DaisySP

Contents

PolyPluck

Description

Simplified Pseudo-Polyphonic Pluck Voice

Template Based Pluck Voice, with configurable number of voices and simple pseudo-polyphony.

DC Blocking included to prevent biases from causing unwanted saturation distortion.

Credits

Author: shensley

Date Added: March 2020

Init

Initializes the PolyPluck instance.

Arguments: - sample_rate: rate in Hz that the Process() function will be called.

```
void Init(float sample_rate)
```

Process

Process function, synthesizes and sums the output of all voices, triggering a new voice with frequency of MIDI note number when trig > 0.

Arguments: - float &trig: value by reference of trig. When trig > 0 a the next voice will be triggered, and trig will be set to 0. - float note: MIDI note number for the active_voice.

```
float Process(float &trig, float note)
```

increment active voice set new voice to new note

SetDecay

Sets the decay coefficients of the pluck voices. Expects 0.0-1.0 input.

```
void SetDecay(float p)
```

Member Variables

Example

No example Provided

AdEnv

Author: shensley

Trigger-able envelope with adjustable min/max, and independent per-segment time control.

TODO:

- Add Cycling
- Implement Curve (its only linear for now).
- Maybe make this an ADsr_ that has AD/AR/Asr_ modes.

Envelope Segments

Distinct stages that the phase of the envelope can be located in.

- IDLE = located at phase location 0, and not currently running
- ATTACK = First segment of envelope where phase moves from MIN value to MAX value
- DECAY = Second segment of envelope where phase moves from MAX to MIN value
- LAST = The final segment of the envelope (currently decay)

```
enum
{
    ADENV_SEG_IDLE,
    ADENV_SEG_ATTACK,
    ADENV_SEG_DECAY,
    ADENV_SEG_LAST,
};
```

Init

Initializes the ad envelope

float sample_rate - sample rate of the audio engine being run.

Defaults

- current segment = idle
- curve = linear

- phase = 0
- min = 0
- max = 1

```
void Init(float sample_rate);
```

Process

processes the current sample of the envelope. Returns the current envelope value. This should be called once per sample period.

```
float Process();
```

Trigger

Starts or retriggers the envelope.

```
inline void Trigger() { trigger_ = 1; }
```

Mutators

SetTime

Sets the length of time (in seconds) for a specific segment.

```
inline void SetTime(uint8_t seg, float time)
```

SetCurve

Sets the amount of curve applied. A positve value will create a log Input range: -100 to 100. (or more)

```
inline void SetCurve(float scalar) { curve_scalar_ = scalar; }
```

SetMin

Sets the minimum value of the envelope output Input range: -FLTmax_, to FLTmax_

```
inline void SetMin(float min) { min_ = min; }
```

SetMax

Sets the maximum value of the envelope output Input range: -FLTmax_, to FLTmax_

```
inline void SetMax(float max) { max_ = max; }
```

Accessors

GetValue

Returns the current output value without processing the next sample

GetCurrentSegment

Returns the segment of the envelope that the phase is currently located in.

```
inline uint8_t GetCurrentSegment() { return current_segment_; }
```

IsRunning

Returns true if the envelope is currently in any stage apart from idle.

```
inline bool IsRunning() const
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

// Shortening long macro for sample rate
#ifndef sample_rate

#endif

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
```

```
using namespace daisy;

static DaisySeed seed;
static AdEnv env;
static Oscillator osc;
static Metro tick;

static void AudioCallback(float *in, float *out, size_t size)
{
    float osc_out, env_out;
    for (size_t i = 0; i < size; i += 2)
    {
        // When the metro ticks, trigger the envelope to start.
        if (tick.Process())
        {
            env.Trigger();
        }

        // Use envelope to control the amplitude of the oscillator.
        env_out = env.Process();
        osc.SetAmp(env_out);
        osc_out = osc.Process();

        out[LEFT] = osc_out;
        out[RIGHT] = osc_out;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    env.Init(sample_rate);
    osc.Init(sample_rate);
```

```
    // Set up metro to pulse every second
    tick.Init(1.0f, sample_rate);

    // set adenv parameters
    env.SetTime(ADENV_SEG_ATTACK, 0.15);
    env.SetTime(ADENV_SEG_DECAY, 0.35);
    env.SetMin(0.0);
    env.SetMax(0.25);
    env.SetCurve(0); // linear

    // Set parameters for oscillator
    osc.SetWaveform(osc.WAVE_TRI);
    osc.SetFreq(220);
    osc.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Compressor

influenced by compressor in soundpipe (from faust).

Modifications made to do:

- Less calculations during each process loop (coefficients recalculated on parameter change).
- C++-ified
- added sidechain support

TODO:

- With fixed controls this is relatively quick, but changing controls now costs a lot more
- Still pretty expensive
- Add soft/hard knee settings
- Maybe make stereo possible? (needing two for stereo is a bit silly, and their gain shouldn't be totally unique.

by: shensley

Init

Initializes compressor

sample_rate - rate at which samples will be produced by the audio engine.

```
void Init(float sample_rate);
```

Process

compresses the audio input signal, either keyed by itself, or a secondary input.

in - audio input signal (to be compressed)

(optional) key - audio input that will be used to side-chain the compressor.

```
float Process(float in, float key);
float Process(float in);
```

setters

SetRatio

amount of gain reduction applied to compressed signals

Expects 1.0 -> 40. (untested with values < 1.0)

```
inline void SetRatio(const float &ratio)
```

SetThreshold

threshold in dB at which compression will be applied

Expects 0.0 -> -80.

```
inline void SetThreshold(const float &thresh)
```

SetAttack

envelope time for onset of compression for signals above the threshold.

Expects 0.001 -> 10

```
inline void SetAttack(const float &atk)
```

SetRelease

envelope time for release of compression as input signal falls below threshold.

Expects 0.001 -> 10

```
inline void SetRelease(const float &rel)
```

internals from faust struct

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

// Shortening long macro for sample rate
#ifndef sample_rate
```

```
#endif

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;

static Compressor comp;
// Helper Modules
static AdEnv env;
static Oscillator osc_a, osc_b;
static Metro tick;


static void AudioCallback(float *in, float *out, size_t size)
{
    float osc_a_out, osc_b_out, env_out, sig_out;
    for (size_t i = 0; i < size; i += 2)
    {
        // When the metro ticks:
        // trigger the envelope to start
        if (tick.Process())
        {
            env.Trigger();
        }

        // Use envelope to control the amplitude of the oscillator.
        env_out = env.Process();
        osc_a.SetAmp(env_out);
        osc_a_out = osc_a.Process();
        osc_b_out = osc_b.Process();
        // Compress the steady tone with the enveloped tone.
        sig_out = comp.Process(osc_b_out, osc_a_out);
```

```
        // Output
        out[LEFT] = sig_out; // compressed
        out[RIGHT] = osc_a_out; // key signal
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    comp.Init(sample_rate);
    env.Init(sample_rate);
    osc_a.Init(sample_rate);
    osc_b.Init(sample_rate);

    // Set up metro to pulse every second
    tick.Init(1.0f, sample_rate);

    // set compressor parameters
    comp.SetThreshold(-64.0f);
    comp.SetRatio(2.0f);
    comp.SetAttack(0.005f);
    comp.SetRelease(0.1250);

    // set adenv parameters
    env.SetTime(ADENV_SEG_ATTACK, 0.001);
    env.SetTime(ADENV_SEG_DECAY, 0.50);
    env.SetMin(0.0);
    env.SetMax(0.25);
    env.SetCurve(0); // linear

    // Set parameters for oscillator
    osc_a.SetWaveform(Oscillator::WAVE_TRI);
    osc_a.SetFreq(110);
```

```
    osc_a.SetAmp(0.25);
    osc_b.SetWaveform(Oscillator::WAVE_TRI);
    osc_b.SetFreq(220);
    osc_b.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

CrossFade

Performs a CrossFade between two signals

Original author: Paul Batchelor

Ported from Soundpipe by Andrew Ikenberry added curve option for constant power, etc.

TODO:

- implement exponential curve Process
- implement logarithmic curve Process

Curve Options

Curve applied to the CrossFade

- LIN = linear
- CPOW = constant power
- LOG = logarithmic
- EXP exponential
- LAST = end of enum (used for array indexing)

```
enum
{
    CROSSFADE_LIN,
    CROSSFADE_CPOW,
    CROSSFADE_LOG,
    CROSSFADE_EXP,
    CROSSFADE_LAST,
};
```

Init

Initializes CrossFade module

Defaults

- current position = .5
- curve = linear

```
inline void Init(int curve)
```

```
inline void Init()
```

Process

processes CrossFade and returns single sample

```
float Process(float &in1, float &in2);
```

Setters

SetPos

Sets position of CrossFade between two input signals

Input range: 0 to 1

```
inline void SetPos(float pos) { pos_ = pos; }
```

SetCurve

Sets current curve applied to CrossFade

Expected input: See Curve Options

```
inline void SetCurve(uint8_t curve) { curve_ = curve; }
```

Getters

GetPos

Returns current position

```
inline float GetPos(float pos) { return pos_; }
```

GetCurve

Returns current curve

```
inline uint8_t GetCurve(uint8_t curve) { return curve_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static CrossFade cfade;
static Oscillator osc_sine, osc_saw, lfo;

static void AudioCallback(float *in, float *out, size_t size)
{
    float saw, sine, pos, output;
    for (size_t i = 0; i < size; i += 2)
    {
        sine = osc_sine.Process();
        saw = osc_saw.Process();

        // lfo output = -1 to 1
        pos = lfo.Process();

        // scale signal between 0 and 1
        pos = (pos + 1) / 2;

        cfade.SetPos(pos);
        output = cfade.Process(sine, saw);

        // left out
        out[i] = output;

        // right out
        out[i+1] = output;
    }
}
```

```
int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // set params for CrossFade object
    cfade.Init();
    cfade.SetCurve(CROSSFADE_LIN);

    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);

    // set parameters for sawtooth oscillator object
    osc_saw.Init(sample_rate);
    osc_saw.SetWaveform(Oscillator::WAVE_POLYBLEP_SAW);
    osc_saw.SetFreq(100);
    osc_saw.SetAmp(0.25);

    // set parameters for triangle lfo oscillator object
    lfo.Init(sample_rate);
    lfo.SetWaveform(Oscillator::WAVE_TRI);
    lfo.SetFreq(.25);
    lfo.SetAmp(1);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
```

```
}
```

DcBlock

Removes DC component of a signal

Init

Initializes DcBlock module

```
void Init(float sample_rate);
```

Process

performs DcBlock Process

```
float Process(float in);
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static DcBlock block;
static Oscillator osc_sine;

static void AudioCallback(float *in, float *out, size_t size)
{
    float output;
    for (size_t i = 0; i < size; i += 2)
    {
        output = osc_sine.Process();

        // add dc to signal
        output += 1;
```

```
        // remove dc from signal
        output = block.Process(output);

        // left out
        out[i] = output;

        // right out
        out[i+1] = output;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    block.Init(sample_rate);

    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Decimator

Performs downsampling and bitcrush effects

Init

Initializes downsample module

```
void Init();
```

Process

Applies downsample and bitcrush effects to input signal. Returns one sample. This should be called once per sample period.

```
float Process(float input);
```

Mutators

SetDownsampleFactor

Sets amount of downsample Input range:

```
inline void SetDownsampleFactor(float downsample_factor)
```

SetBitcrushFactor

Sets amount of bitcrushing Input range:

```
inline void SetBitcrushFactor(float bitcrush_factor)
```

SetBitsToCrush

Sets the exact number of bits to crush

0-16 bits

```
inline void SetBitsToCrush(const uint8_t &bits)
```

Accessors

GetDownsampleFactor

Returns current setting of downsample

```
inline float GetDownsampleFactor() { return downsample_factor_; }
```

GetBitcrushFactor

Returns current setting of bitcrush

```
inline float GetBitcrushFactor() { return bitcrush_factor_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

// Shortening long macro for sample rate
#ifndef sample_rate

#endif

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Oscillator osc;
static Decimator decim;
static Phasor phs;

static void AudioCallback(float *in, float *out, size_t size)
{
    float osc_out, decimated_out;
```

```
    float downsample_amt;
    for (size_t i = 0; i < size; i += 2)
    {
        // Generate a pure sine wave
        osc_out = osc.Process();
        // Modulate downsample amount via Phasor
        downsample_amt = phs.Process();
        decim.SetDownsampleFactor(downsample_amt);
        // downsample and bitcrush
        decimated_out = decim.Process(osc_out);
        // outputs
        out[LEFT] = decimated_out;
        out[RIGHT] = decimated_out;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    osc.Init(sample_rate);
    phs.Init(sample_rate, 0.5f);
    decim.Init();

    // Set parameters for oscillator
    osc.SetWaveform(osc.WAVE_SIN);
    osc.SetFreq(220);
    osc.SetAmp(0.25);
    // Set downsampling, and bit crushing values.
    decim.SetDownsampleFactor(0.4f);
    decim.SetBitsToCrush(8);
```

```
    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

DelayLine

Simple Delay line.

November 2019

Converted to Template December 2019

declaration example: (1 second of floats)

```
DelayLine<float, SAMPLE_RATE> del;
```

By: shensley

Init

initializes the delay line by clearing the values within, and setting delay to 1 sample.

```
void Init()
```

Reset

clears buffer, sets write ptr to 0, and delay to 1 sample.

```
void Reset() {
```

SetDelay

sets the delay time in samples

If a float is passed in, a fractional component will be calculated for interpolating the delay line.

```
inline void SetDelay(size_t delay)
```

```
inline void SetDelay(float delay)
```

Write

writes the sample of type T to the delay line, and advances the write ptr

```
inline void Write(const T sample)
```

Read

returns the next sample of type T in the delay line, interpolated if necessary.

```
inline const T Read() const
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

// Set max delay time to 0.75 of samplerate.
#define MAX_DELAY static_cast<size_t>(48000 * 0.75f)

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;

// Helper Modules
static AdEnv env;
static Oscillator osc;
static Metro tick;

// Declare a DelayLine of MAX_DELAY number of floats.
static DelayLine<float, MAX_DELAY> del;

static void AudioCallback(float *in, float *out, size_t size)
{
    float osc_out, env_out, feedback, del_out, sig_out;
    for (size_t i = 0; i < size; i += 2)
    {
        // When the Metro ticks:
        // trigger the envelope to start, and change freq of oscillator.
```

```
        if (tick.Process())
        {
            float freq = rand() % 200;
            osc.SetFreq(freq + 100.0f);
            env.Trigger();
        }

        // Use envelope to control the amplitude of the oscillator.
        env_out = env.Process();
        osc.SetAmp(env_out);
        osc_out = osc.Process();

        // Read from delay line
        del_out = del.Read();
        // Calculate output and feedback
        sig_out = del_out + osc_out;
        feedback = (del_out * 0.75f) + osc_out;

        // Write to the delay
        del.Write(feedback);

        // Output
        out[LEFT] = sig_out;
        out[RIGHT] = sig_out;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    env.Init(sample_rate);
    osc.Init(sample_rate);
    del.Init();
```

```
    // Set up Metro to pulse every second
    tick.Init(1.0f, sample_rate);

    // set adenv parameters
    env.SetTime(ADENV_SEG_ATTACK, 0.001);
    env.SetTime(ADENV_SEG_DECAY, 0.50);
    env.SetMin(0.0);
    env.SetMax(0.25);
    env.SetCurve(0); // linear

    // Set parameters for oscillator
    osc.SetWaveform(osc.WAVE_TRI);
    osc.SetFreq(220);
    osc.SetAmp(0.25);

    // Set Delay time to 0.75 seconds
    del.SetDelay(sample_rate * 0.75f);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Core DSP

Helpful defines, functions, and other utilities for use in/with daisysp modules.

Generic Defines

PIs

```
#define PI_F 3.1415927410125732421875f
#define TWOPI_F (2.0f * PI_F)
#define HALFPI_F (PI_F * 0.5f)
```

Generic min/max/clamp

```
#define DSY_MIN(in, mn) (in < mn ? in : mn)
#define DSY_MAX(in, mx) (in > mx ? in : mx)
#define DSY_CLAMP(in, mn, mx) (DSY_MIN(DSY_MAX(in, mn), mx))
```

fast helpers

fmax/fmin

efficient floating point min/max

c/o stephen mccaul

```
inline float fmax(float a, float b)
```

```
inline float fmin(float a, float b)
```

fclamp

quick fp clamp

```
inline float fclamp(float in, float min, float max)
```

fastpower and fastroot

From Musicdsp.org "Fast power and root estimates for 32bit floats)

Original code by Stefan Stenzel

These are approximations

```
inline float fastpower(float f, int n)
```

```
inline float fastroot(float f, int n)
```

mtof

Midi to frequency helper

```
inline float mtof(float m)
{
    return powf(2, (m - 69.0f) / 12.0f) * 440.0f;
}
```

Filters

fonepole

one pole lpf

out is passed by reference, and must be retained between calls to properly filter the signal

coeff can be calculated:

coeff = 1.0 / (time * sample_rate) ; where time is in seconds

```
inline void fonepole(float &out, float in, float coeff)
```

median

Simple 3-point median filter

c/o stephen mccaul

```
template <typename T>
T median(T a, T b, T c)
```

Quick Effects

Soft Saturate

Based on soft saturate from:

musicdsp.org

Bram de Jong (2002-01-17)

This still needs to be tested/fixed. Definitely does some weird stuff

```
described as:
x < a:
     f(x) = x
x > a:
     f(x) = a + (x-a)/(1+((x-a)/(1-a))^2)
x > 1:
     f(x) = (a + 1)/2
```

```
inline float soft_saturate(float in, float thresh)
```

Example

No example Provided

Limiter

Description

Simple Peak Limiter

Credits

This was extracted from pichenettes/stmlib.

Credit to pichenettes/Mutable Instruments

Functions

Init

Initializes the Limiter instance.

```
void Init();
```

Process

Processes a block of audio through the limiter.

*in - pointer to a block of audio samples to be processed. The buffer is operated on directly.

size - size of the buffer “in”

pre_gain - amount of pre_gain applied to the signal.

```
void ProcessBlock(float *in, size_t size, float pre_gain);
```

Example

No example Provided

Line

creates a Line segment signal

Init

Initializes Line module.

```
void Init(float sample_rate);
```

Process

Processes Line segment. Returns one sample.

value of finished will be updated to a 1, upon completion of the Line's trajectory.

```
float Process(uint8_t *finished);
```

Start

Begin creation of Line.

Arguments:

- start - beginning value
- end - ending value
- dur - duration in seconds of Line segment

```
void Start(float start, float end, float dur);
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Line line_seg;
```

```
static Oscillator osc_sine;

uint8_t finished;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sine, freq;
    for (size_t i = 0; i < size; i += 2)
    {
        if (finished)
        {
            // Start creating a Line segment from 100 to 500 in 1 seconds
            line_seg.Start(100, 500, 1);
        }

        freq = line_seg.Process(&finished);
        osc_sine.SetFreq(freq);
        sine = osc_sine.Process();

        // left out
        out[i] = sine;

        // right out
        out[i+1] = sine;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // initialize Line module
    line_seg.Init(sample_rate);
    finished = 1;
```

```
    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);

    // Start callback
    seed.StartAudio(AudioCallback);

    while(1) {}
}
```

Metro

Creates a clock signal at a specific frequency.

Init

Initializes Metro module.

Arguments: - freq: frequency at which new clock signals will be generated Input Range: - sample_rate: sample rate of audio engine Input range:

```
void Init(float freq, float sample_rate);
```

Process

checks current state of Metro object and updates state if necesary.

```
uint8_t Process();
```

Reset

resets phase to 0

```
inline void Reset() { phs_ = 0.0f; }
```

Setters

SetFreq

Sets frequency at which Metro module will run at.

```
void SetFreq(float freq);
```

Getters

GetFreq

Returns current value for frequency.

```
inline float GetFreq() { return freq_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Metro clock;
static Oscillator osc_sine;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sine, freq;
    uint8_t tic;
    for (size_t i = 0; i < size; i += 2)
    {
        tic = clock.Process();
        if (tic)
        {
            freq = rand() % 500;
            osc_sine.SetFreq(freq);
        }

        sine = osc_sine.Process();

        // left out
        out[i] = sine;

        // right out
        out[i+1] = sine;
    }
}

int main(void)
{
```

```
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // initialize Metro object at 2 hz
    clock.Init(2, sample_rate);

    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Mode

Description

Resonant Modal Filter

Credits

Extracted from soundpipe to work as a Daisy Module, originally extracted from csound by Paul Batchelor.

Original Author(s): Francois Blanc, Steven Yi

Year: 2001

Location: Opcodes/biquad.c (csound)

Functions

Init

Initializes the instance of the module.

sample_rate: frequency of the audio engine in Hz

```
void Init(float sample_rate);
```

Process

Processes one input sample through the filter, and returns the output.

```
float Process(float in);
```

Clear

Clears the filter, returning the output to 0.0

```
void Clear();
```

Parameters

SetFreq

Sets the resonant frequency of the modal filter.

Range: Any frequency such that sample_rate / freq < PI (about 15.2kHz at 48kHz)

```
inline void SetFreq(float freq) { freq_ = freq; }
```

SetQ

Sets the quality factor of the filter.

Range: Positive Numbers (Good values range from 70 to 1400)

```
inline void SetQ(float q) { q_ = q; }
```

Example

No example Provided

NlFilt

port by: Stephen Hensley, December 2019

Non-linear filter.

The four 5-coefficients: a, b, d, C, and L are used to configure different filter types.

Structure for Dobson/Fitch nonlinear filter

Revised Formula from Risto Holopainen 12 Mar 2004

`Y{n} =tanh(a Y{n-1} + b Y{n-2} + d Y^2{n-L} + X{n} - C)`

Though traditional filter types can be made, the effect will always respond differently to different input.

This Source is a heavily modified version of the original source from Csound.

TODO:

- make this work on a single sample instead of just on blocks at a time.

Init

Initializes the NlFilt object.

```
void Init();
```

ProcessBlock

Process the array pointed to by *in and updates the output to *out;

This works on a block of audio at once, the size of which is set with the size.

```
void ProcessBlock(float *in, float *out, size_t size);
```

setters

SetCoefficients

inputs these are the five coefficients for the filter.

```
inline void SetCoefficients(float a, float b, float d, float C, float L)
```

individual setters for each coefficients.

```
inline void SetA(float a) { a_ = a; }

inline void SetB(float b) { b_ = b; }

inline void SetD(float d) { d_ = d; }

inline void SetC(float C) { C_ = C; }

inline void SetL(float L) { L_ = L; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

// Shortening long macro for sample rate
#ifndef sample_rate

#endif

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;

// Helper Modules
static AdEnv env;
static Oscillator osc;
static Metro tick;

static NlFilt filt;

static void AudioCallback(float *in, float *out, size_t size)
```

```
{
    // The NlFilt object currently only works on blocks of audio at a time.
    // This can be accomodated easily with an extra loop at the end.
    // We use size/2 since we only need to process mono
    float dry[size/2];
    float wet[size/2];
    float env_out;
    // loop through mono process
    for (size_t i = 0; i < size/2; i++)
    {
        // When the Metro ticks:
        // trigger the envelope to start, and change freq of oscillator.
        if (tick.Process())
        {
            float freq = rand() % 150;
            osc.SetFreq(freq + 25.0f);
            env.Trigger();
        }
        // Use envelope to control the amplitude of the oscillator.
        env_out = env.Process();
        osc.SetAmp(env_out);
        dry[i] = osc.Process();
    }
    // nonlinear filter
    filt.ProcessBlock(dry, wet, size/2);
    // Now write wet signal to both outputs.
    for (size_t i = 0; i < size; i+=2)
    {
        out[LEFT] = wet[i/2];
        out[RIGHT] = wet[i/2];
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
```

```
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    env.Init(sample_rate);
    osc.Init(sample_rate);

    // Set up Metro to pulse every 3 seconds
    tick.Init(0.333f, sample_rate);

    // Set adenv parameters
    env.SetTime(ADENV_SEG_ATTACK, 1.50);
    env.SetTime(ADENV_SEG_DECAY, 1.50);
    env.SetMin(0.0);
    env.SetMax(0.25);
    env.SetCurve(0); // linear

    // Set parameters for oscillator
    osc.SetWaveform(osc.WAVE_POLYBLEP_SAW);

    // Set coefficients for non-linear filter.
    filt.SetCoefficients(0.7f, -0.2f, 0.95f, 0.24f, 1000.0f);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Oscillator

Synthesis of several waveforms, including polyBLEP bandlimited waveforms.

Waveforms

Choices for output waveforms, POLYBLEP are appropriately labeled. Others are naive forms.

```
enum
{
    WAVE_SIN,
    WAVE_TRI,
    WAVE_SAW,
    WAVE_RAMP,
    WAVE_SQUARE,
    WAVE_POLYBLEP_TRI,
    WAVE_POLYBLEP_SAW,
    WAVE_POLYBLEP_SQUARE,
    WAVE_LAST,
};
```

Init

Initializes the Oscillator

float sample_rate - sample rate of the audio engine being run, and the frequency that the Process function will be called.

Defaults: - freq_ = 100 Hz - amp_ = 0.5 - waveform_ = sine wave.

```
void Init(float sample_rate)
```

SetFreq

Changes the frequency of the Oscillator, and recalculates phase increment.

```
inline void SetFreq(const float f)
```

SetAmp

Sets the amplitude of the waveform.

```
inline void SetAmp(const float a) { amp_ = a; }
```

SetWaveform

Sets the waveform to be synthesized by the Process() function.

```
inline void SetWaveform(const uint8_t wf)
{
    waveform_ = wf < WAVE_LAST ? wf : WAVE_SIN;
}
```

Process

Processes the waveform to be generated, returning one sample. This should be called once per sample period.

```
float Process();
```

PhaseAdd

Adds a value 0.0-1.0 (mapped to 0.0-TWO_PI) to the current phase. Useful for PM and “FM” synthesis.

```
void PhaseAdd(float _phase) { phase_ += (_phase * float(M_TWOPI)); }
```

Reset

Resets the phase to the input argument. If no argument is present, it will reset phase to 0.0;

```
void Reset(float _phase = 0.0f) { phase_ = _phase; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"
```

```
using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Oscillator osc;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sig;
    for (size_t i = 0; i < size; i += 2)
    {
        sig = osc.Process();

        // left out
        out[i] = sig;

        // right out
        out[i+1] = sig;
    }
}

int main(void)
{
    // initialize seed hardware and oscillator daisysp module
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    osc.Init(sample_rate);

    // Set parameters for oscillator
    osc.SetWaveform(osc.WAVE_SIN);
    osc.SetFreq(440);
    osc.SetAmp(0.5);
```

```
    // start callback
    seed.StartAudio(AudioCallback);

    while(1) {}
}
```

Phasor

Generates a normalized signal moving from 0-1 at the specified frequency.

TODO:

I'd like to make the following things easily configurable:

- Selecting which channels should be initialized/included in the sequence conversion.
- Setup a similar start function for an external mux, but that seems outside the scope of this file.

Init

Initializes the Phasor module

sample rate, and freq are in Hz

initial phase is in radians

Additional Init functions have defaults when arg is not specified:

- phs = 0.0f
- freq = 1.0f

```
inline void Init(float sample_rate, float freq, float initial_phase)
```

```
inline void Init(float sample_rate, float freq)
```

```
inline void Init(float sample_rate)
```

Process

processes Phasor and returns current value

```
float Process();
```

Setters

SetFreq

Sets frequency of the Phasor in Hz

```
void SetFreq(float freq);
```

Getters

GetFreq

Returns current frequency value in Hz

```
        inline float GetFreq() { return freq_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Phasor ramp;
static Oscillator osc_sine;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sine, freq;
    for (size_t i = 0; i < size; i += 2)
    {
        // generate Phasor value (0-1), and scale it between 0 and 300
        freq = ramp.Process()*300;

        osc_sine.SetFreq(freq);
        sine = osc_sine.Process();

        // left out
        out[i] = sine;

        // right out
        out[i+1] = sine;
    }
}
```

```
int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // initialize Phasor module
    ramp.Init(sample_rate, 1, 0);

    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

pitchshift

From ucsd.edu "Pitch Shifting" t = 1 - ((s *f) / R) where: s is the size of the delay f is the frequency of the lfo r is the sample_rate solving for t = 12.0 f = (12 - 1)* 48000 / SHIFT_BUFFER_SIZE; Shift can be 30-100 ms lets just start with 50 for now. 0.050 * SR = 2400 samples (at 48kHz) First Process delay mod/crossfade Handle Delay Writing Modulate Delay Lines mod_a_amt = mod_b_amt = 0.0f; d_[0].SetDelay(mod_[0] + mod_a_amt_); d_[1].SetDelay(mod_[1] + mod_b_amt_); lfo stuff pitch stuff

Example

```
// Example that takes the mono input from channel 1 (left input),
// and pitchshifts it up 1 octave.
// The left output will be pitchshifteed, while the right output
// stays will be the unshifted left input.

#include "daisysp.h"
#include "daisy_seed.h"

// Defines for Interleaved Audio
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
using namespace daisy;

DaisySeed seed;
PitchShifter ps;

static void AudioCallback(float *in, float *out, size_t size)
{
    float shifted, unshifted;
    for (size_t i = 0; i < size; i += 2)
    {
        unshifted = in[LEFT];
```

```
        shifted = ps.Process(unshifted);
        out[LEFT] = shifted;
        out[RIGHT] = unshifted;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    ps.Init(sample_rate);
    // set transposition 1 octave up (12 semitones)
    ps.SetTransposition(12.0f);

    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Pluck

Produces a naturally decaying plucked string or drum sound based on the Karplus-Strong algorithms.

This code has been extracted from the Csound opcode “pluck” It has been modified to work as a Daisy Soundpipe module.

Original Author(s): Barry Vercoe, John ffitch

Year: 1991

Location: OOps/ugens4.c

Mode

The method of natural decay that the algorithm will use.

- RECURSIVE: 1st order recursive filter, with coefs .5.
- WEIGHTED_AVERAGE: weighted averaging.

```
enum
{
    PLUCK_MODE_RECURSIVE,
    PLUCK_MODE_WEIGHTED_AVERAGE,
    PLUCK_LAST,
};
```

Init

Initializes the Pluck module.

Arguments:

- sample_rate: Sample rate of the audio engine being run.
- buf: buffer used as an impulse when triggering the Pluck algorithm
- npt: number of elementes in buf.
- mode: Sets the mode of the algorithm.

```
void Init (float sample_rate, float *buf, int32_t npt, int32_t mode);
```

Process

Processes the waveform to be generated, returning one sample. This should be called once per sample period.

```
float Process (float &trig);
```

Mutators

SetAmp

Sets the amplitude of the output signal.

Input range: 0-1?

```
inline void SetAmp(float amp) { amp_ = amp; }
```

SetFreq

Sets the frequency of the output signal in Hz.

Input range: Any positive value

```
inline void SetFreq(float freq) { freq_ = freq; }
```

SetDecay

Sets the time it takes for a triggered note to end in seconds.

Input range: 0-1

```
inline void SetDecay(float decay) { decay_ = decay; }
```

SetDamp

Sets the dampening factor applied by the filter (based on PLUCK_MODE)

Input range: 0-1

```
inline void SetDamp(float damp) { damp_ = damp; }
```

SetMode

Sets the mode of the algorithm.

```
inline void SetMode(int32_t mode) { mode_ = mode; }
```

Accessors

GetAmp

Returns the current value for amp.

```
inline float GetAmp() { return amp_; }
```

GetFreq

Returns the current value for freq.

```
inline float GetFreq() { return freq_; }
```

GetDecay

Returns the current value for decay.

```
inline float GetDecay() { return decay_; }
```

GetDamp

Returns the current value for damp.

```
inline float GetDamp() { return damp_; }
```

GetMode

Returns the current value for mode.

```
inline int32_t GetMode() { return mode_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"
#include <algorithm>

// Shortening long macro for sample rate
```

```
#ifndef sample_rate

#endif

// Interleaved audio definitions
#define LEFT (i)
#define RIGHT (i+1)

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;

// Helper Modules
static Metro tick;
static Pluck plk;

// MIDI note numbers for a major triad
const float kArpeggio[3] = { 48.0f, 52.0f, 55.0f };
uint8_t arp_idx;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sig_out, freq, trig;
    for (size_t i = 0; i < size; i += 2)
    {
        // When the Metro ticks:
        // advance the kArpeggio, and trigger the Pluck.
        trig = 0.0f;
        if (tick.Process())
        {
            freq = mtof(kArpeggio[arp_idx]); // convert midi nn to frequency.
            arp_idx = (arp_idx + 1) % 3; // advance the kArpeggio, wrapping at th
            plk.SetFreq(freq);
            trig = 1.0f;
        }
        sig_out = plk.Process(trig);
        // Output
```

```
        out[LEFT] = sig_out;
        out[RIGHT] = sig_out;
    }
}

int main(void)
{
    float init_buff[256]; // buffer for Pluck impulse

    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // Set up Metro to pulse every second
    tick.Init(1.0f, sample_rate);
    // Set up Pluck algo
    plk.Init(sample_rate, init_buff, 256, PLUCK_MODE_RECURSIVE);
    plk.SetDecay(0.95f);
    plk.SetDamp(0.9f);
    plk.SetAmp(0.3f);

    arp_idx = 0;




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Port

Applies portamento to an input signal. At each new step value, the input is low-pass filtered to move towards that value at a rate determined by ihtim. ihtim is the half-time of the function (in seconds), during which the curve will traverse half the distance towards the new value, then half as much again, etc., theoretically never reaching its asymptote.

This code has been ported from Soundpipe to DaisySP by Paul Batchelor.

The Soundpipe module was extracted from the Csound opcode "portk".

Original Author(s): Robbin Whittle, John ffitch

Year: 1995, 1998

Location: Opcodes/biquad.c

Init

Initializes Port module

Arguments:

- sample_rate: sample rate of audio engine
- htime: half-time of the function, in seconds.

```
void Init(float sample_rate, float htime);
```

Process

Applies portamento to input signal and returns processed signal.

```
float Process(float in);
```

Setters

SetHtime

Sets htime

```
inline void SetHtime(float htime) { htime_ = htime; }
```

Getters

GetHtime

returns current value of htime

```
        inline float GetHtime() { return htime_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Port slew;
static Metro clock;
static Oscillator osc_sine;

float freq;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sine, slewed_freq;
    uint8_t tic;
    for (size_t i = 0; i < size; i += 2)
    {
        tic = clock.Process();
        if (tic)
        {
            freq = rand() % 500;
        }

        slewed_freq = slew.Process(freq);
        osc_sine.SetFreq(slewed_freq);
```

```
        sine = osc_sine.Process();

        // left out
        out[i] = sine;

        // right out
        out[i+1] = sine;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();

    // set params for Port object
    slew.Init(sample_rate, .09);

    clock.Init(1, sample_rate);

    // set parameters for sine oscillator object
    osc_sine.Init(sample_rate);
    osc_sine.SetWaveform(Oscillator::WAVE_SIN);
    osc_sine.SetFreq(100);
    osc_sine.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

ReverbSc

Description

Stereo Reverb

Credits

Reverb SC: Ported from csound/soundpipe

Original author(s): Sean Costello, Istvan Varga

Year: 1999, 2005

Ported to soundpipe by: Paul Batchelor

Ported by: Stephen Hensley

Init

Initializes the reverb module, and sets the sample_rate at which the Process function will be called.

Returns 0 if all good, or 1 if it runs out of delay times exceed maximum allowed.

```
int Init(float sample_rate);
```

Process

Process the input through the reverb, and updates values of out1, and out2 with the new processed signal.

```
int Process(const float &in1, const float &in2, float *out1, float *out2);
```

SetFeedback

controls the reverb time. reverb tail becomes infinite when set to 1.0

range: 0.0 to 1.0

```
inline void SetFeedback(const float &fb) { feedback_ = fb; }
```

SetLpFreq

controls the internal dampening filter's cutoff frequency.

range: 0.0 to sample_rate / 2

```
inline void SetLpFreq(const float &freq) { lpfreq_ = freq; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;

ReverbSc verb;

static void AudioCallback(float *in, float *out, size_t size)
{
    for (size_t i = 0; i < size; i += 2)
    {
        verb.Process(in[i], in[i+1], &out[i], &out[i+1]);
    }
}

int main(void)
{
    // initialize seed hardware and whitenoise daisysp module
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    verb.Init(sample_rate);
    verb.SetFeedback(0.85f);
    verb.SetLpFreq(18000.0f);
```

```
    // start callback
    seed.StartAudio(AudioCallback);

    while(1) {}
}
```

Svf

```
Double Sampled, Stable State Variable Filter
```

Credit to Andrew Simper from musicdsp.org

This is his "State Variable Filter (Double Sampled, Stable)"

Additional thanks to Laurent de Soras for stability limit, and Stefan Diedrichsen for the correct notch output

Ported by: Stephen Hensley

example: daisysp/examples/Svf/

Init

Initializes the filter

float sample_rate - sample rate of the audio engine being run, and the frequency that the Process function will be called.

```
void Init(float sample_rate);
```

Process

Process the input signal, updating all of the outputs.

```
void Process(float in);
```

Setters

SetFreq

sets the frequency of the cutoff frequency.

f must be between 0.0 and sample_rate / 2

```
void SetFreq(float f);
```

SetRes

sets the resonance of the filter.

Must be between 0.0 and 1.0 to ensure stability.

```
void SetRes(float r);
```

SetDrive

sets the drive of the filter, affecting the response of the resonance of the filter..

```
inline void SetDrive(float d) { drive_ = d; }
```

Filter Outputs

Lowpass Filter

```
inline float Low() { return out_low_; }
```

Highpass Filter

```
inline float High() { return out_high_; }
```

Bandpass Filter

```
inline float Band() { return out_band_; }
```

Notch Filter

```
inline float Notch() { return out_notch_; }
```

Peak Filter

```
inline float Peak() { return out_peak_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisy;
using namespace daisysp;
using namespace daisy;
```

```
static DaisySeed seed;

Oscillator osc;
Svf filt;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sig;

    for (size_t i = 0; i < size; i += 2)
    {
        sig = osc.Process();

        filt.Process(sig);

        // left out
        out[i] = filt.Low();

        // right out
        out[i + 1] = filt.High();
    }
}

int main(void)
{
    // initialize seed hardware and Svf daisysp module
    float sample_rate;
    seed.Configure();
    seed.Init();
    sample_rate = seed.AudioSampleRate();
    // Initialize Oscillator, and set parameters.
    osc.Init(sample_rate);
    osc.SetWaveform(osc.WAVE_POLYBLEP_SAW);
    osc.SetFreq(250.0);
    osc.SetAmp(0.5);
    // Initialize Filter, and set parameters.
    filt.Init(sample_rate);
```

```
    filt.SetFreq(500.0);
    filt.SetRes(0.85);
    filt.SetDrive(0.8);




    dsy_adc_start();

    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

Tone

A first-order recursive low-pass filter with variable frequency response.

Init

Initializes the Tone module.

sample_rate - The sample rate of the audio engine being run.

```
void Init(float sample_rate);
```

Process

Processes one sample through the filter and returns one sample.

in - input signal

```
float Process(float &in);
```

Setters

SetFreq

Sets the cutoff frequency or half-way point of the filter.

Arguments

- freq - frequency value in Hz. Range: Any positive value.

```
inline void SetFreq(float &freq)
```

Getters

GetFreq

Returns the current value for the cutoff frequency or half-way point of the filter.

```
inline float GetFreq() { return freq_; }
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static Tone flt;
static Oscillator osc, lfo;

static void AudioCallback(float *in, float *out, size_t size)
{
    float saw, freq, output;
    for (size_t i = 0; i < size; i += 2)
    {
        freq = 2500 + ( lfo.Process()*2500 );
        saw = osc.Process();

        flt.SetFreq(freq);
        output = flt.Process(saw);

        // left out
        out[i] = output;

        // right out
        out[i+1] = output;
    }
}

int main(void)
{
    // initialize seed hardware and daisysp modules
    float sample_rate;
    seed.Configure();
    seed.Init();
```

```
    sample_rate = seed.AudioSampleRate();

    // initialize Tone object
    flt.Init(sample_rate);

    // set parameters for sine oscillator object
    lfo.Init(sample_rate);
    lfo.SetWaveform(Oscillator::WAVE_TRI);
    lfo.SetAmp(1);
    lfo.SetFreq(.4);

    // set parameters for sine oscillator object
    osc.Init(sample_rate);
    osc.SetWaveform(Oscillator::WAVE_POLYBLEP_SAW);
    osc.SetFreq(100);
    osc.SetAmp(0.25);




    // start callback
    seed.StartAudio(AudioCallback);


    while(1) {}
}
```

WhiteNoise

fast white noise generator I think this came from musicdsp.org at some point

Init

Initializes the WhiteNoise object

```
void Init()
```

SetAmp

sets the amplitude of the noise output

```
inline void SetAmp(float a) { amp_ = a; }
```

Process

returns a new sample of noise in the range of -amp_ to amp_

```
inline float Process()
```

Example

```
#include "daisysp.h"
#include "daisy_seed.h"

using namespace daisysp;
using namespace daisy;

static DaisySeed seed;
static WhiteNoise nse;

static void AudioCallback(float *in, float *out, size_t size)
{
    float sig;

    for (size_t i = 0; i < size; i += 2)
    {
```

```
        sig = nse.Process();

        // left out
        out[i] = sig;

        // right out
        out[i + 1] = sig;
    }
}

int main(void)
{
    // initialize seed hardware and WhiteNoise daisysp module
    seed.Configure();
    seed.Init();
    nse.Init();

    // start callback
    seed.StartAudio(AudioCallback);

    while(1) {}
}
```