Embedded Sequencer

Brandon Cruz, Adrian Florea & Alexander Ranschaert
Specifications/Desired behavior

16 slots

Select slot with button

2 Modes:
- Record
- Playback; choose BPM

4 Instruments; Controlled with MIDI keyboard
User Interface

- Memory mapped messages to userspace:

```
[track|step|playback|bpm]
```

8bits  8bits  1bit  15bits

```c
// writes x and y coordinates
static void read_props(user_interface_props_t *props)
{
    unsigned int bpm_playback = ioread16(UI_BPM_PLAYBACK(dev.virtbase));
    unsigned int step_track = ioread16(UI_STEP_TRACK(dev.virtbase));
    props->step = (unsigned char)step_track;
    props->track = (unsigned char) (step_track >> 0);
    props->bpm = (unsigned short)(bpm_playback & 0x0000FFFF);
    printk(KERN_INFO "Here: %d", props->bpm);
    props->playback = (unsigned char) ((bpm_playback & 0x00000000) >> 15);
    dev.props = *props;
}
```
Decoding USB-MIDI

Handled via USB Bulk Transfer:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cable Number</td>
<td>Code Index</td>
<td>MIDI_0</td>
<td>MIDI_1</td>
</tr>
<tr>
<td>Number</td>
<td>Number</td>
<td>Note on or off</td>
<td>Note velocity</td>
</tr>
</tbody>
</table>

USB-MIDI Packets

Zero padded

Libusb Device Handle:

```c
// Get MIDI Device
if ( inter->bInterfaceClass == 1 &&
     inter->bInterfaceProtocol == 0 &&
     inter->bInterfaceSubClass == 3 ) {

*endpoint_address = inter->endpoint[1].bEndpointAddress;

// Input: packet.keyCode[1]
NoteInfo mapCodeToNote(int num) {
    NoteInfo result;
    if(num != 0){
        int note[12] = {1,2,3,4,5,6,7,8,9,10,11,12};
        int index = (num - MIN_KEY_CODE) % 12;
        int octave = (num - MIN_KEY_CODE) / 12;
        int noteVal = note[index] + (octave * 12);

        result.noteVal = index;
        result.octave = octave;
        result.noteIndex = noteVal;
    } else{
        result.noteVal = 0;
        result.octave = 0;
        result.noteIndex = 0;
    }

    return result; }
```
Wolfson WM8731 Audio CODEC Config.

24b, 48kHz, MSB first

(driver we used was a wrapper around preconfigured Intel IP)
Driver Interface (Moore FSM)

1. Store sample from userspace to RAM
2. Shift Pitch & Pass samples to CODEC

\[ w \text{ & } cs \]

\[ w : \text{write} \]
\[ cs : \text{chipselect} \]
\[ \text{adv} : \text{advance} \]
\[ MD : \text{Memory Depth (48000)} \]
Driver Interface (Moore FSM, State Transitions)

```verilog
always_ff @(posedge clk)
if (reset) state <= RST;
else case(state)
    RST: begin // reset internal signals
        w_r_address <= 16'b0;
        fract_index <= 32'b0;
        control <= 32'b0;
        mem_we <= 1'b0;
        state <= WAIT_STORE;
    end
    WAIT_STORE: if (write && chipselect) begin
        mem_we <= 1'b1;
        state <= STORE;
    end else begin
        mem_we <= 1'b0;
        state <= WAIT_STORE;
    end
    STORE: begin
        if (w_r_address == MEM_DEPTH-1) begin
            w_r_address <= 16'b0; // reset for read
            fract_index <= 32'b0;
            mem_we <= 1'b0;
            state <= WAIT_READ; // sample stored, continue to read mode
        end else if (write) begin // wait for falling edge of write
            w_r_address <= w_r_address + 16'b1;
            state <= WAIT_STORE;
        end else begin
            mem_we <= 1'b0;
            state <= STORE;
        end
    end
    DONE_READ: begin // arrival of a new wav file
        if (write && chipselect) begin
            w_r_address <= 16'b0;
            fract_index <= 32'b0;
            mem_we <= 1'b1;
            state <= STORE;
        end else begin
            state <= DONE_READ;
        end
    end
endcase
end
```
Driver Interface (Moore FSM, Output Logic)

```verilog
// output logic
always_comb begin
  case(state)
  READ: begin
    if (!advance) begin
      fract_index_sum = fract_index + pitch_shift;
      leftSample = {mem_out, 8'b0};
      rightSample = {mem_out, 8'b0};
    end else begin
      leftSample = {mem_out, 8'b0};
      rightSample = {mem_out, 8'b0};
      fract_index_sum = 32'b0;
    end
  end
  WAIT_READ: begin
    leftSample = {mem_out, 8'b0};
    rightSample = {mem_out, 8'b0};
    fract_index_sum = 32'b0;
  end
  default: begin // make sure data is ready before advance signal arrives
    leftSample = 24'b0;
    rightSample = 24'b0;
    fract_index_sum = 32'b0;
  end
endcase
end
endmodule
```
Software Control

- Memory mapped messages:
  
  \[
  \text{[playbackmode|active channels|pitch_shift|note_velocity|channel|audio_sample]} \]
  
  1 bit  
  4 bits  
  4 bits  
  3 bits  
  2 bits  
  16 bits

- Audio driver with 4 device registers (1 per track) for memory mapped write

```c
/* Device registers */
#define REG_AUDIO1(x) ((x)+4)
#define REG_AUDIO2(x) ((x)+8)
#define REG_AUDIO3(x) ((x)+12)
#define REG_AUDIO4(x) ((x)+16)

int active_chan[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int control_notes[4][16] = {
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
};

// set file names:
char* sample_name[4][3] = {
    {"bassoon2.wav","bassoon3.wav","bassoon4.wav"},
    {"cello2.wav","cello2C.wav","cello2C4.wav"},
    {"piano2.wav","piano3.wav","piano4.wav"},
    {"synthbrass2.wav","synthbrass3.wav","synthbrass4.wav"}
};
```
Pitch Shifter

- Just intonation vs Equal Temperament
- Implemented using fixed point numbers and skipping samples

<table>
<thead>
<tr>
<th>Note/Interval</th>
<th>Just Intervals</th>
<th>CENTS</th>
<th>&quot;Pythagorean&quot; (True intervals)</th>
<th>CENTS</th>
<th>Equal Temperament</th>
<th>CENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tonic C</td>
<td>1</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Minor 2nd c#</td>
<td>16/15</td>
<td>111.73</td>
<td>256/243</td>
<td>90.22</td>
<td>2^{122}</td>
<td>100.00</td>
</tr>
<tr>
<td>Major 2nd D</td>
<td>10/9</td>
<td>182.40</td>
<td>9/8</td>
<td>203.91</td>
<td>2^{125}</td>
<td>200.00</td>
</tr>
<tr>
<td>Minor 3rd e#</td>
<td>6/5</td>
<td>315.64</td>
<td>32/27</td>
<td>294.13</td>
<td>2^{74}</td>
<td>300.00</td>
</tr>
<tr>
<td>Major 3rd E</td>
<td>5/4</td>
<td>386.31</td>
<td>81/64</td>
<td>407.82</td>
<td>2^{123}</td>
<td>400.00</td>
</tr>
<tr>
<td>Perfect 4th F</td>
<td>4/3</td>
<td>498.04</td>
<td>4/3</td>
<td>498.04</td>
<td>2^{71/2}</td>
<td>500.00</td>
</tr>
<tr>
<td>Augmented 4th f#</td>
<td>45/32</td>
<td>590.22</td>
<td>729/512</td>
<td>611.73</td>
<td>\sqrt{2}</td>
<td>600.00</td>
</tr>
<tr>
<td>Diminished 5th Gb</td>
<td>64/45</td>
<td>609.78</td>
<td>1024/729</td>
<td>588.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perfect 5th G</td>
<td>3/2</td>
<td>701.96</td>
<td>3/2</td>
<td>701.96</td>
<td>2^{122}</td>
<td>700.00</td>
</tr>
<tr>
<td>Minor 6th g#</td>
<td>8/5</td>
<td>813.69</td>
<td>128/81</td>
<td>792.18</td>
<td>2^{73}</td>
<td>800.00</td>
</tr>
<tr>
<td>Major 6th A</td>
<td>5/3</td>
<td>884.36</td>
<td>27/16</td>
<td>905.87</td>
<td>2^{74}</td>
<td>900.00</td>
</tr>
<tr>
<td>Minor 7th a#</td>
<td>9/5</td>
<td>1017.60</td>
<td>16/9</td>
<td>996.09</td>
<td>2^{76}</td>
<td>1000.00</td>
</tr>
<tr>
<td>Major 7th B</td>
<td>15/8</td>
<td>1088.27</td>
<td>243/128</td>
<td>1109.78</td>
<td>2^{11/2}</td>
<td>1100.00</td>
</tr>
<tr>
<td>Octave C'</td>
<td>2</td>
<td>1200.00</td>
<td>2</td>
<td>1200.00</td>
<td>2</td>
<td>1200.00</td>
</tr>
</tbody>
</table>

1.887 + 1.887 = 3.77 = 3
wav_handler.c

```
struct HEADER {
    unsigned char riff[4];          // RIFF string
    unsigned int overall_size;     // overall size of file in bytes
    unsigned char wave[4];         // WAVE string
    unsigned char fmt_chunk_marker[4]; // fmt string with trailing null char
    unsigned int length_of_fmt;    // length of the format data
    unsigned int format_type;      // format type. 1-PCM, 3- IEEE float, 6 - 8bit A-law, 7 - 8bit mu law
    unsigned int channels;         // no of channels
    unsigned int sample_rate;      // sampling rate (blocks per second)
    unsigned int byte_rate;        // SampleRate * NumChannels * BitsPerSample/8
    unsigned int block_align;      // NumChannels * BitsPerSample/8
    unsigned int bits_per_sample;  // bits per sample, 8- 8bits, 16-16 bits etc
    unsigned char data_chunk_header[4]; // DATA string or FLLA string
    unsigned int data_size;         // NumSamples * NumChannels * BitsPerSample/8 - size of the next chunk that will be read
};
```

Parses input .wav headers & data segment
Conclusions

- Successfully Implemented the desired behavior.
- Pitch-shifting in hardware.
- User Interface logic & file handling in software.
- We did not implement a mixer.