**
  Determines how extreme the effect is. 0.0 provides a gentle bit reduction while 1.0 is an aggressive sample rate reduction.
- **param_2 : (not used)**

SHAPER Updated!

A shaper uses a sine function to "wrap" values outside of the clipping threshold. The result is louder notes end up with new higher-order harmonics.

Parameters

- *param_1* : *scale*
  Determines how extreme the effect is. 0.0 provides a gentle bit reduction while 1.0 (or higher in this case) is an aggressive clipping function.
- **param_2 : (not used)**

DIGITAL_PULVERIZER

Creates nasty, beautiful digital destruction of the incoming waveform.

Parameters

- *param_1* : **death**
  Play around with it – typically between 0.0 and 1.0
- **param_2 : and destruction**
  Play around with it – typically between 0.0 and 1.0

FLIP_FLOP_OCTAVE New!

Emulates a flip-flop circuit used in early synth and octave pedals.

Parameters

- *param_1* : **octaves to drop down**
  A value of 1.0 is same frequency, 2.0 is one octave down, 3.0 is two octaves down, etc. Doesn't do anything with fractional values – input rounded to whole number.
- **param_2 : (not used)**

DreamMaker FX Cheatsheet

OSC_TYPES

OSC_SINE
Sine wave

OSC_TRIANGLE
Triangle wave

OSC_TRIANGLE_CLIPPED
Clipped triangle wave (param1 is clipping threshold)

OSC_SQUARE
Square wave

OSC_SQUARE_SOFT
Square wave with smoothed transtions

OSC_PULSE
Pulse wave (param 1 is the duty cycle of pulse)

OSC_RAMP_POS
Ramp signal (positive slope)

OSC_RAMP_NEG
Ramp signal (negative slope)

OSC_RANDOM
Random smoothed waveform

Shape of the waveform for an LFO (low frequency oscillator). Google these for more info.

EFFECT_TRANSITION_SPEED

TRANS_VERY_FAST
TRANS_FAST
TRANS_MED
TRANS_SLOW
TRANS_VERY_SLOW

Determines how quickly the effect transitions between states or values.