S-89.3510 DSP Processors and Audio Signal Processing

Virtual analog synthesis

Freescale/Chameleon

Group 2

Konsta Hölttä, 79149S
Nuutti Hölttä, 217437

Contents

1 Introduction

In this project, we implement a subtractive sound synthesizer based on the analog devices from 70's and 80's. Our synth works on instruments, that contain separate oscillators, filters, effects, ADSRs and LFOs. Oscillators generate sound samples, which are manipulated by the other blocks, finally producing audible output. The notes are to be commanded via a MIDI connection.

The synth core is very modular and new instruments are easily created. In addition to the basic analog synth mimicing way, where the synth is monophonic and has a specific number of instruments where each contains a single oscillator source, we support instrument *c*hannels that work like polyphonic instruments: when a new note is input, a new channel is reserved without killing the possibly playing old note on the same instrument. (This mimics several identical analog instruments working in parallel.)

The system reads MIDI events, and creates playing notes on corresponding instruments at the note frequency. Each instrument sets up new playing channels, unless the classic style is separately enabled. The notes end when the key is released and their adsr releases.

2 Realization

2.1 DSP

The core contains instrument-agnostic basic building blocks such as oscillators (sine wave, triangle wave, noise, etc.) and lowpass filters. They take in some parameters and produce audio samples with no side effects, besides modifying the supplied state structure. These can then be combined as instrument functions that operate on instrument parameters and channel data. The program structure is optimized mainly for clarity, not performance.

The data flow works on single instruments so that many channels can operate on the same instrument code, but with different state. There is one separate set of functions and parameter set (e.g. lowpass cutoff and resonance frequencies) per each instrument.

The runtime structure is composed of audio channels that are rendered per each sample. The code (flow of data from a function to another) is hard-coded on the DSP, but the parameter data can be modified on-the-fly for easy tuning. The channels' state data is initialized when a new note is found and a channel is allocated, and it contains time-dependent data for that specific note, e.g. note frequency, filter history values and LFO phase (time since note start).

The audio data flow is visualized in the accompanying example pictures: 1 for combination of all playing notes, 2 for a single instrument's structure working on the data of one channel, 3 for one instrument's oscillators, and 4 for one

instrument's filter chain.

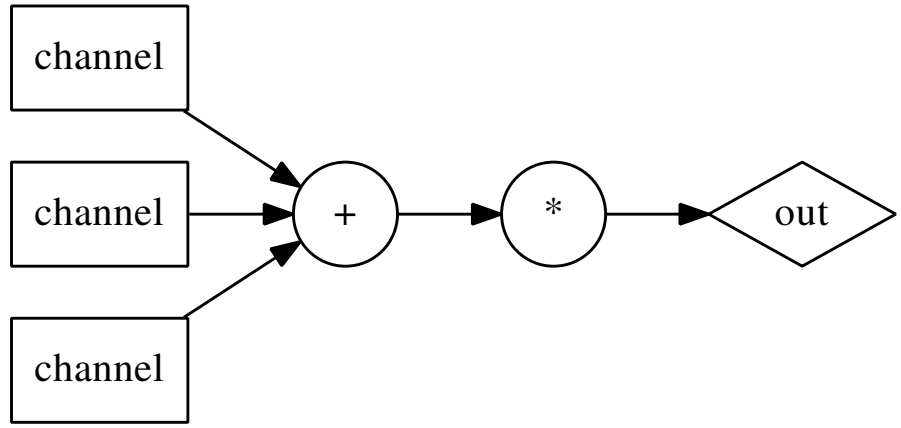

Figure 1: "main loop" contents. Diamonds: numbers, blocks: functions, circles: operators.

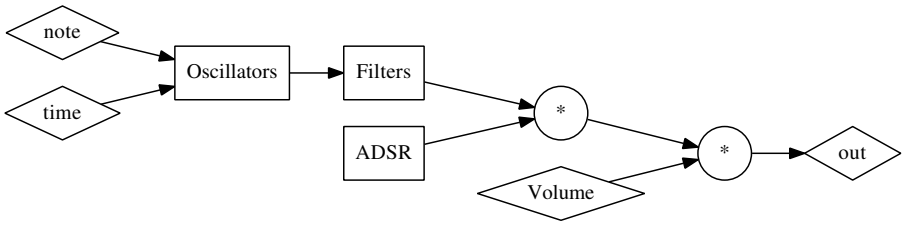

Figure 2: The basic flow for one instrument

In the following pseudocode, the main "framework" code is presented:

```
sample eval_channel(Channel* ch) {
    const Instrument* instr = ch->instr;
    sample a = instr->oscfunc(instr, ch->instrstate, ch->note);
    sample b = instr->filtfunc(instr, ch->instrstate, a);
    sample c = adsr(ch->adsrstate, instr->adsrcoefs, b);
    if (ch->adsrstate->finished)
        ch->alive = false;
    return ch->volume * c;
}

sample render() {
    sample out = 0;
    for (int i = 0; i < NUM_CHANNELS; i++) {
        if (channels[i].alive)
            out += eval_channel(&channels[i]);
    }
    return out * mastervol;
```

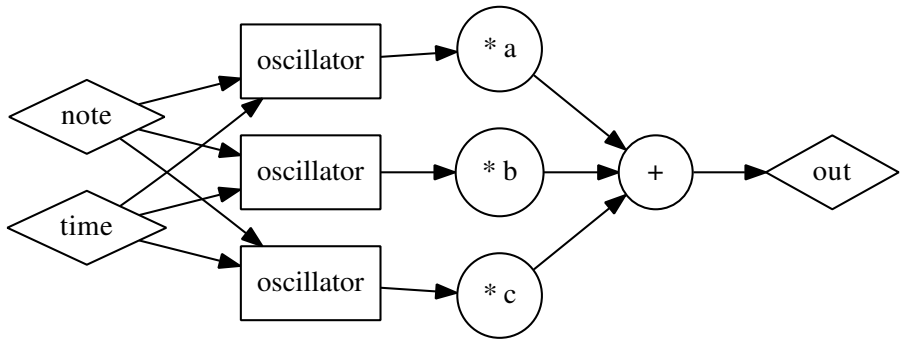

Figure 3: Imaginary example: several oscillators for one instrument

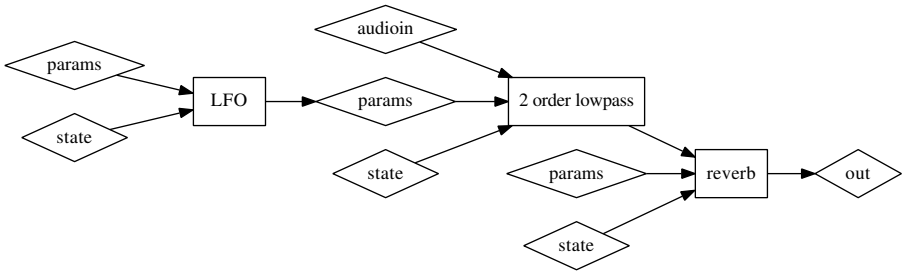

Figure 4: Imaginary example: some filter objects for one instrument

```
}

bool note_on(int midinote, int instrument) {
    for (int i = 0; i < NUM_CHANNELS; i++) {
        Channel* ch = channels[i];
        if (!ch->alive) {
            ch->instr = instruobj_by_num(instrument);
            memset(ch->instrstate, 0, sizeof(ch->instrstate));
            memset(ch->adsrstate, 0, sizeof(ch->adsrstate));
            ch->note = midinote;
            ch->alive = true;
            return true;
        }
    }
    return false;
}
bool note_off(int midinote, int instrument) {
    for (int i = 0; i < NUM_CHANNELS; i++) {
        Channel* ch = channels[i];
        if (ch->alive
                && ch->instr->midinum == instrument
                && ch->note == midinote) {
            ch->adsrstate->off = true;
```

```
            return;
        }
    }
}
```

The instrument-specific `oscfunc` might contain just a single call to a specific waveform with frequency determined by the channel's note, or then several calls to generate time-varying beating with an LFO. The `filtfunc` is separated for clarity and because of the thought that many instruments might end up using a similar `oscfunc`; it might look like the figure 4.

New notes are allocated when they arrive from the MIDI handler. We assume that the musician never plays too many notes so that the channel array never actually fills up.

2.2 ColdFire

The helper microcontroller is used to process the MIDI input and to translate it to the DSP. In the pseudocode above, the functions `note_on` and `note_off` will be initiated from an interrupt or alternatively called in the main loop if the interrupt handler simply puts the note events to a buffer.

In addition to forwarding the plain MIDI input to the DSP, it's also planned to store some events in a buffer and replay them to the DSP, like a sequencer (aligning their time to the nearest beat). This should be trivial.

We won't go into details in this part, because the main point in this project is the DSP assembly code.

3 Time Schedule

Konsta is pretty much offline for the first three weeks except their last two weekends. Coding is very much possible, but collaboration and testing with real hardware is more difficult.

Week 1 (Mon 11.3.) Plan DL, initial structure with simple MIDI reading, prototyping in C/Pd/Matlab

Week 2 (18.3.-) Structure formed, some oscillators, channel dataflow working with a single trivial instrument, an oscillators plays when MIDI keyboard key is pressed

Week 3 (25.3.-) ADSR for instrument output modulation, one or two (good-sounding) instruments with proper filters

Week 4 (1.4.-) Pitstop, more oscillators, awesome instruments, LFOs

Week 5 (8.4.-) Note storing and playback, bug hunting, efficiency, filters/effects

Week 6 (15.4.-) Preliminary demo - no bugs, just polishing

Week 7 (demo Thu 25.4.) Preparing for the demo

Week 8 (29.4.-) Report writing

Week 9 (7.5.) Report DL