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, a subtractive sound synthesizer based on the analog devices from 70's and 80's [1] was implemented on a Chameleon DSP hardware. Our synth works on instruments, that contain separate oscillators, filters, ADSRs and LFOs. Oscillators generate sound samples, which are manipulated by the filters, finally producing audible output. The oscillator and filter behaviour can be tuned with ADSRs or LFOs in realtime. The notes are read from a MIDI connection or a test button on the panel.

The synth is written in a very modular way; new instruments are easily created. The system differs from analog synths mostly in the way that it is not monophonic, but instead we support instrument *channels* ("voices" in some sources – our terminology is not standard) that work like separate polyphonic instruments: when a new note is started, 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 notes end when their adsr finally releases, which usually happens after releasing the corresponding key on a keyboard.

The panel interface is not very convenient. Sorry about that. For maximal user experience, using a MIDI keyboard is recommended.

2 Realization

The program is divided into two high-level parts, the actual synth (written in DSP56k assembly) and a user interface with MIDI event and panel handling part (written in C). The synth runs on the DSP and the interface on the ColdFire. The structure follows largely the figures in our original plan.

The sampling frequency is 48000 Hz.

Much of the math is described in the article by Huovilainen and Välimäki [2].

2.1 Synth

The synth code consists of oscillators and filters which are combined into instruments (with ADSR envelopes), and a main routine that evaluates the instruments and generates each sample.

The DSP uses 24-bit fixed-point math, i.e. usual calculations happen with values between $[-1, 1)$, with uniform spacing (in contrast to floating-point). If, for example, a value needs to be multiplied by a value bigger than 1, the multiplier must be scaled down, and the final result must then be e.g. bit-shifted by the scaling factor. The number 1 also cannot be represented exactly as-is; the largest 24-bit fixed point value is $1 - 2^{-23}$. The accumulator registers are 56-bit, though, so they can hold larger intermediate values. From here on in this report, in the context of fixed-point numbers, 1.0 shall be understood as $1 - 2^{-23}$, which is the value that is the closest to 1.0 in 24-bit fixed point representation.

A struct-like convention is employed in several places in the DSP code. For instance, a channel can be seen as a struct whose members are the currently playing note, the instrument type, the oscillator's state and so on. Each of these members is at a fixed offset with respect to the beginning of the memory block reserved for the channel, with these offsets simply defined with the equ assembler directive. So, in practice, a struct type definition simply consists of the struct's size (in words) and the offsets of its members (also in words, counted from the beginning of the struct), which are all assembly-time constants. In addition to channels, several other things are defined as structs, such as oscillator and filter states.

2.2 Main routine

The main routine is what puts everything in the DSP part together. When the user presses a key on the MIDI keyboard, an interrupt is sent by the ColdFire to the DSP. The interrupt places its data into specific memory slots which are then read in the main loop. Whenever the code in the main loop detects that a key just went down, it proceeds to allocate a new "channel" for this new note - channels are data structures containing information about currently active notes (such as note number and instrument type). There is a fixed maximum number of channels, and if they're all in use when a new key-down event arrives, the new key is simply ignored. Otherwise, the channel's contents are initialized to appropriate values.

The output generation acts on a per-sample basis (in contrast to a block-based behavior). The samples are generated as follows. The main routine loops through the channels, and for each active channel, the corresponding instrument's oscillator and filter subroutines are called. The oscillator is evaluated first, and its output goes to the filter. The output of the filter is then modulated by an ADSR envelope. The structure of one channel is shown in figure 1.

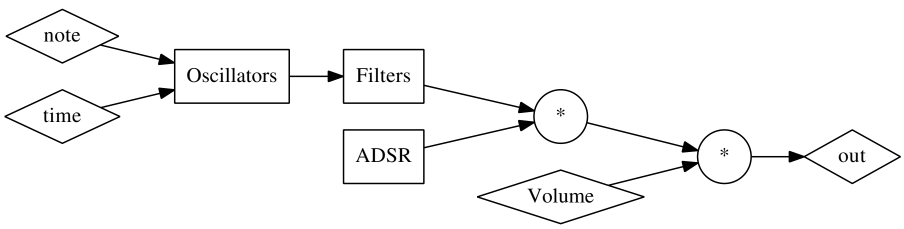

Figure 1: A basic channel data flow

The results of each channel are summed together to form the final output. The output is sent to the DAC peripheral, which syncs the sample rate.

When the user releases a key, an interrupt is sent in a manner similar to when a key went down. The main routine processes this event by finding the corresponding channel and marks it as released. The channel is not killed at this point; instead, the ADSR state is set to release. The channel is killed when the release stage ends, i.e. the ADSR value goes to zero.

The channel numbers come from the coldfire code, and they are indices to the AllInstruments table. We implemented the following instruments:

1. Bass: simple dpw saw, low-pass filtered,
2. BassSinLfo: same as previous, but a LFO sine wave controls the filter's cutoff,
3. BassAdsrLfo: same as previous, but instead of a sine wave, a separate ADSR controls the cutoff,
4. PulseBass: just a dpw pulse wave without a filter, and the pulse duty cycle is controlled by an ASDR
5. Noise: white noise, high-pass filtered, imitates a hi-hat drum
6. Bass4: like the first one, but with a 4-pole filter
7. Noise4: like the previous noise, but with a 4-pole filter

See more about these in the assembly code.

2.3 Oscillators

Oscillators consist of a set of parameter and state values. Parameters are per-instrument, whereas the state contains data specific for an oscillator. When a new note is started, the oscillator's state is initialized with values depending on the note value. When an oscillator is evaluated, it generates its output value using these values. Furthermore, it advances its state so that the next time the oscillator is evaluated, it produces the next value. The output of an oscillator is a function of only the parameters and the state; that is, there is no global time counter.

2.3.1 Sawtooth

The sawtooth oscillator is basically just a counter that is incremented every time it is evaluated. The amount by which it is incremented depends on the frequency of the note and thus the MIDI note value. These constants are compile-time precalculated per each MIDI note (of which there are just 128, so not much memory is used). The value of the sawtooth ranges from -1.0 to 1.0; a neat branchless bit-shifting trick is used to wrap the values from $1.0 + x$ back to $-1.0 + x$.

2.3.2 DPW sawtooth

Because of the aliased nature of the pure sawtooth it sounds rather unpleasant, and it must be corrected using the DPW (differentiated parabolic waveform) method [2]. DPW sawtooth basically outputs the derivative of a squared sawtooth. A DPW sawtooth oscillator is thus based on a pure sawtooth, but its state contains also the previous squared value of the sawtooth signal. Its evaluation consists of taking the difference of the square of the current value of the pure sawtooth and the previous squared value, scaled by a factor that depends on the frequency of the oscillator.

$$dpwSaw(n) = (saw(n)^2 - saw(n-1)^2) * c$$

where

$$c = \frac{f_s}{4f(1 - f/f_s)}$$

f is the saw's frequency and f_s is the sampling frequency. `saw(n)` is the pure saw function at the note frequency.

2.3.3 DPW pulse

The same problem as with sawtooths is presented in [2], which is corrected here similarly. While a pure pulse wave is implementable as the difference of two phase-shifted pure sawtooths, a DPW pulse wave is similarly the difference of two DPW sawtooths. The amount of phase-shifting depends on the desired duty cycle of the pulse wave. The duty cycle can be modified on the fly, e.g. with a LFO.

2.3.4 Noise

The noise oscillator is implemented with a simple white noise pseudo-random xorshift algorithm [3]. Unlike the other oscillators, the output of the noise oscillator doesn't vary according to the note, since white noise contains all frequencies. Instead it is convenient to combine this oscillator with e.g. a high-pass filter.

The xorshift algorithm corresponds to the following pseudo-code:

```
v := previous output value (or seed)
v := v ^ (v<<8)
v := v ^ (v>>1)
v := v ^ (v<<11)
output := v
```

where v is a 24-bit temporary, and logical shifts are used. The shift amounts were computed rather brute-forcily using methods presented in [3]. The period of the pseudo-random number sequence is $2^{24} - 1$.

2.3.5 Sine wave

The sine wave is implemented with a lookup table with linear interpolation between the samples. Since a sine wave makes a rather uninteresting oscillator for an instrument, it is not used as such; instead it is used for LFOs. Since LFOs only require fairly low frequencies, this also permits the usage of a rather small lookup table without audible deficiencies.

2.4 Filters

Similarly to oscillators, filters also have parameters and states. Filter evaluation routines differ from oscillators in the way that oscillators take no per-sample input, whereas filters do take input, namely the output of an oscillator.

2.4.1 Trivial lowpass

This filter is a one-pole lowpass whose state contains its last output and the smoothing factor. The implementation is rather straightforward (a simple RC filter, see [1]), however one must pay attention to fixed-point issues and scale values appropriately.

The output y changes according to the following pseudo-code (x is the input):

```
y := y + (x-y) * g
```

where

$$
\begin{aligned}
g &= \frac{Kf_c}{Kf_c + 1}, \\
K &= \frac{2\pi}{f_s}
\end{aligned}
$$

f_c is the cutoff frequency and f_s is the sampling frequency.

2.4.2 Trivial highpass

This is structured quite similarly to the lowpass, only the output calculation differs.

The output y changes according to the following pseudo-code (x1 is the new input, x0 is the previous input):

```
y := (y + x1 - x0) * g
```

where

$$g = \frac{1}{Kf_c + 1}$$
$$K = \frac{2\pi}{f_s}$$

f_c is the cutoff frequency and f_s is the sampling frequency.

2.4.3 Four-pole versions

Low- and high-pass filters are realized by implementing the digital moog filter as described in [2]. The feedback delay compensation is used, and the filter coefficients (such as the frequency) are compensated accordingly. Our implementation is missing the non-linearization effect, though. We implemented two filters, 4-pole lowpass and 4-pole highpass. The resonance is also there, but it seems a bit buggy. It does not affect the signal as much as would be expected. See more about this in the source code.

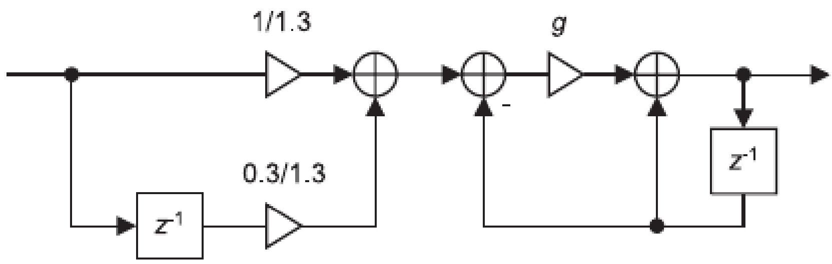

Figure 2: A compensated LP filter (from [2])

2.5 ADSR envelope

The output volume of each channel is modulated with an attack-decay-sustain-release envelope generator. This makes the plain volume sound more instrument-like, when the volume jumps first up and then decays slowly to some level, imitating how real instruments are used. When a key is released, the volme decays

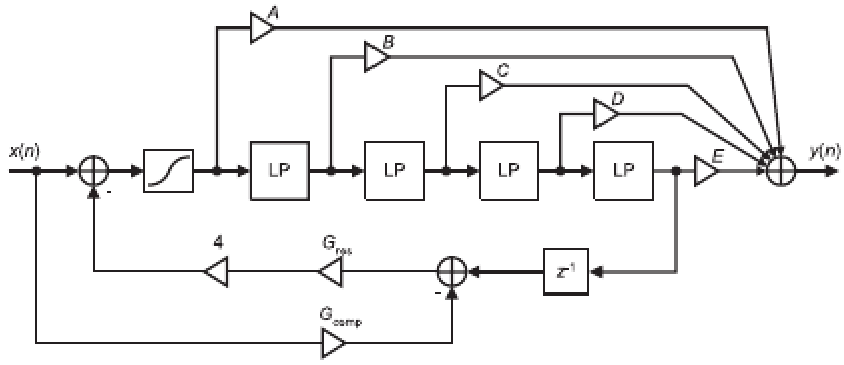

Figure 3: The 4-pole filter structure (from [2])

slowly to zero. A normal ADSR is plotted in figure 4. Our implementation consists of three separate stages: attack, decay, and release. Four parameters are used: time coefficients for attack, decay, and release, and a volume level for sustain. The envelope value changes exponentially in each stage towards a preset target value.

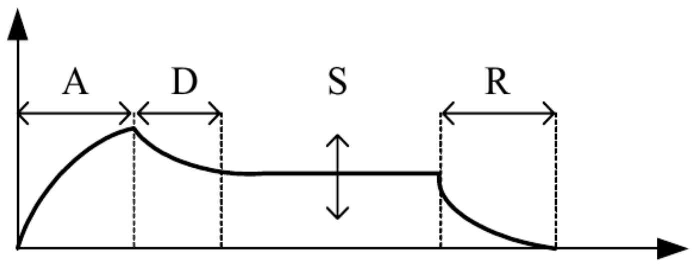

Figure 4: The adsr envelope output level, as a function of time

Attack and release stages have their time constant target beyond the actual value, i.e. if they would run infinitely, the envelope value would overflow; this is because an exponentially decaying function never actually reaches its target exactly. After the specified attack time, our envelope will reach 1; after the release time, it will go to zero. The decay phase goes (virtually) infinitely long towards the sustain level. Finite-precision calculation makes this stop at some time, but it's not noticeable by human ear.

Filter parameters are specified in the following table.

name	Specified as	Used and stored as
attack	Time	Modified LP coefficient
decay	Time	LP coefficient
sustain	Level	Final note volume
release	Time	Modified LP coefficient

An LP filter, i.e. exponentially decaying function, is used as follows:

```
state += g * (target - state)
```

The coefficient g is computed from the time where approx. 63 % (we'll call this λ) of the target value is reached (a time constant of an RC circuit represents the time it takes for the step response to reach $1 - 1/e$ of the target value):

$$g = 1 - e^{\frac{-1}{T * f_c}}$$

Because of the natural decay constant λ, the attack and release target values are computed by multiplying the target value by

$$\lambda = \frac{e}{e-1} \approx 0.63$$

so that λ of this new target is actually what we want (from zero to 1 in attack phase, or from the current value to 0 in release phase). The actual target t is computed from the wanted value w when starting value is s with

$$\begin{aligned} \lambda(t - s) &= w - s \\ t - s &= (w - s)/\lambda \\ t &= s + (w - s)/\lambda \end{aligned}$$

In attack phase, this is

$$\begin{aligned} t &= 0 + (1 - 0)/\lambda \\ &= 1/\lambda \approx 1.58 \end{aligned}$$

and in release phase when the starting value s represents the current state, the target becomes

$$\begin{aligned} t &= s + (0 - s)/\lambda \\ &= (1 - \lambda)s \end{aligned}$$

Because the magnitudes of these coefficients will be over 1 and fixed-point calculation of the DSP deals with values between -1 and 1, and also the subtraction $1 - (-1)$ does not fit between $[-1, 1)$, we divide everything by 2 in the computation stage of the ADSR, and finally multiply by 2 when the final value has been obtained.

2.6 Modulation

In addition to the output ADSR, the instruments' oscillators and filters can be modulated with an ADSR, or an LFO. Because the instruments are implemented by hard-coding the signal handling in assembly, it's possible to multiply their outputs with a modulator or even change their state coefficients over time. As an example, we coded several instruments demonstrating this:

- filter cutoff modified with an ADSR
- filter cutoff modified with an LFO sinewave
- pulse oscillator duty cycle modified with an ADSR.

2.7 Control interface

The actual sound rendering code is in pure DSP assembly, but interfacing to the real world is done with the help of the ColdFire microcontroller. Code for it is written in C. In this chapter, the code and the user interface is described.

The microcontroller code runs several RTEMS tasks:

- panel interface handling
- midi reading
- DSP debugging reading
- Sequencer tracking

2.7.1 Panel interface

The user interface is as follows:

Shift key Panic button: kill all notes and clear the sequencer memory. Kind of a soft reset.

Edit Enable the sequencer recording and playback.

Part up Currently edited midi channel up.

Part down Currently edited midi channel down.

Group up Channel mapping up.

Group down Channel mapping down.

Page up Currently edited pot up.

Page down Currently edited pot down.

Param up Pot tunable up.

Param down Pot tunable down.

Value up Unused.

Value down Test note key.

The potentiometer tunables are as follows:

1	1st instru adsr A
2	1st instru adsr D
3	1st instru adsr R
4	1st instru filt cutoff
5	2nd instru filt base
6	2nd instru filt sine freq
7	3rd instru filt adsr A
8	3rd instru filt adsr D
9	3rd instru filt adsr R
a	4th instru dutycycle base
b	4th instru dutycycle amplitude
c	5th instru filt cutoff
d	6th instru filt cutoff
e	6th instru filt resonance
f	7th instru filt cutoff

Not much thought is given to these. For example, the units are not scaled very consistently, but they just are tuned so that everything sounds good enough.

The original code had a buggy sine amplitude tunable at #6. It is replaced with the sine frequency in the final code that also was in the demo.

The midi channel and potentiometer mappings are somewhat clumsy to use. First the currently edited item is selected with the "part" or "page" keys for channels or pots, and then the selected value can be rotated with "group" or "param" keys, respectively. Only the eight first midi channels can be mapped to synth instruments, so try to configure your midi keypad to one of these. Our keypad always sent its midi events to midi channel 0.

The value down key acts as one midi key at midi channel 0. The encoder turns the test note's note number up and down. There is no safety restrictions on changing the value beyond the [0,127] range.

The volume potentiometer works as expected.

The panel LCD display looks like this:

```
TxAaBbCc 01234567
NNNNNNNN QWERTYUI
```

where a, b, and c represent the first, second and third pot tunable mapping, respectively; N's mean the debugging value from the DSP (the runtime of a single sample, in cycles); QWERTYUI means the numbers for instruments for midi channels that are above on the first row, e.g. Q is the synth instrument for midi channel 0. The currently selected channel and tunable are blinking.

2.7.2 Midi code

The midi task simply reads events from the MidiShare library and sends "note on" and "note off" events to the DSP via interrupts and the data port.

2.7.3 DSP value reading

This is a task from the template code to display words from the DSP on the panel. We output the clock cycles it took to render the last sample, to keep track of the code complexity.

2.7.4 Sequencer

The sequencer, when turned on, loops an array of fixed size 16, and sends all recorded key on or key off events at each slot to the DSP as if they were plain MIDI events. The event array consists of linked lists of event structures. The events are recoded when a MIDI event is handled. There is space for a maximum of 64 events total.

3 Self-assessment

Since there were only two of us working on this assignment, the amount of work per group member was somewhat higher, which was to be expected. The initial channel management and sample generation structure along with some simple oscillators and filters, as well as the initial rudimentary panel interface and later also a simple MIDI handling (in the C code), were written by Nuutti. At this time Konsta was not available for coding. Later, Konsta improved these, and lately wrote better implementations of the more mathematical DSP stuff (oscillators, filters, ADSRs etc.). Nuutti wrote the sine LFO and the noise oscillator. Much of the C code and all the multipole filters were kind of hacked together during the last evenings by Konsta, and are not guaranteed to be bug-free. We feel that the workload was divided pretty fairly, while Konsta did a bit more work because he had more time and experience.

4 Conclusions

We are quite happy with how the synth turned out; we feel it became more or less what we initially planned. In retrospect, it would have likely been better to take a block-based approach instead of the current per-sample generation. This would have allowed us to better take advantage of the DSP architecture. Of course, that would have meant a short delay between a key-press and the produced sound, but with a block size of e.g. a few dozens of samples, the delay would have been unnoticeable. We noticed that the MonoSynth example code that came with the Chameleon SDK used a block-based approach.

As it stands now with our synth, depending on the instrument, playing about five or so notes at the same time can cause the workload to be too heavy, effectively resulting in half-speed output that sounds lower than normal. This mostly happens with complex instruments.

In the end, we didn't feel that the DSP's assembly was all that different from that of traditional processors. The main differences, namely, the separate X and Y memory spaces and the MAC instruction, could have put to better use had we taken the block-based approach.

5 Appendices

Program code. Better readable in the accompanying files.

Listing 1: code/main.c

```
/*******************************************************************
 * C H A M E L E O N   ColdFire C file                             *
 *******************************************************************
 * Digimoog project for Aalto ELEC S-89.3510 SP NK                 *
 * By Konsta and Nuutti H ltt  2013                                *
 * Based on:                                                       *
 * Project work template for sample-based audio input and output   *
 * Based on the example dspthru by Soundart                        *
 * Hannu Pulakka, March 2006, February 2007                        *
 * Modified by Antti Pakarinen, February, 2012                     *
 *      (Panel input and communication routines)                   *
 *******************************************************************/


/* Usage: see the report pdf */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```
#include <string.h>
#include <rtems.h>
#include <midishare.h>
#include <chameleon.h>

#include "dsp/dsp_code.h"

#include "seq.h"


#define RATE 48000
#define DT (1.0 / RATE)
#define PI 3.14159265
#define FILT_K (DT * 2 * PI)

enum Key {
        KEY_VALUE_DOWN,
        KEY_PARAM_DOWN,
        KEY_VALUE_UP,
        KEY_PARAM_UP,

        KEY_PAGE_DOWN,
        KEY_GROUP_DOWN,
        KEY_PAGE_UP,
        KEY_GROUP_UP,

        KEY_PART_DOWN,
        KEY_SHIFT,
        KEY_PART_UP,
        KEY_EDIT,
};

// number of keys that play a note instead of controlling something
#define NOTE_KEYS 1

// Required definitions for a Chameleon application
/*******************************************************************/

#define WORKSPACE_SIZE  128*1024
rtems_unsigned32 rtems_workspace_size = WORKSPACE_SIZE;
rtems_unsigned32 rtems_workspace_start[WORKSPACE_SIZE];
/*******************************************************************/


// Handles of the panel and the DSP
static int panel, dsp;

static volatile int seqtick, seqevs, seqenabled;
static rtems_unsigned32 encoval;

// midi channel (0..7) to synth instrument (1..7) mapping
// if the value is 0, all events to this channel are ignored
// several midi channels can be assigned to the same instrument
// otherwise, synth_idx = midichan_to_synth[midichan] - 1
// note that the program change events are not used for anything
// the keypad buttons work at channel 0
#define SYNTH_INSTRUS 7
#define MIDI_CHAN_MAP_SIZE 8
static int midichan_to_synth[MIDI_CHAN_MAP_SIZE];
static int midichanedit;

// pot to tunable mapping in the same way as midi channels
```

```
// three pots, TUNABLES_SIZE parameters
// these in InstruTunables array in instruparams.asm
#define TUNABLES_SIZE (0xe + 1)
static int pot_to_tunable[3];
static int tunableedit;

// This function is called if an unexpected error occurs
static void Error(char *error) {
        TRACE(error);
        exit(1);
}

// Show a data word on the LCD display
static void show_data(rtems_signed32 data) {
        char str[9];
        sprintf(str, "%06X", data);
        panel_out_lcd_print(panel, 1, 0, str);
}

static char dbgbuf[32];

static void DSP_write_cmd(rtems_unsigned32 vecnum) {
        sprintf(dbgbuf, "DSP_write_cmd %d\n", vecnum); TRACE(dbgbuf);
        if (!dsp_write_command(dsp, vecnum / 2, TRUE))
                Error("ERROR: cannot write command to DSP.\n");
}

static void DSP_write_cmd_data(rtems_unsigned32 vecnum,
    rtems_unsigned32 data) {
        sprintf(dbgbuf, "DSP_write_cmd %d %d\n", vecnum, data); TRACE(
            dbgbuf);
        if (!dsp_write_data(dsp, &data, 1))
                Error("ERROR: cannot write data to DSP.\n");
        DSP_write_cmd(vecnum);
}

static void DSP_write_cmd_data2(rtems_unsigned32 vecnum,
    rtems_unsigned32 data1, rtems_unsigned32 data2) {
        sprintf(dbgbuf, "DSP_write_cmd_data2 %d %d %d\n", vecnum, data1
            , data2); TRACE(dbgbuf);
        if (!dsp_write_data(dsp, &data1, 1))
                Error("ERROR: cannot write data to DSP.\n");
        DSP_write_cmd_data(vecnum, data2);
}


// FIXME: cannot send three words, would get stuck (?!)
// hangs all threads, even the blinking led stops, wtf
static void DSP_write_cmd_data3(rtems_unsigned32 vecnum,
    rtems_unsigned32 data1, rtems_unsigned32 data2, rtems_unsigned32
    data3) {
        // HACK used only in sending the note on events
        // note number fits well in the lower 8 bits
        rtems_unsigned32 juttu;
        sprintf(dbgbuf, "DSP_write_cmd_data3 %d %d %d\n", vecnum, data1
            , data2); TRACE(dbgbuf);
        juttu = float_to_fix_round(data3 / 127.0);
        DSP_write_cmd_data2(vecnum, data1, data2 | (juttu & 0xffff00));
}


// Initialization of the panel and the DSP
```

```
void initialize()
{
    // Initialize panel and DSP
    panel = panel_init();
    if (!panel)
        Error("ERROR: cannot access the panel.\n");

    dsp = dsp_init(1, dspCode);
    if (!dsp)
        Error("ERROR: cannot access the DSP.\n");

    panel_out_lcd_print(panel, 0, 0, "digimoog");
}

// Functions for transforming potentiometer values to
// parameter-specific ranges
static rtems_unsigned32 lowpass_pot(rtems_unsigned32 pot) {
        float freq = (float)pot / 0xffffff * 16000.0;
        float c = (FILT_K * freq) / (FILT_K * freq + 1);
        return c * 0x7fffff;
}

static rtems_unsigned32 lowpass_dif(rtems_unsigned32 pot) {
        // NOTE: THIS IS BROKEN!
        // This would need the frequency also, of course
        // potentiometer value should set the deviation amplitude
        // the coefficient for that is a slope value computed from the
            frequency
        float freq = (float)pot / 0xffffff * 16000.0;
        float c = FILT_K / ((FILT_K * freq + 1) * (FILT_K * freq + 1));
        c *= 1000; // max amplitude
        return c * 0x7fffff;
}

static rtems_unsigned32 multipole_pot(rtems_unsigned32 pot) {
        float freq = (float)pot / 0xffffff * 8000.0;
        float c = 2 * PI * freq / RATE;
        return c * 0x7fffff;
}

static rtems_unsigned32 hihpass_pot(rtems_unsigned32 pot) {
        float freq = (float)pot / 0xffffff * 16000.0;
        float c = 1.0 / (FILT_K * freq + 1);
        return c * 0x7fffff;
}

static rtems_unsigned32 adsr_time(rtems_unsigned32 pot) {
        float time_secs = (float)pot / 0xffffff * 0.5;
        float c = 1 - exp(-1.0 / (time_secs * RATE));
        return c * 0x7fffff;
}

// MIDI input event handler, also save to the sequencer
static void synth_note_off(int notenum, int midichan) {
        if (midichan >= 0 && midichan < MIDI_CHAN_MAP_SIZE) {
                int synthinstru = midichan_to_synth[midichan] - 1;
                if (synthinstru != -1) {
                        DSP_write_cmd_data2(
                            DSPP_VecHostCommandMidiKeyOff, notenum,
                            synthinstru);
                        if (seqenabled)
```

```
                                seqevs += seq_add_event(seqtick,
                                    synthinstru, SEQ_EVTYPE_KEYOFF,
                                    notenum);
                }
        }
}

// MIDI input event handler, also save to the sequencer
static void synth_note_on(int notenum, int midichan, int velocity) {
        if (midichan >= 0 && midichan < MIDI_CHAN_MAP_SIZE) {
                int synthinstru = midichan_to_synth[midichan] - 1;
                if (synthinstru != -1) {
                        DSP_write_cmd_data3(
                            DSPP_VecHostCommandMidiKeyOn, notenum,
                            synthinstru, velocity);
                        if (seqenabled)
                                seqevs += seq_add_event2(seqtick,
                                    synthinstru, SEQ_EVTYPE_KEYON,
                                    notenum, velocity);
                }
        }
}

// Handle the physical keypads on the chameleon panel
static void keydown(enum Key key) {
        switch (key) {
        case KEY_SHIFT:
                seq_init();
                seqevs = 0;
                DSP_write_cmd(DSPP_VecHostCommandPanic);
                break;
        case KEY_EDIT:
                seqenabled ^= 1;
                break;
        case KEY_PART_UP:
                midichanedit = midichanedit == MIDI_CHAN_MAP_SIZE-1 ? 0
                     : midichanedit + 1;
                break;
        case KEY_PART_DOWN:
                midichanedit = midichanedit == 0 ? MIDI_CHAN_MAP_SIZE-1
                     : midichanedit - 1;
                break;
        case KEY_GROUP_UP:
                midichan_to_synth[midichanedit]++;
                if (midichan_to_synth[midichanedit] == SYNTH_INSTRUS +
                    1)
                        midichan_to_synth[midichanedit] = 0;
                break;
        case KEY_GROUP_DOWN:
                midichan_to_synth[midichanedit]--;
                if (midichan_to_synth[midichanedit] == -1)
                        midichan_to_synth[midichanedit] = SYNTH_INSTRUS
                            ;
                break;
        case KEY_PAGE_UP:
                tunableedit = tunableedit == 2 ? 0 : tunableedit + 1;
                break;
        case KEY_PAGE_DOWN:
                tunableedit = tunableedit == 0 ? 2 : tunableedit - 1;
                break;
        case KEY_PARAM_UP:
                pot_to_tunable[tunableedit]++;
```

```
                if (pot_to_tunable[tunableedit] == TUNABLES_SIZE+1)
                        pot_to_tunable[tunableedit] = 0;
                break;
        case KEY_PARAM_DOWN:
                pot_to_tunable[tunableedit]--;
                if (pot_to_tunable[tunableedit] == -1)
                        pot_to_tunable[tunableedit] = TUNABLES_SIZE;
                break;
        default:
                if (key < NOTE_KEYS)
                        synth_note_on((int)key + NOTE_KEYS * encoval,
                            0, 50);
        }
}
static void keyup(enum Key key) {
        if (key < NOTE_KEYS)
                synth_note_off((int)key + NOTE_KEYS * encoval, 0);
}

static rtems_signed32   volume_table[128];
static rtems_signed32   linear_table[128];

// Manually hard-coded by the tunable array in the assembly code
// See the pointers in the corresponding assembly array for details
void update_tunable(int i, int potvalue) {
        int tunable;
        rtems_unsigned32 sendval;

        tunable = pot_to_tunable[i];
        if (tunable != 0) {
                tunable -= 1;
                switch (tunable) {
                        case 0x0:
                        case 0x1:
                        case 0x2: sendval = adsr_time(linear_table[
                            potvalue]); break;
                        case 0x3: sendval = lowpass_pot(volume_table[
                            potvalue]); break;
                        case 0x4: sendval = lowpass_pot(volume_table[
                            potvalue]); break;
                        case 0x5: sendval = linear_table[potvalue];
                            break;
                        case 0x6:
                        case 0x7:
                        case 0x8: sendval = adsr_time(linear_table[
                            potvalue]); break;
                        case 0x9:
                        case 0xa: sendval = linear_table[potvalue];
                            break;
                        case 0xb: sendval = hihpass_pot(volume_table[
                            potvalue]); break;
                        case 0xc:
                        case 0xd:
                        case 0xe: sendval = multipole_pot(volume_table[
                            potvalue]); break;
                        default: Error("Bad tunable"); break;
                }

                DSP_write_cmd_data2(DSPP_VecHostCommandUpdateTunable,
                    tunable, sendval);
        }
}
```

```
// Panel task: interaction with the Chameleon panel
static rtems_task panel_task(rtems_task_argument argument)
{
        rtems_unsigned32        key_bits, key_bits_prev;
        rtems_unsigned8         potentiometer;
        rtems_unsigned8         encoder;
        rtems_signed8           increment;
        char                    text[17];

        rtems_unsigned8 value;
        int i;
        float dB;

        TRACE("digimoog");

        // Precalculate gain values for different volume settings
        for (i = 0; i < 128; i++) {
                if (i < 27)
                        dB = -90.0 + (float)40.0 * i/27.0;
                else
                        dB = -50.0 + (float)50.0 * (i-27)/100.0;
                volume_table[i] = float_to_fix_round(pow(10.0, dB/20.0)
                    );
        }
        volume_table[0] = 0;

        // Precalculate a linear table to scale the potentiometer
            values linearily between 0..~1
        for (i = 1; i < 128; i++) {
                linear_table[i]=float_to_fix_round((float)i/127.0);
        }

        key_bits_prev = 0;
        encoval = 0;

        // Main loop
        while (TRUE) {
                //Poll for panel events
                if (!panel_in_new_event(panel, TRUE))
                        Error("ERROR: unexpected exit waiting new panel
                             event.\n");

                if (panel_in_potentiometer(panel, &potentiometer, &
                    value)) {
                        switch (potentiometer)
                        {
                        case PANEL01_POT_VOLUME:
                                DSP_write_cmd_data(
                                    DSPP_VecHostCommandUpdateVolume,
                                    volume_table[value]);
                                break;
                        case PANEL01_POT_CTRL1:
                                update_tunable(0, value);
                                break;
                        case PANEL01_POT_CTRL2:
                                update_tunable(1, value);
                                break;
                        case PANEL01_POT_CTRL3:
                                update_tunable(2, value);
                                break;
                        default:
```

```
                                        break;
                                }
                        } else if (panel_in_keypad(panel, &key_bits)) {
                                rtems_unsigned32 key_diff = key_bits ^
                                    key_bits_prev;
                                rtems_unsigned32 mask = 0x80000000;
                                int key = 0;

                                key_bits_prev = key_bits;
                                while (key_diff) {
                                        if (key_diff & mask) {
                                                // last 4 (8..11) are shifted
                                                    by 4, move 12 -> 8 etc
                                                if (key_bits & mask)
                                                        keydown(key < 8 ? key :
                                                             key - 4);
                                                else
                                                        keyup(key < 8 ? key :
                                                            key - 4);
                                                key_diff ^= mask;
                                        }
                                        key++;
                                        mask >>= 1;
                                }
                        } else if (panel_in_encoder(panel, &encoder, &increment
                            )) {
                                encoval += increment;
#if 0
                                sprintf(text, "Encoder: %+3d ", increment);
                                if(increment > 0)
                                        DSP_write_cmd(
                                            DSPP_VecHostCommandEncoderUp);
                                else
                                        DSP_write_cmd(
                                            DSPP_VecHostCommandEncoderDown);
                                panel_out_lcd_print(panel, 0, 0, text);
                                panel_out_lcd_print(panel, 1, 0, "
                                                   ");
#endif
                        }
        }

        panel_exit(panel);
        rtems_task_delete(RTEMS_SELF);
}

#define EVENT_MIDI RTEMS_EVENT_1

static void receive_alarm(short ref)
{
        rtems_event_send((rtems_id) MidiGetInfo(ref), EVENT_MIDI);
}

// Midi task: receive midi events from MidiShare and send to dsp
static rtems_task midi_task(rtems_task_argument ignored)
{
        MidiEvPtr               ev;
        rtems_event_set         pending;
        rtems_status_code       status;
        rtems_id                task_id;
        short                   ref_midi;
        char                    debugmsg[32];
```

```
        ref_midi = MidiOpen("Synth");
        if (ref_midi < 0)
        {
                TRACE("ERROR: cannot open MidiShare.\n");
                rtems_task_delete(RTEMS_SELF);
        }

        rtems_task_ident(RTEMS_SELF, 0, &task_id);

        MidiSetInfo(ref_midi, (void *) task_id);
        MidiSetRcvAlarm(ref_midi, receive_alarm);

        MidiConnect(0, ref_midi, TRUE);

        while (TRUE)
        {
                status = rtems_event_receive(
                        EVENT_MIDI,
                        RTEMS_WAIT | RTEMS_EVENT_ANY,
                        RTEMS_NO_TIMEOUT,
                        &pending
                );
                if (status != RTEMS_SUCCESSFUL)
                        break;

                while ((ev = MidiGetEv(ref_midi)) != NULL) {
                        if (EvType(ev) == typeKeyOff || (EvType(ev) ==
                            typeKeyOn && Vel(ev) == 0)) {
                                synth_note_off(Pitch(ev), Chan(ev));
                        } else if (EvType(ev) == typeKeyOn) {
                                synth_note_on(Pitch(ev), Chan(ev), Vel(
                                    ev));
                        }
                        sprintf(debugmsg, "MIDI:type=%d chan=%d key=%d
                            vel=%d\n", EvType(ev), Chan(ev), Pitch(ev),
                             Vel(ev));
                        TRACE(debugmsg);
                }
        }

        MidiConnect(0, ref_midi, FALSE);

        MidiClose(ref_midi);

        rtems_task_delete(RTEMS_SELF);
}

// Read task: read data from the DSP
static rtems_task read_task(rtems_task_argument ignored)
{
  rtems_signed32 data;
  rtems_boolean res;

  while (TRUE) {
    res = dsp_read_data(dsp, &data, 1);
    if (res) {
      data &= 0x00FFFFFF;       //clear the sign extension
      show_data(data);

      // *** You can implement your own data handling here ***
```

```
    }
  }

  rtems_task_delete(RTEMS_SELF);
}

// Sequencer handling: read the recorded notes and play them to the
    synth
// Also display some useful information on the panel
static rtems_task seq_task(rtems_task_argument ignored) {
        int bpm = 4;
        rtems_interval secticks, period;
        struct seqevent* ev;
        char text[20];
        int n;

        rtems_clock_get(RTEMS_CLOCK_GET_TICKS_PER_SECOND, &secticks);
        period = secticks / bpm;

        seq_init();

        while (TRUE) {
                panel_out_led(panel, PANEL01_LED_EDIT | (seqenabled ?
                    PANEL01_LED_SHIFT : 0));
                ev = seq_events_at(seqtick);
                n = 0;
                while (ev) {
                        switch (ev->type) {
                        case SEQ_EVTYPE_KEYON:
                                DSP_write_cmd_data3(
                                    DSPP_VecHostCommandMidiKeyOn, ev->
                                    param1, ev->instrument, ev->param2)
                                    ;
                                break;
                        case SEQ_EVTYPE_KEYOFF:
                                DSP_write_cmd_data2(
                                    DSPP_VecHostCommandMidiKeyOff, ev->
                                    param1, ev->instrument);
                                break;
                        }
                        ev = ev->next;
                        n++;
                }
                sprintf(text, "%xA%xB%xC%x 01234567", seqtick & 15,
                    pot_to_tunable[0], pot_to_tunable[1],
                    pot_to_tunable[2]);
                panel_out_lcd_print(panel, 0, 0, text);
                sprintf(text, "%d%d%d%d%d%d%d%d",
                        midichan_to_synth[0],
                        midichan_to_synth[1],
                        midichan_to_synth[2],
                        midichan_to_synth[3],
                        midichan_to_synth[4],
                        midichan_to_synth[5],
                        midichan_to_synth[6],
                        midichan_to_synth[7]
                        );
                panel_out_lcd_print(panel, 1, 8, text);
                strcat(text,"\n");
                //TRACE(text);

                seqtick++;
```

```
                rtems_task_wake_after(period/2);
                panel_out_led(panel, seqenabled ? PANEL01_LED_SHIFT :
                    0);
                panel_out_lcd_print(panel, 0, 2 + 2 * tunableedit, " ")
                    ;
                panel_out_lcd_print(panel, 1, 8 + midichanedit, " ");
                rtems_task_wake_after(period/2);
        }

        rtems_task_delete(RTEMS_SELF);
}
rtems_boolean create_task(rtems_task (*task)(rtems_task_argument),
    const char *name) {
        rtems_id task_id;
        rtems_status_code status;

        status = rtems_task_create(
                        rtems_build_name(name[0], name[1], name[2],
                            name[3]),
                        50,
                        RTEMS_MINIMUM_STACK_SIZE,
                        RTEMS_DEFAULT_MODES,
                        RTEMS_DEFAULT_ATTRIBUTES,
                        &task_id
                        );
        if (status != RTEMS_SUCCESSFUL) {
                TRACE("ERROR: cannot create "); TRACE(name); TRACE("
                    seq_task.\n");
                return FALSE;
        }
        status = rtems_task_start(task_id, task, 0);
        if (status != RTEMS_SUCCESSFUL) {
                TRACE("ERROR: cannot start "); TRACE(name); TRACE("
                    seq_task.\n");
                return FALSE;
        }
        return TRUE;
}

// The main function that is called when the application is started
rtems_task rtems_main(rtems_task_argument ignored)
{
    initialize();
        create_task(panel_task, "PANE");
        create_task(midi_task, "MIDI");
        create_task(read_task, "READ");
        create_task(seq_task, "SEQR");
    rtems_task_delete(RTEMS_SELF);
}
```

Listing 2: code/seq.c

```
#include <string.h>
#include "seq.h"

#define SEQ_BUFSIZE 64
#define SEQ_LENGTH 16
#define SIZEMASK(sz) (sz - 1)
#define ARRMOD(val, sz) ((val) & SIZEMASK(sz))

// storage for everything
struct seqevent seqstore[SEQ_BUFSIZE];
// the linked lists
```

```
struct seqevent* seqqueue[SEQ_LENGTH];
// Index to next free item in seqstore
int next_free;

// Get space for one event, or NULL if no available
static struct seqevent* alloc_ev(void) {
        struct seqevent* ret;
        int next;

        if (next_free == -1)
                return NULL;

        ret = &seqstore[next_free];
        ret->flags = SEQ_FLAG_USED;

        next = ARRMOD(next_free + 1, SEQ_BUFSIZE);
        while ((seqstore[next].flags & SEQ_FLAG_USED) && (next !=
            next_free)) {
                next = ARRMOD(next + 1, SEQ_BUFSIZE);
        }
        next_free = next != next_free ? next : -1;

        return ret;
}

int seq_add_event(int time, int instrument, int type, rtems_unsigned32
    param) {
        seq_add_event2(time, instrument, type, param, 0);
}
// Add an event to a particular time slot for one instrument
int seq_add_event2(int time, int instrument, int type, rtems_unsigned32
     param1, rtems_unsigned32 param2) {
        struct seqevent* next = alloc_ev();
        struct seqevent* queue;

        if (!next)
                return 0;

        next->instrument = instrument;
        next->type = type;
        next->param1 = param1;
        next->param2 = param2;

        queue = seqqueue[ARRMOD(time, SEQ_LENGTH)];
        if (queue) {
                while (queue->next) {
                        queue = queue->next;
                }
                queue->next = next;
        } else {
                seqqueue[ARRMOD(time, SEQ_LENGTH)] = next;
        }
        return 1;
}

// Get the first item of the linked list at "time"
struct seqevent* seq_events_at(int time) {
        return seqqueue[ARRMOD(time, SEQ_LENGTH)];
}

// Clear the sequencer state
void seq_init(void) {
```

```
        memset(seqstore, 0, sizeof(seqstore));
        memset(seqqueue, 0, sizeof(seqqueue));
        next_free = 0;
}
```

Listing 3: code/seq.h

```
#ifndef SPANK_SEQ_H
#define SPANK_SEQ_H

#include <rtems.h>

#define SEQ_EVTYPE_KEYON 0
#define SEQ_EVTYPE_KEYOFF 1

#define SEQ_FLAG_USED 1

struct seqevent {
        int instrument;
        int type;
        rtems_unsigned32 param1, param2;
        struct seqevent* next;
        int flags;
};

int seq_add_event(int time, int instrument, int type, rtems_unsigned32
    param);
int seq_add_event2(int time, int instrument, int type, rtems_unsigned32
     param1, rtems_unsigned32 param2);
struct seqevent* seq_events_at(int time);
void seq_init(void);

#endif
```

Listing 4: code/main.asm

```
;*******************************************************************
; C H A M E L E O N   DSP Assembler file                           *
;*******************************************************************
; Digimoog project for Aalto ELEC S-89.3510 SP NK                  *
; Virtual analog synthesizer                                       *
;                                                                  *
; By Konsta H lt t  and Nuutti H lt t  2013                        *
;                                                                  *
; Some basic stuff based on:                                       *
; Project work template for sample-based audio I/O (polling)       *
; Based on the example dspthru by Soundart                         *
; Hannu Pulakka, March 2006, February 2007                         *
; Modified by Antti Pakarinen, February 2012, update in March 2012 *
; Registers r7+n7 are reserved for interrupt routines as default   *
;*******************************************************************

        nolist
        page    255,0
        opt     MU,S,CC,CEX,MEX,MD
        list

        nolist
        include "SDK\include\dsp\dsp_equ.asm"
        list

;*******************************************************************
; The following definition switches between a simulator version and
```

```
; a real-time version of the program. Set this to '1' if you are
; analyzing the program with the simulator, and to '0' if you are
; running the program in Chameleon.
;
; The definition is used later in this assembly file to skip or
; include sample synchronization, which does not work in the simulator
; but is essential for correct operation in Chameleon.

        define  simulator       '0'

;**********************************************************************

PI      equ     3.14159265
RATE    equ     48000
DT      equ     1.0/RATE

        include 'oscinc.asm'
        include 'filtinc.asm'
        include 'multipoleinc.asm'
        include 'adsrinc.asm'
        include 'sininc.asm'

; ChannelCapacity is the fixed size for each channel.
; Depending on the actual oscillator and filter state sizes, this may
    be
; more than needed, but doesn't matter. NOTE: this must be increased
; if it's not enough for some oscillator+filter combination.
ChannelCapacity  equ 63 ; words per one channel
OscStateCapacity equ 25 ; NOTE: just a constant sized block, hope that
    no one is bigger
NumChannels      equ 10 ; maximum number of playable notes at a time

; Channel structure variable indices
ChDataIdx_Note          equ 0
ChDataIdx_FiltStateAddr equ 1 ; pointer to beginning of the filter
    state, deprecated
ChDataIdx_AdsrState     equ 2
ChDataIdx_InstruPtr     equ (2+AdsrStateSize)
ChDataIdx_InstruIdx     equ (2+AdsrStateSize+1)
ChDataIdx_Velocity      equ (2+AdsrStateSize+2)
ChDataIdx_OscState      equ (2+AdsrStateSize+3)

ChDataIdx_FiltState     equ (ChDataIdx_OscState+OscStateCapacity)

; Bit flags in the note number if the channel is in a special state
; Note that these cripple the actual note value, but it's not used
    anymore
; when the channel has been set to key off state
ChNoteDeadBit           equ 23
ChNoteKeyoffBit         equ 22

;**********************************************************************
; Memory allocations
;**********************************************************************
        org     X:$000000

; Starting at ChannelData, there is data for NumChannels channels; each
     has a ChannelCapacity-sized block of memory.
ChannelData: ds NumChannels*ChannelCapacity
AccumBackup ds 3
AccumBackup2 ds 3
AccumBackupLfo ds 3
```

```
        org     Y:$000000
        include 'sin_table.asm' ; NOTE: this must be included here (
            well, it's not quite that strict - see sin_table.asm)

MasterVolume:
        ds      1
NoteThatWentDown: ; If a key just went down, this holds the note value.
     Otherwise, this has highest bit set.
        ds      1
InstrumentThatWentDown: ; If a key just went down, this holds the new
    instrument index for that
        ds      1
NoteThatWentUp:   ; If a key just went up, this holds the note value.
    Otherwise, this has highest bit set.
        ds      1
InstrumentThatWentUp: ; If a key just went up, this holds the
    instrument index for that
        ds      1

; NOTE: the above NoteThatWentDown end NoteThatWentUp currently don't
    support it when several keys
; go down (or up) at about the same time (before the last one has been
    processed).
; Might want to fix this if trouble ensues. Seems to work pretty well,
    though.

PanelKeys_NoteOffset:
        dc 0

;For debug and simulation

        if simulator

OutputL:
        ds      1
OutputR:
        ds      1
OutputMiddle:
        ds 1
OutputAdsr:
        ds 1
OutputOsc:
        ds 1
OutputHax
        ds 1

        endif

; Helper macro for moving registers to debug places
; Only used in simulator
        if simulator
SimulatorMove macro Reg,YDst
        move Reg,Y:(YDst)
        endm
        else
SimulatorMove macro Reg,YDst
        endm
        endif

PanicState:
        ds 1
```

```
        ; must be here, goes to Y memory
        include 'instruparams.asm'
        include 'dpw_coefs.asm'
        include 'saw_ticks.asm'
;**********************************************************************
; Interrupt vectors
;**********************************************************************
        org     P:VecHostCommandDefault
VecHostCommandUpdateVolume:
        JSR     >UpdateVolume
VecHostCommandUpdateTunable:
        JSR     >UpdateTunable
VecHostCommandEncoderUp:
        JSR     >EncoderUp
VecHostCommandEncoderDown:
        JSR     >EncoderDown
VecHostCommandKeyEvent:
        JSR     >KeyEvent
VecHostCommandMidiKeyOn:
        JSR     >MidiKeyOn
VecHostCommandMidiKeyOff:
        JSR     >MidiKeyOff
VecHostCommandPanic:
        JSR     >Panic

;**********************************************************************
; Program code
;**********************************************************************
        org     P:$000100

Start:
        CLR     A
        ; Enable ESSI0 transmit and receive
        BSET    #CRB_TE0,X:<<CRB0               ; Enable Transmit 0
        BSET    #CRB_RE,X:<<CRB0                ; Enable Receive  0
        ;Interrupt enable
        ANDI    #<$FC,MR                        ; Enable interrupts
        BCLR    #SR_I0,SR                       ; Unmask interrupts
        BCLR    #SR_I1,SR
        BSET    #HCR_HCIE,X:<<HCR               ; Enable Host command
            interrupt
        ; Initialize Master Volume variables
        MOVE    A,Y:MasterVolume
        ; Channel synchronization
        if !simulator
          BRSET #SSISR_RFS,X:<<SSISR0,*         ; Wait while receiving
              left frame
          BRCLR #SSISR_RFS,X:<<SSISR0,*         ; Wait while receiving
              right frame
        endif

        ; initialize all channels as dead

        move #>(1<<ChNoteDeadBit),x0
        move #>ChannelData,r1

        do #NumChannels,DeadChannelInitLoopEnd
                move x0,X:(r1+ChDataIdx_Note)
                lua (r1+ChannelCapacity),r1
        DeadChannelInitLoopEnd:
```

```
        ; no note has just went up or down

        move x0,Y:NoteThatWentUp
        move x0,Y:NoteThatWentDown

        ; The cycle count is computed with a free-running timer in the
            background
        ; The counter increments by one every 2 cycles

        ; timer prescale load: just in case, reset the source to clk/2
        move #>0,x0
        move x0,X:TPLR
        ; load reg, start counting from here
        move x0,X:TLR0

        move x0,Y:PanicState

        ; simulate just one keypress
        if simulator
                move #>63,x0 ; 440 hz
                move x0,Y:NoteThatWentDown
                move #>$7fff05,x0 ; velocity (7fff00) and the
                    instrument (5)
                move x0,Y:InstrumentThatWentDown
        endif

MainLoop:
        ; timer handling for computing the cycle count
        ; reset the counter control reg first
        move #>0,x0
        move x0,X:TCSR0
        ; TRM bit (restart mode) cleared -> free running counter
        ; tc0|tc1: mode 3 = event counter (just count clock cycles)
        move #>(TCSR_TC0|TCSR_TC1|TCSR_TE),x0
        move x0,X:TCSR0 ; mode 3, enable
        ; it seems that we need these nops first to correctly count the
             work cycles
        ; (could as well just add 4 to the counter when displaying it)
        nop
        nop
        nop
        nop
        ; timed code seems to start from here
        ; for example, with these nops we get the number 1 out of TCR0
            (with 1 nop, value 0, with 3, value 1 again)
        ;nop
        ;nop
        ;move X:TCR0,a
        ;asl #1,a,a
        ;move a,X:<<HTX

        move #>ChDataIdx_Note,n1 ; NOTE: used in the two following
            loops

        ; check if a key just went up

        brset #23,Y:NoteThatWentUp,NoNoteWentUp
                move Y:NoteThatWentUp,x0
                move Y:InstrumentThatWentUp,x1
                bset #23,Y:(NoteThatWentUp)

                ; find and kill the channel
```

```
        ; (don't actually kill, but turn up the key off bit)
        ; this starts the decay phase, and the ADSR kills this
            channel after having decayed to silence

        move #>ChannelData,r1
        do #NumChannels,ChannelKillLoopEnd
                move X:(r1+ChDataIdx_InstruIdx),b
                cmp x1,b
                bne NotTheNoteToKill
                move X:(r1+ChDataIdx_Note),b
                cmp x0,b
                bne NotTheNoteToKill
                        bset #ChNoteKeyoffBit,X:(r1+n1)
                        enddo
                NotTheNoteToKill:
                nop ; enddo too close
                lua (r1+ChannelCapacity),r1
        ChannelKillLoopEnd:
NoNoteWentUp:

; the mass-murderer panic key kills everything right now
brclr #0,Y:PanicState,PanicLoopEnd
        move #>ChannelData,r1
        do #NumChannels,PanicLoopEnd
                bset #ChNoteDeadBit,X:(r1+n1)
                lua (r1+ChannelCapacity),r1
PanicLoopEnd:
bclr #0,Y:PanicState ; not needed anymore (don't bother
    checking if it was on, just clear)

; check if a key just went down
brset #23,Y:NoteThatWentDown,NoNoteWentDown
        move Y:NoteThatWentDown,n2
        bset #23,Y:(NoteThatWentDown)

        ; find a free channel and initialize there
        ; NOTE: if no free channels are available, the new note
             is just ignored.
AllocChannel:
        move #>ChannelData,r1
        do #NumChannels,ChannelAllocationLoopEnd
                move X:(r1+ChDataIdx_Note),y0
                brclr #ChNoteDeadBit,y0,NotFreeChannel
                        move n2,X:(r1+ChDataIdx_Note)
                        ; r1: workspace pointer
                        ; r4: instrument pointer
                        lua (r1+ChDataIdx_AdsrState),r0
                        bsr AdsrInitState

                        move Y:InstrumentThatWentDown,a
                        and #>$ff,a
                        move Y:InstrumentThatWentDown,b
                        and #>~$ff,b
                        move a,r4
                        move b,X:(r1+ChDataIdx_Velocity)
                        move r4,X:(r1+ChDataIdx_InstruIdx)
                        move Y:(r4+AllInstruments),r4
                        move r4,X:(r1+ChDataIdx_InstruPtr)

                        move Y:(r4+InstruParamIdx_InitFunc),r0
```

```
                        lua (r1+ChDataIdx_OscState+
                            OscStateCapacity),r2
                        move r2,X:(r1+ChDataIdx_FiltStateAddr)

                        ChAlloc_InitInstruState:
                        bsr r0

                        enddo ; too near the loop end
                        nop
                NotFreeChannel:
                lua (r1+ChannelCapacity),r1
        ChannelAllocationLoopEnd:
NoNoteWentDown:

; evaluate channels, sum into b

RenderSample:
clr b
move #>ChannelData,r1

do #NumChannels,ChannelEvaluateLoopEnd
        move X:(r1+ChDataIdx_Note),y0
        brset #ChNoteDeadBit,y0,DeadChannel
                ; Could maybe read this and r1 always before
                    calling
                ; those so they don't need to backup these. it'
                    s just a
                ; couple of cycles.
                move X:(r1+ChDataIdx_InstruPtr),r4

                ; save value of b so far
                ; both accumulators are used by the osc/filt/
                    adsr functions
                move b0,X:(AccumBackup)
                move b1,X:(AccumBackup+1)
                move b2,X:(AccumBackup+2)

                ; evaluate oscillator
                move Y:(r4+InstruParamIdx_OscFunc),r2
                lua (r1+ChDataIdx_OscState),r0
                ChEval_OscEvalBranch:
                bsr r2
                SimulatorMove a,OutputOsc

                ; evaluate filter
                move Y:(r4+InstruParamIdx_FiltFunc),r2
                move X:(r1+ChDataIdx_FiltStateAddr),r0
                ChEval_FiltEvalBranch:
                bsr r2
                SimulatorMove a,OutputMiddle

                ; save a, as it's used in adsr
                move a0,X:(AccumBackup2)
                move a1,X:(AccumBackup2+1)
                move a2,X:(AccumBackup2+2)

                ; compute adsr envelope and apply (multiply by)
                     it
                lua (r4+InstruParamIdx_Adsr),r0
                lua (r1+ChDataIdx_AdsrState),r4
                move X:(r1+ChDataIdx_Note),r2
                bsr AdsrEval
```

```
                SimulatorMove r3,OutputAdsr

                move X:(AccumBackup2),a0
                move X:(AccumBackup2+1),a1
                move X:(AccumBackup2+2),a2

                brclr #23,r3,_notkilled ; negative -> killed
                    flag?
        _killthischannel:
                move #>0,r3
                bset #ChNoteDeadBit,r2
                move r2,X:(r1+ChDataIdx_Note)

        _notkilled:
                move a,x0
                move r3,x1 ; FIXME: return adsr value in x1?
                mpy x0,x1,a ; a *= adsr

                move X:(r1+ChDataIdx_Velocity),x1
                move a,x0
                mpy x0,x1,a

                ; restore b's value and sum the new sample from
                     a (though scaled by 1/NumChannels)
                move X:(AccumBackup),b0
                move X:(AccumBackup+1),b1
                move X:(AccumBackup+2),b2
                move a,x0
                maci #1.0/NumChannels,x0,b
        DeadChannel:

        lua (r1+ChannelCapacity),r1
ChannelEvaluateLoopEnd:

move b,y0

; display the clock ticks on the panel
; FIXME: this replaces the pot readings - bind printing to a
    panel button?
move X:TCR0,a
asl #1,a,a
movep a,X:<<HTX

;Output routines for left Ch
MOVE    Y:MasterVolume,X0               ; Current volume value
    from memory to X0
SimulatorMove Y0,OutputL                ; Move the output value
     to memory for simulator use
MPYR    X0,Y0,B                         ; Multiply the current
    output sample with the current volume value
NOP
if !simulator
BRCLR   #SSISR_TDE,X:<<SSISR0,*         ; Wait for transmit
    register
endif
MOVEP   B,X:<<TX00                      ; Write new output
    sample to the DAC

;Output routines for right Ch
MOVE    Y:MasterVolume,X1               ; Current volume value
    from memory to X0
```

```
        SimulatorMove Y0,OutputR                ; Move the output value
             from Y1 to memory for simulator use
        MPYR    X1,Y0,B                         ; Scale the input
            sample according to the volume curve
        NOP
        if !simulator
        BRCLR   #SSISR_TDE,X:<<SSISR0,*         ; Wait for transmit
            register
        endif
        MOVEP   B,X:<<TX00                      ; Write new output
            sample to the DAC


        BRA     MainLoop

        include 'instrucode.asm'
        include 'adsr.asm'
        include 'osc.asm'
        include 'filt.asm'
        include 'multipole.asm'
        include 'sin.asm'
        include 'isr.asm'
        end     Start
```

Listing 5: code/oscinc.asm

```
; oscillator struct defs

; Saw oscillator contains the increment value (tick) and previous
    output value
SawOscIdx_Tick  equ     0
SawOscIdx_Val   equ     1
SawOscSize      equ     2

DpwOscIdx_Saw   equ     0
DpwOscIdx_Val   equ     SawOscSize ; previous saw^2
DpwOscIdx_Coef  equ     SawOscSize+1
DpwOscSize      equ     SawOscSize+2
; size: 2+2=4

PlsOscIdx_Saw0  equ     0
PlsOscIdx_Saw1  equ     SawOscSize
PlsOscIdx_Duty  equ     2*SawOscSize
PlsOscSize      equ     2*SawOscSize+1 ; TODO: optimize into using just
     one saw and differentiating the second one locally?
; size: 2*2+1=5

PlsDpwIdx_Saw0  equ     0
PlsDpwIdx_Saw1  equ     DpwOscSize
PlsDpwIdx_Duty  equ     2*DpwOscSize ; 0=0% (1:0), 1=50% (1:1)
PlsDpwSize      equ     2*DpwOscSize+1 ; don't optimize this, too much
    copypasta in dpw
; size: 2*5+1=11

NoiseOscIdx_Current equ 0
NoiseOscSize equ 1
```

Listing 6: code/filtinc.asm

```
; trivial lowpass and highpass structures

FiltTrivLpK     equ     (DT*2*PI) ; just a shorthand constant
```

```
; magic coefficient to multiply with
FiltTrivialLpParams     macro   fc
        dc      ((FiltTrivLpK*fc)/(FiltTrivLpK*fc+1))
        endm

; the coefficient for a frequency, and a derivarive of the magic coef
    function
; at the same point - multiplied by the lfo amplitude.

; the second derivative is really small, let's not bother using 2nd
    order
; taylor (yet)
FiltTrivialLpParamsLfo  macro   fc,lfo
        dc      ((FiltTrivLpK*fc)/(FiltTrivLpK*fc+1))
        dc      (FiltTrivLpK/@pow(FiltTrivLpK*fc+1,2))*lfo
        ;dc     (-pow(FiltTrivLpK,2)/@pow(FiltTrivLpK*fc+1,3))
        endm

FiltTrivialLpParamsSize equ 1
FiltTrivialLpParamsLfoSize equ 2

FiltTrivialLpStateSize equ 2


; magic coefficient to multiply with
FiltTrivialHpParams     macro   fc
        dc      (1/(1+FiltTrivLpK*fc))
        endm

FiltTrivialHpParamsSize equ 1

FiltTrivialHpStateSize equ 2
```

Listing 7: code/multipoleinc.asm

```
; Multipole filter structures
; Not much thought given to these
; everything is pretty much the same as in the
; computer music journal paper referenced in the report
Filt4PartStateIdx_x0    equ 0
Filt4PartStateIdx_y0    equ 1
Filt4PartStateSize      equ 2

Filt4StateIdx_Part0     equ 0
Filt4StateIdx_Part1     equ 1*Filt4PartStateSize
Filt4StateIdx_Part2     equ 2*Filt4PartStateSize
Filt4StateIdx_Part3     equ 3*Filt4PartStateSize
Filt4StateIdx_Mem       equ 4*Filt4PartStateSize
Filt4StateIdx_Gres      equ 4*Filt4PartStateSize+1
Filt4StateIdx_Coef      equ 4*Filt4PartStateSize+2

Filt4ParamsIdx_A        equ 0
Filt4ParamsIdx_B        equ 1
Filt4ParamsIdx_C        equ 2
Filt4ParamsIdx_D        equ 3
Filt4ParamsIdx_E        equ 4
Filt4ParamsIdx_Coef     equ 5
Filt4ParamsIdx_Gres     equ 6
Filt4ParamsIdx_Gcomp    equ 7
Filt4ParamsSize         equ 8

Filt4CoefResComp macro coef,res,comp
        dc coef
```

```
        dc res
        dc comp
        endm

Filt4LP4Coefs macro
        dc 0
        dc 0
        dc 0
        dc 0
        dc 1.0
        endm

; NOTE: these do not sum to 1!
; must be scaled back where used
Filt4HP4Coefs macro
        dc  1/8.0
        dc -4/8.0
        dc  6/8.0
        dc -4/8.0
        dc  1/8.0
        endm
```

Listing 8: code/adsrinc.asm

```
; natural constants
E       equ     2.718281828
TGTCOEF equ     E/(E-1) ; ~1,58, ~1/0.63, decay target multiplier to
    get to actual target in a time constant

; use this in instrument definitions
; params: A=time, D=time, S=level, R=time
; NOTE: time 0 gives division by zero, but use some really small value
    instead
; NOTE: D is meaningless if S is 1, obviously
; times in seconds
AdsrParamBlock  macro   At,Dt,Sl,Rt
        dc      (1-@POW(E,-1.0/(At*RATE)))
        dc      (1-@POW(E,-1.0/(Dt*RATE)))
        dc      Sl
        dc      (1-@POW(E,-1.0/(Rt*RATE)))
        endm

AdsrStateSize     equ   4
AdsrParamsSize  equ     4
```

Listing 9: code/sininc.asm

```
LFOSinStateSize equ 3
```

Listing 10: code/sin_table.asm

```
; NOTE: the sine table must be located in Y memory, and its start
    address must
; be a multiple of 2**k, k is an integer such that 2**k >= SinTableSize
    .
; For example, this is trivially satisfied by placing the table to
    start at Y:0.

SinTableSize equ 32 ; NOTE: this must be a power of two

SinTable:
        dupf Index,0,SinTableSize-1
```

```
        dc @SIN(Index*2*PI/SinTableSize)
        endm
```

Listing 11: code/instruparams.asm

```
; Instrument parameters, never changed by DSP code
; These live in the Y memory space
; Runtime state lives in X inside the channel workspaces

; would be easy and handy to rely on that init routines are not needed
    and the
; states would just get zeroed when initializing, but oscillators still
     need at
; least some period magic number - thus, init function for instruments.
; calling convention docs in main.asm so far

; these are used for each instrument always
InstruParamIdx_InitFunc equ     0
InstruParamIdx_OscFunc  equ     1
InstruParamIdx_FiltFunc equ     2
InstruParamIdx_Adsr     equ     3
InstruParamIdx_End      equ     3+AdsrStateSize
; no size constant needed

InstruBassIdx_Lp        equ     InstruParamIdx_End

; pretty stupid to call almost every instrument a bass, but whatever
; this is also quite copy-pasta

; a simple lp-filtered dpw saw
Instrument_Bass:
        dc BassInit-ChAlloc_InitInstruState
        dc OscDpwsawEval-ChEval_OscEvalBranch
        dc BassFilt-ChEval_FiltEvalBranch
        if !simulator
tune1   AdsrParamBlock 0.1,0.1,0.5,0.1
        else
tune1   AdsrParamBlock 0.005,0.005,0.5,0.005 ; faster to debug with
    smaller values
        endif
tune2   FiltTrivialLpParams 5000


; dpw saw, filter cutoff tuned by a sine lfo
Instrument_BassSinLfo:
        dc BassSinLfoInit-ChAlloc_InitInstruState
        dc OscDpwsawEval-ChEval_OscEvalBranch
        dc BassSinLfoFilt-ChEval_FiltEvalBranch
        if !simulator
        AdsrParamBlock 0.1,0.1,0.5,0.1
        else
        AdsrParamBlock 0.005,0.005,0.5,0.005
        endif
tune3   FiltTrivialLpParamsLfo 1200,1000
tune31  dc 0.1

; as above but sine replaced with an adsr
Instrument_BassAdsrLfo:
        dc BassAdsrLfoInit-ChAlloc_InitInstruState
        dc OscDpwsawEval-ChEval_OscEvalBranch
        dc BassAdsrLfoFilt-ChEval_FiltEvalBranch
        AdsrParamBlock 0.1,0.1,0.5,0.1
        FiltTrivialLpParamsLfo 500,3500
```

```
        ; NOTE: R phase >= main adsr R so that gets killed
            appropriately
        ;AdsrParamBlock 2.5,0.1,1.0,1.0
tune4   AdsrParamBlock 0.001,0.2,0.0,1.0

InstruBassAdsrIdx_FiltAdsr      equ     InstruParamIdx_End+
    FiltTrivialLpParamsLfoSize

; pulse wave, no filters, duty cycle adsr'd
Instrument_PulseBass:
        dc PulseBassInit-ChAlloc_InitInstruState
        dc PulseBassOsc-ChEval_OscEvalBranch
        dc PulseBassFilt-ChEval_FiltEvalBranch
        AdsrParamBlock 0.1,0.1,0.5,0.1
        ; NOTE: R phase >= main adsr R so that gets killed
            appropriately
        if !simulator
        AdsrParamBlock 3,0.00000001,1.0,1.0
        else
        AdsrParamBlock 0.03,0.00000001,1.0,1.0
        endif
tune5   dc 0.1 ; base lfo duty cycle
        dc 0.9 ; adsr amplitude

InstruPulseBassIdx_FiltAdsr     equ     InstruParamIdx_End
; base value = where we add lfo stuff to.
InstruPulseBassIdx_DutyBase     equ     InstruParamIdx_End+
    AdsrParamsSize
InstruPulseBassIdx_DutyAmpl     equ     InstruParamIdx_End+
    AdsrParamsSize+1

InstruNoiseIdx_Hp       equ     InstruParamIdx_End

; hp-filtered noise, like a hihat drum
Instrument_Noise:
        dc NoiseInstInit-ChAlloc_InitInstruState
        dc NoiseEval-ChEval_OscEvalBranch
        dc NoiseInstFilt-ChEval_FiltEvalBranch
        if !simulator
        AdsrParamBlock 0.0001,0.3,0.0,0.3
        else
        AdsrParamBlock 0.005,0.005,0.5,0.005
        endif
tune6   FiltTrivialHpParams 5000

; 4-pole version of the first instrument
Instrument_Bass4:
        dc Bass4Init-ChAlloc_InitInstruState
        dc OscDpwsawEval-ChEval_OscEvalBranch
        dc Bass4Filt-ChEval_FiltEvalBranch
        if !simulator
        AdsrParamBlock 0.1,0.1,0.5,0.1
        else
        AdsrParamBlock 0.005,0.005,0.5,0.005
        endif
filt4p  Filt4LP4Coefs
tune7   Filt4CoefResComp 500.0*2*PI/RATE,0.5,0.5

; 4-pole version of the hihat
Instrument_Noise4:
        dc Noise4Init-ChAlloc_InitInstruState
        dc NoiseEval-ChEval_OscEvalBranch
```

```
        dc Noise4Filt-ChEval_FiltEvalBranch
        if !simulator
        AdsrParamBlock 0.0001,0.3,0.0,0.3
        else
        AdsrParamBlock 0.005,0.005,0.5,0.005
        endif
        Filt4HP4Coefs
tune8   Filt4CoefResComp 5000.0*2*PI/RATE,0,0

; pointer lookup table for indexing the instrument structures
AllInstruments:
        dc Instrument_Bass
        dc Instrument_BassSinLfo
        dc Instrument_BassAdsrLfo
        dc Instrument_PulseBass
        dc Instrument_Noise
        dc Instrument_Bass4
        dc Instrument_Noise4

; addresses of tunable parameters
; these shall come with an accompanying manual with number mappings (
    see pdf)
InstruTunables:
        dc tune1        ; 0: 1st instru adsr A
        dc tune1+1      ; 1: 1st instru adsr D
        dc tune1+3      ; 2: 1st instru adsr R
        dc tune2        ; 3: 1st instru filt cutoff
        dc tune3        ; 4: 2nd instru filt base
        dc tune31       ; 5: 2nd instru filt sin freq
        dc tune4        ; 6: 3rd instru filt adsr A
        dc tune4+1      ; 7: 3rd instru filt adsr D
        dc tune4+3      ; 8: 3rd instru filt adsr R
        dc tune5        ; 9: 4th instru dutycycle base
        dc tune5+1      ; a: 4th instru dutycycle amplitude
        dc tune6        ; b: 5th instru filt cutoff
        dc tune7        ; c: 6th instru filt cutoff
        dc tune7+1      ; d: 6th instru filt resonance
        dc tune8        ; e: 7th instru filt cutoff


; CALLING CONVENTION
; Init:
;       args:
;               X:r1: channel workspace pointer
;               Y:r4: instrument
;               n2: note number
; Osc and filt:
;       as with plain oscillators, and then some
;       args:
;               X:r0: state pointer
;               X:r1: channel pointer
;               Y:r4: instrument pointer
        ; input and output: A
```

Listing 12: code/dpw˙coefs.asm

```
DpwCoefs: ; rate / (4 * freq * (1 - freq / rate)), midi notes 0..127
        dc      1467.996513/2048
        dc      1385.618236/2048
        dc      1307.863497/2048
        dc      1234.472796/2048
        dc      1165.201199/2048
        dc      1099.817519/2048
```

```
dc      1038.103542/2048
dc      979.853306/2048
dc      924.872405/2048
dc      872.977344/2048
dc      823.994931/2048
dc      777.761689/2048
dc      734.123320/2048
dc      692.934186/2048
dc      654.056820/2048
dc      617.361474/2048
dc      582.725680/2048
dc      550.033845/2048
dc      519.176862/2048
dc      490.051749/2048
dc      462.561304/2048
dc      436.613780/2048
dc      412.122579/2048
dc      389.005965/2048
dc      367.186788/2048
dc      346.592228/2048
dc      327.153554/2048
dc      308.805889/2048
dc      291.488001/2048
dc      275.142093/2048
dc      259.713612/2048
dc      245.151066/2048
dc      231.405855/2048
dc      218.432105/2048
dc      206.186518/2048
dc      194.628224/2048
dc      183.718650/2048
dc      173.421385/2048
dc      163.702064/2048
dc      154.528249/2048
dc      145.869323/2048
dc      137.696388/2048
dc      129.982168/2048
dc      122.700917/2048
dc      115.828334/2048
dc      109.341483/2048
dc      103.218715/2048
dc      97.439596/2048
dc      91.984838/2048
dc      86.836236/2048
dc      81.976608/2048
dc      77.389734/2048
dc      73.060308/2048
dc      68.973879/2048
dc      65.116810/2048
dc      61.476228/2048
dc      58.039982/2048
dc      54.796606/2048
dc      51.735273/2048
dc      48.845768/2048
dc      46.118447/2048
dc      43.544208/2048
dc      41.114459/2048
dc      38.821092/2048
dc      36.656452/2048
dc      34.613315/2048
dc      32.684863/2048
dc      30.864660/2048
```

```
        dc      29.146630/2048
        dc      27.525040/2048
        dc      25.994478/2048
        dc      24.549837/2048
        dc      23.186294/2048
        dc      21.899299/2048
        dc      20.684557/2048
        dc      19.538014/2048
        dc      18.455843/2048
        dc      17.434433/2048
        dc      16.470375/2048
        dc      15.560452/2048
        dc      14.701626/2048
        dc      13.891033/2048
        dc      13.125965/2048
        dc      12.403872/2048
        dc      11.722341/2048
        dc      11.079100/2048
        dc      10.472002/2048
        dc      9.899020/2048
        dc      9.358242/2048
        dc      8.847864/2048
        dc      8.366184/2048
        dc      7.911593/2048
        dc      7.482574/2048
        dc      7.077697/2048
        dc      6.695611/2048
        dc      6.335040/2048
        dc      5.994783/2048
        dc      5.673703/2048
        dc      5.370731/2048
        dc      5.084855/2048
        dc      4.815123/2048
        dc      4.560636/2048
        dc      4.320545/2048
        dc      4.094050/2048
        dc      3.880397/2048
        dc      3.678875/2048
        dc      3.488813/2048
        dc      3.309579/2048
        dc      3.140577/2048
        dc      2.981247/2048
        dc      2.831060/2048
        dc      2.689519/2048
        dc      2.556155/2048
        dc      2.430529/2048
        dc      2.312228/2048
        dc      2.200864/2048
        dc      2.096073/2048
        dc      1.997514/2048
        dc      1.904871/2048
        dc      1.817845/2048
        dc      1.736162/2048
        dc      1.659567/2048
        dc      1.587825/2048
        dc      1.520720/2048
        dc      1.458058/2048
        dc      1.399665/2048
        dc      1.345386/2048
        dc      1.295090/2048
```

Listing 13: code/saw˙ticks.asm

```
; freq / (rate / 2), midi notes 0..127
; freq = 2^((midinote-69)/12) * 440
SawTicks:
        dupf note,0,127
        dc (@pow(2,(note-69)/12.0)*440/(RATE/2))
        endm
```

Listing 14: code/instrucode.asm

```
; These functions mostly set up the parameters for their internal
    oscillators or filters,
; should be pretty straightforward
; Sometimes an instrument contains a lot of state and the initial state
; pointer (r4) from the main routine won't be enough to index
    everything
; Calling convention in instruparams.asm

BassInit:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruBassIdx_Lp),r5 ; r5 specified in
            FiltTrivialLpInit; same pattern repeats in this file
        bsr FiltTrivialLpInit

        lua (r1+ChDataIdx_OscState),r0
        move n2,r4
        bsr OscDpwsawInit

        rts

BassFilt:
        ; LFO-like effect: simply replace the coefficient with probably
             newly updated value from the panel
        ; could use Instrument_Bass etc. in all of these instead of r4,
             as we know what instrument we're dealing with
        ; but let's be nice and generic anyway
        move Y:(r4+InstruBassIdx_Lp+FiltTrivialLpParamsIdx_Coef),x1
        move x1,X:(r0+FiltTrivialLpStateIdx_Coef)
        bra FiltTrivialLpEval

Noise4Init:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruParamIdx_End),r5
        bsr Filt4Init

        lua (r1+ChDataIdx_OscState),r0
        bsr NoiseInit

        rts

Noise4Filt:
        move Y:(r4+InstruParamIdx_End+Filt4ParamsIdx_Coef),x1
        move x1,X:(r0+Filt4StateIdx_Coef)
        lua (r4+InstruParamIdx_End),r5
        bsr Filt4Eval
        asl #4,a,a ; hp coefs attenuate a * 1/8
        rts

Bass4Init:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruParamIdx_End),r5
        bsr Filt4Init
```

```
        lua (r1+ChDataIdx_OscState),r0
        move n2,r4
        bsr OscDpwsawInit

        rts

Bass4Filt:
        move Y:(r4+InstruParamIdx_End+Filt4ParamsIdx_Coef),x1
        move x1,X:(r0+Filt4StateIdx_Coef)
        lua (r4+InstruParamIdx_End),r5
        bra Filt4Eval

; indices inside the filter state
BassLfoStateIdx_LpFilt equ 0
BassLfoStateIdx_Lfo    equ FiltTrivialLpStateSize

BassSinLfoInit:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruBassIdx_Lp),r5
        bsr FiltTrivialLpInit

        lua (r1+ChDataIdx_FiltState+BassLfoStateIdx_Lfo),r0

        move Y:tune31,y0
        mpy #(30.0*SinTableSize/RATE),y0,a
        move a,x0
        bsr LFOSinInitState

        lua (r1+ChDataIdx_OscState),r0
        move n2,r4
        bsr OscDpwsawInit

        rts

; Remap original coef to filter with some lfo value in x1
; replace the state coefficient with a taylor approximated one
DoLfoLp macro
        ; TODO(?): c(f) ~= c(a) + c'(a) * (f - a) + c''(a)/2 * (f - a)
            ^2
        ; currently: c(f) ~= c(a) + c'(a) * (f - a)

        ; c(f) ~= c(a) + c'(a) * (f - a)
        ;       = c(a) + c'(a) * m * lfo [m = amplitude]
        ;       = c(a) + 2048*c'(a) * m/2048 * lfo
        ;       = K1   + K2         * K3     * lfo
        ; K2 and K3 combined into lp param lfo.
        move Y:(r4+InstruBassIdx_Lp+FiltTrivialLpParamsIdx_Lfo),x0 ; K2
            *K3
        move Y:(r4+InstruBassIdx_Lp+FiltTrivialLpParamsIdx_Coef),b ; K1
             = c(a)
        mac x0,x1,b
        move b,X:(r0+FiltTrivialLpStateIdx_Coef)
        endm

BassSinLfoFilt:
        ; use the sine as an LFO:
        lua (r0+BassLfoStateIdx_Lfo),r2
        bsr LFOSinEval
        DoLfoLp
        bra FiltTrivialLpEval
```

```
; indices inside the filter state
BassAdsrStateIdx_LpFilt equ 0
BassAdsrStateIdx_Adsr   equ FiltTrivialLpStateSize

BassAdsrLfoInit:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruBassIdx_Lp),r5
        bsr FiltTrivialLpInit

        lua (r1+ChDataIdx_FiltState+BassAdsrStateIdx_Adsr),r0
        bsr AdsrInitState

        lua (r1+ChDataIdx_OscState),r0
        move n2,r4
        bsr OscDpwsawInit

        rts

BassAdsrLfoFilt:
        bsr FiltTrivialLpEval
        ; use the ADSR as an LFO:
        ; the ADSR needs the A register, and also r0 and r4 are swapped
        ; "push" and "pop" the state to registers temporarily
        move a,r6
        move r4,n4
        lua (r0+BassAdsrStateIdx_Adsr),r2
        move r0,n0

        lua (r4+InstruBassAdsrIdx_FiltAdsr),r0
        move r2,r4
        move #>0,r2 ; don't kill the note
        bsr AdsrEval

        move r6,a
        move n0,r0
        move n4,r4
        move r3,x1

        DoLfoLp
        rts

; indices inside the oscillator state
PulseBassStateIdx_Adsr   equ PlsDpwSize

PulseBassInit:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruBassIdx_Lp),r5
        bsr FiltTrivialLpInit

        lua (r1+ChDataIdx_OscState+PulseBassStateIdx_Adsr),r0
        bsr AdsrInitState

        lua (r1+ChDataIdx_OscState),r0
        move n2,r4
        move Y:(Instrument_PulseBass+InstruPulseBassIdx_DutyBase),x1
        bsr PlsDpwInit

        rts

PulseBassOsc:
        ; use the ADSR as an LFO: tune the duty cycle
        ; r0 and r4 are swapped for adsr
```

```
        ; "push" and "pop" the state to registers temporarily
        move r4,n4
        lua (r0+PulseBassStateIdx_Adsr),r2
        move r0,n0

        lua (r4+InstruPulseBassIdx_FiltAdsr),r0
        move r2,r4
        move #>0,r2 ; don't kill the note
        bsr AdsrEval
        SimulatorMove r3,OutputHax


        move n4,r4
        move r3,x1
        move Y:(r4+InstruPulseBassIdx_DutyBase),a
        move Y:(r4+InstruPulseBassIdx_DutyAmpl),x0
        mac x0,x1,a
        move n0,r0
        move a,X:(r1+ChDataIdx_OscState+PlsDpwIdx_Duty)

        bra OscDpwplsEval

PulseBassFilt:
        rts

NoiseInstInit:
        lua (r1+ChDataIdx_FiltState),r0
        lua (r4+InstruNoiseIdx_Hp),r5
        bsr FiltTrivialHpInit

        lua (r1+ChDataIdx_OscState),r0
        bra NoiseInit

NoiseInstFilt:
        move Y:(r4+InstruNoiseIdx_Hp+FiltTrivialHpParamsIdx_Coef),x1
        move x1,X:(r0+FiltTrivialHpStateIdx_Coef)
        bra FiltTrivialHpEval
```

Listing 15: code/adsr.asm

```
; ADSR envelope (attack-decay-sustain-release)
; ============================================
; A: rise up to 1 in a specified time, using a lowpass constant
; D: fall (after infinite time) to sustain level, using a lowpass
    constant
; S: just a volume level, not a separate state (D handles this too)
; R: fall to 0 in a specified time, using a lowpass constant
;
; * A single instrument contains parameters for its ADSR
; * Each channel then contains a single ADSR state
; * The channels also contain a pointer/index/something to instrument
    table
;   to be able to reference the parameters
;
; Parameters
; ----------
; A: precalculated magic coefficient constant (see below)
; D: magic constant
; S: sustain volume level
; R: magic constant
; Params are kind of constant, this dsp code never changes them
; They may be tuned from the rtems side from the control panel or pc
    terminal
```

```
;
; magic constants: g = 1 - exp(-1 / (T * fs))
; T = time to reach ~63% for decaying, A and R are hacked to reach 1
    and 0
;
; State
; -----
; mode: attack/decay/release/killed
; value: previous value, because we're computing lowpass

; struct indices
AdsrParamIdx_A  equ     0
AdsrParamIdx_D  equ     1
AdsrParamIdx_S  equ     2
AdsrParamIdx_R  equ     3

; struct indices
AdsrStateIdx_Mode equ   0 ; a/d/r
AdsrStateIdx_Val  equ   1 ; previous value
AdsrStateIdx_Tgt  equ   2 ; release target value


; NOTE: these are bit numbers, so that we don't need to compare with
    accumulators
ADSR_MODE_ATTACK_BIT    equ 0
ADSR_MODE_DECAY_BIT     equ 1
ADSR_MODE_RELEASE_BIT   equ 2
ADSR_MODE_KILLED_BIT    equ 3

ADSR_MODE_ATTACK        equ (1<<ADSR_MODE_ATTACK_BIT)
ADSR_MODE_DECAY         equ (1<<ADSR_MODE_DECAY_BIT)
ADSR_MODE_RELEASE       equ (1<<ADSR_MODE_RELEASE_BIT)
ADSR_MODE_KILLED        equ (1<<ADSR_MODE_KILLED_BIT)

; Initialize ASDR state
; Input:
;       X:(r0): state pointer
; Work registers:
;       r2
AdsrInitState:
        move    #>ADSR_MODE_ATTACK,r2
        move    r2,X:(r0+AdsrStateIdx_Mode)
        move    #>0,r2
        move    r2,X:(r0+AdsrStateIdx_Val)
        rts

; lowpassing decayer with stuff divided by 2
; a: value
; b: target (already divided by 2)
; r4: X state struct pointer
; x0: lp coefficient
AdsrLpCareful macro
        asr #1,a,a      ; value /= 2
        sub a,b         ; tgt - value
        nop             ; stall :-(
        move b,x1       ; move to temp to be able to MAC
        mac x0,x1,a     ; a = 0.5*value + coeff * (0.5*tgt - 0.5*value)
        asl #1,a,a      ; multiply back by 2
        nop             ; stall :--(
        move a,r3       ; outval = a
        move a,X:(r4+AdsrStateIdx_Val) ; can I combine these?
        endm
```

```
; Evaluate the ASDR
; Input:
;       Y:(r0): param pointer
;       X:(r4): state pointer
;       r2: note number with key off bit included
; Output:
;       r3: envelope value, or -1 if killed
; Work registers:
;       r3, a, b
AdsrEval:
        move X:(r4+AdsrStateIdx_Mode),r3
        brset #ChNoteKeyoffBit,r2,_gateoff
_gateon:
        brset #ADSR_MODE_ATTACK_BIT,r3,_attack
        brset #ADSR_MODE_DECAY_BIT,r3,_decay
        bra _gotresult
_attack:
        ; NOTE: everything divided by 2 so that we can actually reach 1
        ; exponentially decaying things never actually reach the target
            ,
        ; only 63% of it in the time constant, so we trick it by
        ; specifying a different target, which might be >1
        ; also, when decaying, the target could be 1 + -1/0.63 = -0,59,
        ; and then "target - value" would overflow.
        ; value += coef * (target - value) [ideally]
        ; value = 2 * (value/2 + coef * (target/2 - value/2)) [here]
        ;                                 ^^^^^^^^ precalc'd constant
        ; same thing in release state

        move X:(r4+AdsrStateIdx_Val),a
        move #>(TGTCOEF/2),b
        move Y:(r0+AdsrParamIdx_A),x0
        AdsrLpCareful
        brclr #23,a1,_gotresult ; didn't overflow yet
_gotodecay:
        move #>ADSR_MODE_DECAY,r3
        move r3,X:(r4+AdsrStateIdx_Mode)
        ; when clipped, we should already be decaying (should we
            interpolate somehow?)
_decay:
        move X:(r4+AdsrStateIdx_Val),a
        move Y:(r0+AdsrParamIdx_S),b
        move Y:(r0+AdsrParamIdx_D),x0
        sub a,b         ; b = sustlevel - val
        nop             ; stall :-(
        move b,x1       ; b to temp
        mac x0,x1,a     ; value += coeff * (sust - value)
        nop             ; stall :--(
        move a,r3       ; outval = a
        move a,X:(r4+AdsrStateIdx_Val) ; see above
        bra _gotresult
_gateoff:
        brset #ADSR_MODE_RELEASE_BIT,r3,_relinited
        brset #ADSR_MODE_KILLED_BIT,r3,_gotresult ; NOTE: can this be
            ever called if the note is killed?
_relinit: ; start release state from whatever state we are in (a/d)
        ; compute release target:
        ;   current + (0 - current) * targetcoef
        ; = (1 - targetcoef) * current
        move X:(r4+AdsrStateIdx_Val),x0
        mpyi #((1-TGTCOEF)/2),x0,a ; NOTE: /2
        move #>ADSR_MODE_RELEASE,r3
```

```
        move a,X:(r4+AdsrStateIdx_Tgt)
        move r3,X:(r4+AdsrStateIdx_Mode)
_relinited:
        ; this divide by 2 hax again because we might
        ; roll from 1 to -0.58 which again does not fit in a register
        ; copypasta from attack stage
        move X:(r4+AdsrStateIdx_Val),a
        move X:(r4+AdsrStateIdx_Tgt),b
        move Y:(r0+AdsrParamIdx_R),x0
        AdsrLpCareful
        cmp #0.0,a
        bgt _gotresult
_gotokilled:
        move #>ADSR_MODE_KILLED,x0
        move x0,X:(r4+AdsrStateIdx_Mode)
        move #>0,x0
        move x0,X:(r4+AdsrStateIdx_Val)
        move #>-1.0,r3 ; kill signal
_gotresult:
        rts
```

Listing 16: code/osc.asm

```
; == OSCILLATORS ==
; - trivial saw wave,
; - dpw corrected saw wave,
; - trivial pulse wave (difference of two saws),
; - dpw'd pulse wave (difference of two dpw saws)
; - noise


; INITIALIZATION ROUTINES

; args: workspace at X:(r0), note number at r4
; work regs: x1
OscTrivialsawInit:
        move Y:(r4+SawTicks),x1
        move x1,X:(r0+SawOscIdx_Tick)
        move #>-1.0,x1
        move x1,X:(r0+SawOscIdx_Val) ; counter (-1..1)
        rts

; args: workspace at X:(r0), note number at r4
; work regs: x1
OscDpwsawInit:
        bsr OscTrivialsawInit ; trivial saw on top of this
        move #>1.0,x1
        move x1,X:(r0+DpwOscIdx_Val)
        move Y:(r4+DpwCoefs),x1
        move x1,X:(r0+DpwOscIdx_Coef) ; c coefficient, shifted by 11 (
            max amount 1500, for freq 8Hz)
        rts

; args: workspace at X:(r0), note number at r4, duty cycle (0=0%,
    1=50%) at x1
; work regs: x1, a, r0, x0
; could use triangles in range [0,1) instead of [-1,1)
; would be easier to scale this thing then
; NOTE: high value is at duty cycle, low at duty cycle - 1
; maybe sum it so that high is at 0.5 or at 1?
PlsTrivialInit:
        move x1,X:(r0+PlsOscIdx_Duty)
        move x1,x0
```

```
        ; saw0 is at the beginning
        bsr OscTrivialsawInit
        lea (r0+PlsOscIdx_Saw1),r0
        bsr OscTrivialsawInit
        move X:(r0+SawOscIdx_Val),a
        add x0,a
        move a,X:(r0+SawOscIdx_Val)
        rts

; args: workspace at X:(r0), note number at r4, duty cycle (0=0%,
    1=50%) at x1
; work regs: x1, a, r0, x0
PlsDpwInit:
        move x1,X:(r0+PlsDpwIdx_Duty)
        move x1,x0
        ; saw0 is at the beginning
        bsr OscDpwsawInit
        lea (r0+PlsDpwIdx_Saw1),r0
        bsr OscDpwsawInit
        move X:(r0+SawOscIdx_Val),a
        add x0,a
        move a,X:(r0+SawOscIdx_Val)
        rts

; args: workspace at X:(r0)
; work regs: x1
NoiseInit:
        ; seed = 1
        move #>1,x1
        move x1,X:(r0+NoiseOscIdx_Current)
        rts


; EVALUATION ROUTINES

; params: X:r0 = state pointer
; work regs: x0, a
; output in: a (value range [-1,1)
OscTrivialsawEval:
        move X:(r0+SawOscIdx_Val),a
        move X:(r0+SawOscIdx_Tick),x0
        add x0,a
        asl #8,a,a ; sneaky! 1+x -> -1+x if x >= 0
        asr #8,a,a ; (copy the highest bit and sign-extend back)
        move a,X:(r0+SawOscIdx_Val)
        rts

; params: X:r0 = state pointer
; work regs: x0, a
; output in: a (value range [-1,1)
OscDpwsawEval:
        bsr OscTrivialsawEval
        move a,x0
        mpy x0,x0,a     ; a = val ^ 2
        move X:(r0+DpwOscIdx_Val),x1
        move a,X:(r0+DpwOscIdx_Val)
        sub x1,a        ; dsq = val^2 - old^2
        move X:(r0+DpwOscIdx_Coef),x1
        move a,x0
        mpy x0,x1,a     ; out = c * dsq
        asl #11,a,a     ; fixpt coef
        rts
```

```
; params: X:r0 = state pointer
; work regs: x0, a, b
; output in: a (see value range docs above in init)
; NOTE: this is not really used anymore
OscTrivialplsEval:
        bsr OscTrivialsawEval
        move a,b
        lea (r0+PlsOscIdx_Saw1),r0
        bsr OscTrivialsawEval
        move b,x0
        sub x0,a        ; pulse = saw difference
        cmp #>-1.0,a
        blt _ovf        ; 0.6-0.1=0.5 ok, -0.9-0.6=-1.5 notok
        rts
_ovf:   add #>1.0,a     ; fix <-1 condition
        rts

; params: X:r0 = state pointer
; work regs: x0, a, b
; output in: a (see value range docs above in init)
; NOTE: ugly copypasta from above, PlsOsc -> PlsDpw, OscTrivial ->
    Oscdpw
OscDpwplsEval:
        move X:(r0+PlsDpwIdx_Saw0+DpwOscIdx_Saw+SawOscIdx_Val),a
        move X:(r0+PlsDpwIdx_Duty),y0
        add y0,a
        asl #8,a,a ; sneaky! 1+x -> -1+x if x >= 0
        asr #8,a,a ; (copy the highest bit and sign-extend back)
        move a,x0
        move a,X:(r0+PlsDpwIdx_Saw1+DpwOscIdx_Saw+SawOscIdx_Val)
        mpy x0,x0,a
        move a,X:(r0+PlsDpwIdx_Saw1+DpwOscIdx_Val)
        ; FIXME: saw 1 does not need bsr osctrivialsaweval?

        bsr OscDpwsawEval
        asr #1,a,b      ; /2

        lea (r0+PlsDpwIdx_Saw1),r0
        bsr OscDpwsawEval

        asr a           ; /2
        sub b,a         ; pulse = saw difference
        add #0.5,a      ; originally -1+duty..duty, shift to
        mac #-0.5,y0,a  ; between -1..1 (but half, dpw inaccuracies)
        ;asl a
        rts

; params: workspace at X:(r0)
; work regs: a, x0
; output in: a (see value range docs above in init)
NoiseEval:
        ; 24-bit xorshift, period length 2**24-1
        ; pseudocode (v is 24-bit):
        ;   v = previous value (or seed)
        ;   v ^= v<<8
        ;   v ^= v>>1
        ;   v ^= v<<11
        ;   new previous value = v
        ;   return v

        move X:(r0+NoiseOscIdx_Current),a1
```

```
        move a1,x0
        lsl #8,a
        eor x0,a

        move a1,x0
        lsr a
        eor x0,a

        move a1,x0
        lsl #11,a
        eor x0,a

        move a1,X:(r0+NoiseOscIdx_Current)

        ; sign-extend to a2
        move a1,x0
        move x0,a

        rts
```

Listing 17: code/filt.asm

```
FiltTrivialLpParamsIdx_Coef     equ     0
FiltTrivialLpParamsIdx_Lfo      equ     1

FiltTrivialLpStateIdx_Val       equ     0
FiltTrivialLpStateIdx_Coef      equ     1
FiltTrivialLpState_Size         equ     2

; args: workspace at X:(r0), params at Y:(r5)
; work regs: x1
FiltTrivialLpInit:
        move Y:(r5+FiltTrivialLpParamsIdx_Coef),x1
        move x1,X:(r0+FiltTrivialLpStateIdx_Coef)
        move #>0,x1
        move x1,X:(r0+FiltTrivialLpStateIdx_Val)
        rts

; args: workspace at X:(r0), input at a
; output: a
; work regs: a, b, x0, x1
FiltTrivialLpEval:
        move X:(r0+FiltTrivialLpStateIdx_Val),b
        move X:(r0+FiltTrivialLpStateIdx_Coef),x0
        asr #1,b,b      ; value /= 2
        asr #1,a,a      ; target /= 2
        sub b,a         ; a = 0.5*(tgt - value)
        nop             ; stall :(
        move a,x1       ; temp for mac
        mac x0,x1,b     ; b = 0.5*value + coeff * (0.5*val - 0.5*value)
        asl #1,b,a      ; shift back to output
        nop             ; stall
        move a,X:(r0+FiltTrivialLpStateIdx_Val)
        rts

        move a,b
        move X:(r0+FiltTrivialLpStateIdx_Val),x0 ; a = previous, a =
            current
        sub x0,b        ; b = (inp - x)
        move X:(r0+FiltTrivialLpStateIdx_Coef),x0
        move b,x1
        mac x0,x1,a     ;a = x + c * (inp - x)
```

```
        nop             ; stall :(
        move a,X:(r0+FiltTrivialLpStateIdx_Val)
        rts

FiltTrivialHpParamsIdx_Coef     equ     0

FiltTrivialHpStateIdx_Prevdiff2 equ     0
FiltTrivialHpStateIdx_Coef      equ     1
FiltTrivialHpState_Size         equ     2

; args: workspace at X:(r0), params at Y:(r5)
; work regs: x1
FiltTrivialHpInit:
        move Y:(r5+FiltTrivialHpParamsIdx_Coef),x1
        move x1,X:(r0+FiltTrivialHpStateIdx_Coef)
        move #>0,x1 ; just assume something. will this work or give
            nasty transients?
        move x1,X:(r0+FiltTrivialHpStateIdx_Prevdiff2)
        rts

; args: workspace at X:(r0), input at a
; output: a
; work regs: a, b, x0, x1

; y1 = g * (y0 + x1 - x0)
;    = g * y0 + g * (x1 - x0)
;    = g * (x1 + (y0 - x0))
;    = 2 * g * (x1/2 + (y0 - x0) / 2)
; store: (y0-x0)/2
FiltTrivialHpEval:
        move X:(r0+FiltTrivialHpStateIdx_Prevdiff2),b ; b = (y0-x0) / 2
        asr a ; x1 /= 2
        add a,b ; b = (x1/2 + (y0-x0)/2)
        move X:(r0+FiltTrivialHpStateIdx_Coef),x0
        move b,x1
        mpy x0,x1,b ; b = g * (x1 / 2 + (y0-x0) / 2) = y1 / 2
        sub b,a ; a = x1 / 2 - y1 / 2 = (x1 - y1) / 2
        neg a ; a = (y1 - x1) / 2
        move a,X:(r0+FiltTrivialHpStateIdx_Prevdiff2) ; b = new (y0-x0)
             / 2
        asl #1,b,a ; output
        rts
```

Listing 18: code/multipole.asm

```
; workspace: X:(r0), params Y:(r5)
Filt4Init:
        move #>0,x0
        move x0,X:(r0+Filt4StateIdx_Part0+Filt4PartStateIdx_x0)
        move x0,X:(r0+Filt4StateIdx_Part0+Filt4PartStateIdx_y0)
        move x0,X:(r0+Filt4StateIdx_Part1+Filt4PartStateIdx_x0)
        move x0,X:(r0+Filt4StateIdx_Part1+Filt4PartStateIdx_y0)
        move x0,X:(r0+Filt4StateIdx_Part2+Filt4PartStateIdx_x0)
        move x0,X:(r0+Filt4StateIdx_Part2+Filt4PartStateIdx_y0)
        move x0,X:(r0+Filt4StateIdx_Part3+Filt4PartStateIdx_x0)
        move x0,X:(r0+Filt4StateIdx_Part3+Filt4PartStateIdx_y0)
        move x0,X:(r0+Filt4StateIdx_Mem)
        move Y:(r5+Filt4ParamsIdx_Coef),x0
        bsr Filt4SetCoef
        move Y:(r5+Filt4ParamsIdx_Gres),y0
        move y0,X:(r0+Filt4StateIdx_Gres)
        bsr Filt4SetRes
        rts
```

```
; workspace: X:(r0)
; input: w in x0
; work regs: b, x0, x1
; self.g = 0.9892 * w - 0.4342 * w**2 + 0.1381 * w**3 - 0.0202 * w**4
Filt4SetCoef:
                                ; x0 = w
        mpy #0.9892,x0,b        ; b = 0.9892 * w
        mpy x0,x0,a
        move a,x1               ; x1 = w^2
        mac #-0.4342,x1,b       ; b -= 0.4342 * w^2
        mpy x0,x1,a
        move a,x1               ; x1 = w^3
        mac #0.1381,x1,b        ; b += 0.1381 * w^3
        mpy x0,x1,a
        move a,x1               ; x1 = w^4
        mac -#0.0202,x1,b       ; b -= 0.0202 * w^4
        move b,X:(r0+Filt4StateIdx_Coef)
        rts

; input: w (0..1) in x0, c_res in y0, state in X:r0
; self.g_res = c_res * (1.0029 + 0.0526 * w - 0.0926 * w**2 + 0.0218 *
    w**3)
Filt4SetRes:
                                ; x0 = w
        mpy #0.0526,x0,b        ; b = 0.0526 * w
        mpy x0,x0,a
        move a,x1               ; x1 = w^2
        mac #-0.0926,x1,b       ; b -= 0.0926 * w^2
        mpy x0,x1,a
        move a,x1               ; x1 = w^3
        mac #0.0218,x1,b        ; b += 0.0218 * w^3
        mpy #1.0029/2,y0,a      ; a = 1.0029 * c_res
        move b,y1
        asl a                   ; a = c_res * 1.0029
        mpy y0,y1,b             ; b = c_res * b
        add a,b                 ; b = c_res * (1.0029 + f(w))
        move b,X:(r0+Filt4StateIdx_Gres)
        rts

; one lowpass part of the whole 4-pole system
; input: x1, output: y1
; q = (x1 + 0.3 * self.x0) / 1.3
; self.x0 = x1
; self.y0 += self.g * (q - self.y0) # y = g * a + (1 - g) * y
Filt4RunPart macro
        move X:(r2+Filt4PartStateIdx_x0),x0
        move x1,X:(r2+Filt4PartStateIdx_x0)
        mpy #0.3,x0,a           ; q = 0.3 * x0
        add x1,a                ; q = x1 + 0.3 * x0
        move a,x0
        mpy #(1.0/1.3),x0,a     ; q = (x1 + 0.3 * x0) / 1.3
        move X:(r2+Filt4PartStateIdx_y0),y0
        move X:(r0+Filt4StateIdx_Coef),y1
        sub y0,a                ; a = q - y0
        move a,x0
        mpy x0,y1,a             ; a = g * (q - y0)
        add y0,a                ; a = y0 + g * (q - y0)
        move a,y1
        move a,X:(r2+Filt4PartStateIdx_y0)
        endm
```

```
; workspace: X:(r0), params Y:(r5)
; input: a
; output: a
; work regs: several
Filt4Eval:
        ; x = x_in - 4 * g_res * (mem - g_comp * x_in)
        move Y:(r5+Filt4ParamsIdx_Gcomp),y1
        move X:(r0+Filt4StateIdx_Gres),y0
        move X:(r0+Filt4StateIdx_Mem),x0
        move a,x1
        mpy y1,x1,b     ; b = g_comp * x_in
        sub x0,b        ; b = g_comp * x_in - mem
        neg b           ; b = mem - g_comp * x_in
        move b,x0
        mpy x0,y0,b     ; b = g_res * (mem - g_comp * x_in)
        asl #2,b,b      ; b = 4 * g_res * (mem - g_comp * x_in)
        sub b,a         ; x = x_in - b
        move a,x1       ; x1 = x

        move Y:(r5+Filt4ParamsIdx_A),x0
        mpy x0,x1,b                     ; b = A * x1

        lea (r0+Filt4StateIdx_Part0),r2
filt41  Filt4RunPart
        move Y:(r5+Filt4ParamsIdx_B),x0
        mac x0,y1,b                     ; b = B * lp1 + A * x1

        lea (r0+Filt4StateIdx_Part1),r2
filt42  Filt4RunPart
        move Y:(r5+Filt4ParamsIdx_C),x0
        mac x0,y1,b                     ; b = C * lp2 + B * lp1 + A *
            x1

        lea (r0+Filt4StateIdx_Part2),r2
filt43  Filt4RunPart
        move Y:(r5+Filt4ParamsIdx_D),x0
        mac x0,y1,b

        lea (r0+Filt4StateIdx_Part3),r2
filt44  Filt4RunPart
        move Y:(r5+Filt4ParamsIdx_E),x0
        move b,X:(r0+Filt4StateIdx_Mem)
        mac x0,y1,b
filt4o  move b,a
        rts
```

Listing 19: code/sin.asm

```
; Sin approximator
; ================
; This sin approximator approximates the sin function by interpolating
    values
; in a precalculated table (in sin_table.asm). Like oscillators and
    filters, the
; sin approximator has a state. The state contains three 24-bit numbers
    :
; - M+SinTable where M is the index to the lookup table
; - f, a fixed-point number, fractional part for interpolation
; - c, a fixed-point number, a constant added to f on every evaluation
    step
; such that the current approximation is calculated with
;   (1-f)*rawSin(M) + f*rawSin(M+1)
```

```
; where rawSin(i) is the entry in the lookup table at index i %
    SinTableSize.
; At each step, f is increased by c = frequency*SinTableSize/RATE. If f
     then
; exceeds (or equals) 1.0, M is incremented by one and 1.0 is
    subtracted from
; f. Note that the frequency must be less than RATE/SinTableSize;
    otherwise c
; would exceed 1.0. With LFOs this shouldn't be a problem, since e.g.
    with
; SinTableSize=32 this frequency threshold is 1500 Hz.

LFOSinStateIdx_MPlusSinTable equ 0
LFOSinStateIdx_f equ 1
LFOSinStateIdx_c equ 2

; Initialize sin state
; Input:
;       X:(r0): state
;       x0: c (see above for explanation)
; Work registers:
;       x0
LFOSinInitState:
        move x0,X:(r0+LFOSinStateIdx_c)             ; state.c = c
        move #>SinTable,x0
        move x0,X:(r0+LFOSinStateIdx_MPlusSinTable) ; state.
            MPlusSinTable = SinTable, i.e. M = 0, i.e. start at the
            beginning
        move #>0,x0
        move x0,X:(r0+LFOSinStateIdx_f)             ; state.f = 0.0
        rts

; Compute next value of sin
; Input:
;       X:(r2): state
; Output:
;       x1: approximate sin value
; Work registers:
;       b, x0, y0, y1, r3, r6
LFOSinEval:
        ; compute result
        ; in the comments here, let's abbreviate SinTable by T and
            SinTableSize by N.

        move X:(r2+LFOSinStateIdx_MPlusSinTable),r6 ; r6 = &T[M]
        move #>(SinTableSize-1),m6

        move Y:(r6)+,x0                              ; x0 = T[M % N]
        move Y:(r6)-,b                               ; b = T[(M+1) %
             N]
        sub x0,b                      ; b  = T[(M+1) % N] - T[M % N]
        move X:(r2+LFOSinStateIdx_c),y1 ; y1 = c
        move b,y0                     ; y0 = T[(M+1) % N] - T[M % N]
        move x0,b                     ; b = T[M % N]
        move X:(r2+LFOSinStateIdx_f),x0 ; x0 = f
        mac x0,y0,b                   ; b  = T[M % N] + f*(T[(M+1) %
            N] - T[M % N])  (this is the interpolated result)

        lua (r6)+,r3                  ; r3 = &T[M+1]

        move b,x1 ; result
```

```
        ; advance the state

        move x0,b                       ; b = f
        add y1,b                        ; b = f+c
        move b,x0
        and #>$7fffff,b                 ; if f+c > 1.0, this wraps it
            back to f+c - 1.0
        move b,X:(r2+LFOSinStateIdx_f)
        cmp x0,b                        ; if wrapped around, this
            yields not-equal...
        tne r3,r6                       ; ...and r3, i.e. the address
            of the next table entry, goes to r6
        move r6,X:(r2+LFOSinStateIdx_MPlusSinTable)

        rts
```

Listing 20: code/isr.asm

```
;*******************************************
;INTERRUPT ROUTINES
;*******************************************

UpdateVolume:
        BRCLR   #HSR_HRDF,X:<<HSR,*     ; Make sure that data is
            available
        MOVEP   X:<<HRX,r7              ; Read the data to r7
        MOVE    r7,Y:MasterVolume       ; Write the data to memory
        MOVEP   r7,X:<<HTX              ; Write the read value back to
            the MCU
        RTI                             ; Return from interrupt

UpdateTunable:
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        move    Y:(r7+InstruTunables),r7
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,n7
        move    n7,Y:(r7)
        RTI

KeyEvent:
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        MOVEP   r7,X:<<HTX
        RTI
EncoderUp:
        ;Write your encoder up handler here

        move Y:PanelKeys_NoteOffset,r7
        lua (r7+12),r7
        move r7,Y:PanelKeys_NoteOffset

        RTI

EncoderDown:
        ;Write your encoder down handler here

        move Y:PanelKeys_NoteOffset,r7
        lua (r7-12),r7
        move r7,Y:PanelKeys_NoteOffset

        RTI
```

```
MidiKeyOn:
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        MOVE    r7,Y:NoteThatWentDown
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        MOVE    r7,Y:InstrumentThatWentDown
        ;BRCLR  #HSR_HRDF,X:<<HSR,*
        ;MOVEP  X:<<HRX,r7
        RTI

MidiKeyOff:
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        MOVE    r7,Y:NoteThatWentUp
        BRCLR   #HSR_HRDF,X:<<HSR,*
        MOVEP   X:<<HRX,r7
        MOVE    r7,Y:InstrumentThatWentUp
        RTI

Panic:
        bset #0,Y:PanicState
        rti
```

References

[1] Vesa Välimäki and Antti Huovilainen. Virtuaalista nostalgiaa – digitaalinen vähentävä äänisynteesi. Musiikki, 35(1–2):78–98, 2005.

[2] Vesa Välimäki and Antti Huovilainen. Oscillator and filter algorithms for virtual analog synthesis. Computer Music Journal, 30(2):19–31, Summer 2006.

[3] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.