Project Report: Embedded Sequencer

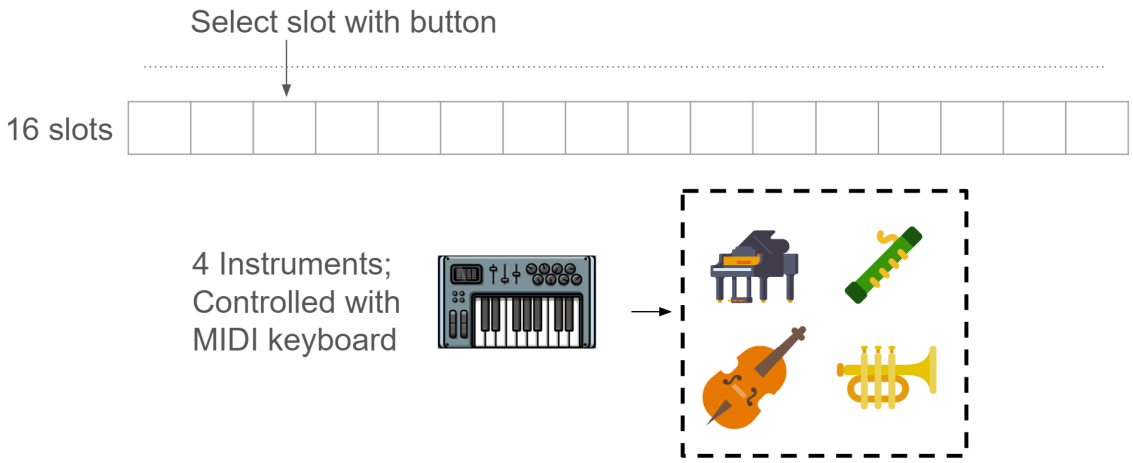
Adrian N Florea (anf2143)
Alexander Ranschaert (anr2157)
Brandon Vidal Cruz (bvc2106)

I. Introduction

A step-sequencer is an essential part of every basic modular synthesizer setup. A fixed pattern of a certain number of steps is repeated continuously. The musician can decide whether to enable or disable each step and which note to play at that step. This makes a step-sequencer very attractive in the context of beat-making and was therefore the starting point of early drum machines. The user can set the number of steps, tracks and the beats per minute (BPM). Opposed to a real-time sequencer, and therefore lends itself more easily to implementation as an embedded system.

Elektron Machinedrum SPS-1 (Wikipedia)

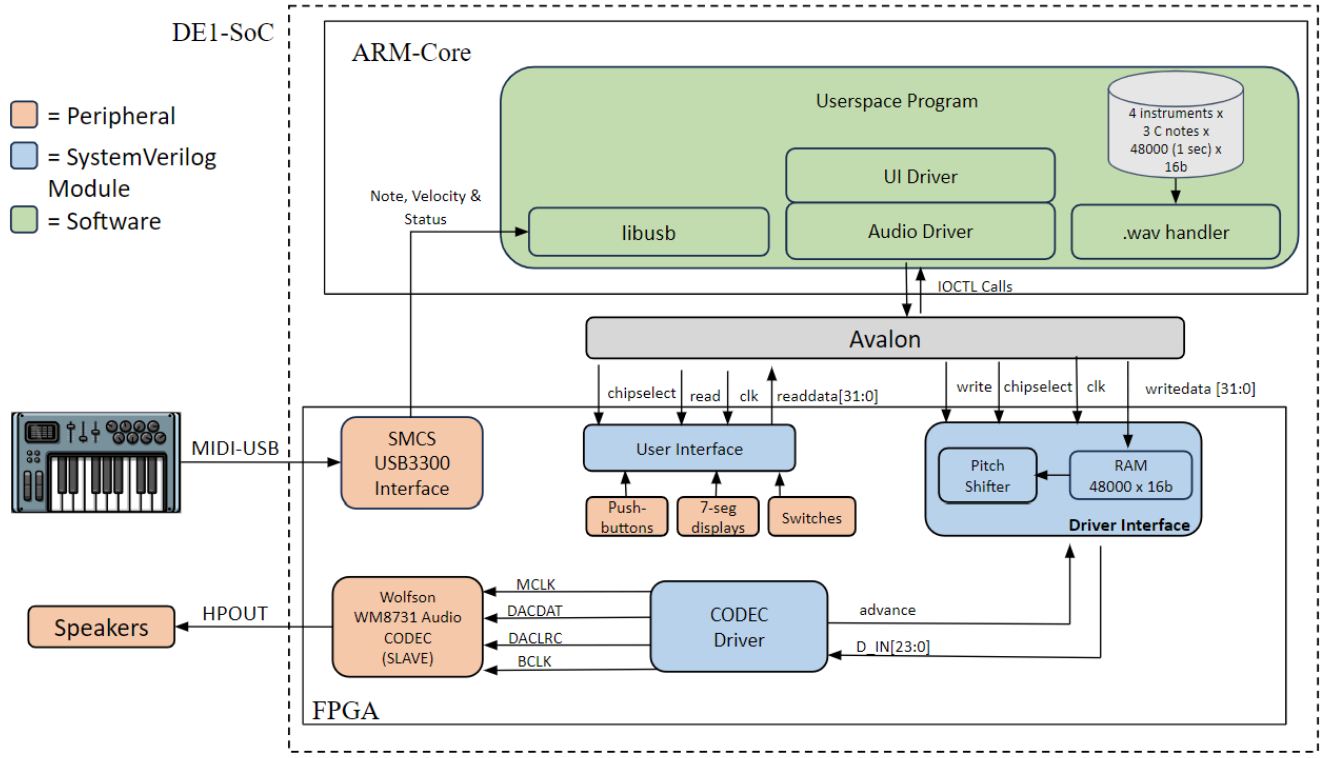
The goal in this project is to emulate such a step-based sequencer. Therefore we propose the behavior in the figure below. There are 16 tracks that can each be filled during the recording phase with a single sample of a note played by 1 of the four instruments: piano, bassoon, cello and brass. The notes are simply entered using a MIDI keyboard The instrument and slot are selected through buttons and displayed by the hexagonal segments. The sequence can then be played completely in the playback mode, where the BPM can be selected by the user, again through the buttons.



Proposed Behavior

II. System design

The diagram below gives a full overview of the architecture of the embedded sequencer. Each module will be discussed in more detail below.



Full-system block diagram

II.a: USB-MIDI

MIDI is a communication protocol established to communicate between MIDI devices such as most digital electronic instruments. A majority of MIDI messages consist of multi-byte packets beginning with a status byte followed by one or two data bytes. MIDI supports critical features for musical instruments such as keypresses, releases, velocity, and aftertouch.

Most computers do not directly support MIDI without an audio interface or USB-MIDI converter. Eventually, a MIDI specification was developed for USB that included the Audio class of devices [3]. USB-MIDI supports baud rates much the 31.2k baud rate of MIDI in order to handle many virtual cables worth of MIDI data [4]. We will use a baud rate of 115200 in our system.

Byte 0		Byte 1	Byte 2	Byte 3
Cable Number	Code Index Number	MIDI_0	MIDI_1	MIDI_2

USB-MIDI transfers data continuously using 32-it USB-MIDI event packets illustrated in the image above. The first byte is a packet header that contains a cable number and code index number while the next three bytes contain the actual MIDI event. The three USB-MIDI event packets virtually maintain the same information structure as the original MIDI events.The code index number (CIN) indicates the classification of the bytes in the MIDI_x_fields. The following table summarizes the three CIN codes we plan to decode in this project as well as the MIDI messages.

CIN	MIDI_x Size	Description
0x8	3	Note-off
0x9	3	Note-on
0xA	3	Poly-KeyPress

When a step is recorded for the sequencer, our software will decode USB-MIDI messages to record the note played during the recording window as well as the velocity. These decoded note values will be used to select an octave and pitch shift amount (counted by semitones above C) in our userspace software.

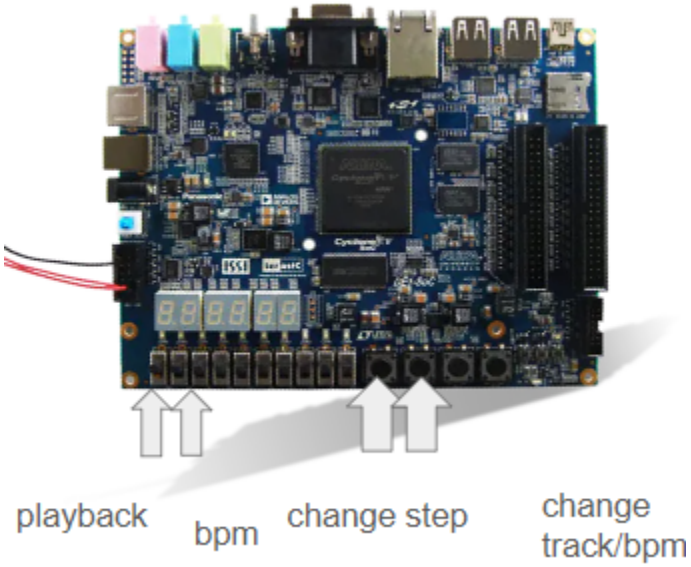
Status Byte	Data Byte 1	Data Byte 2	Message	Legend
1000nnnn	0kkkkkkk	0vvvvvvv	Note Off	n=channel* k=key # 0-127 (60=middle C) v=velocity (0-127)
1001nnnn	0kkkkkkk	0vvvvvvv	Note On	n=channel k=key # 0-127(60=middle C) v=velocity (0-127)
1010nnnn	0kkkkkkk	0ppppppp	Poly Key Pressure	n=channel k=key # 0-127(60=middle C) p=pressure (0-127)

The Arturia Red Analog Microbrute analog synthesizer is used in this lab since it can support USB-MIDI communication and has velocity and 5 octave control.

II.b: User Interface

We decided to emulate the tactile feel of an old school step sequencer by using the buttons and switches that were available on the De1-SOC as an interface for the user to change the step, track, and BPM. The leftmost switch determines if the sequencer is recording a note for a step

or playing back the recorded sequence. The second switch allows the user to change the BPM. The left set of buttons are used to adjust the step, and the right set of buttons change the track/BPM. These messages from the user interface were transferred to the user space program through a custom driver that read these values. This was a memory mapped avalon interface that sent the track, step, playback, and bpm information when this was changed by the user.




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

8bits   8bits 1bit   15bits
```

II.c: Userspace program

The userspace program handles 3 main tasks: decode MIDI samples, load the corresponding .wav file from the channel number provided by the user interface and pass these 16b samples on to the hardware in a 32b write operation, where the first 16b contain information about the pitch shift. Finally, it also contains the recorded sequence so that it can be played back to the hardware in a sample-by-sample fashion. Additionally information about the track, channels active a playback mode are also included in the 32b write operation but this is not ultimately utilized.

```
[playbackmode|active channels|pitch_shift|note_velocity|channel|audio_sample]

    1 bit          4 bits        4 bits      3 bits       2 bits     16 bits
```

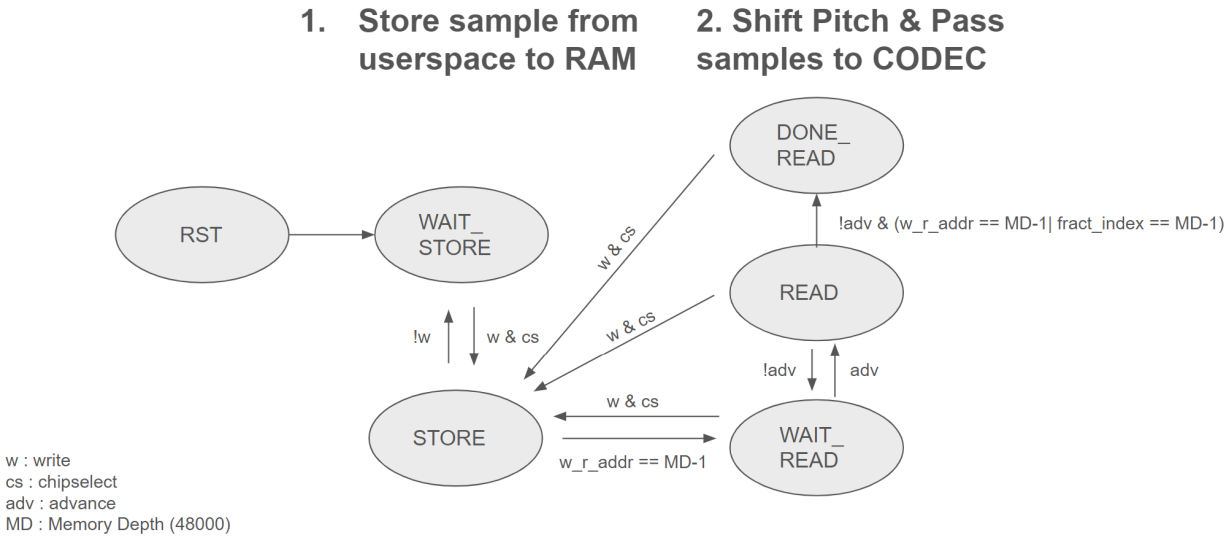
Wav files

The userspace has access to 3 samples of each of the 4 instruments (bassoon, piano, cello and brass). Each sample is a C-note in a different octave (i.e. C2, C3 and C4). We cut these samples to 1s and resampled them so that a single file contains 48,000 16b samples, so that the samples are compatible with the Codec. Note that the Codec takes 24b samples, but this is handled in the hardware by appending zeros to the 16b samples.

II. d: Driver Interface & Pitch Shifter

The driver interface operates as a simple state machine (see state diagram below) with 2 important phases: first it stores a full 1 second audio sample in an internal RAM block (consisting of 48000 16-bit words). It alternates between WAIT_STORE and STORE during this phase, incrementing the RAM address each time a write operation is performed. The completion of a single 32-bit write cycle is detected by asserting the rising & falling edge of the Avalon write signal. In the STORE state, we also check the current RAM write address. If this reaches the depth of the RAM block, we know that a full audio file has been transferred to the hardware. This indicates the start of the 2nd phase of the state diagram, where our module will react to *advance* signals from the codec interface, approximately 48000 per second. Again, we simply alternate between an output state, READ, and a wait state WAIT_READ, where the address RAM address is again incremented by 1 each time a sample is read by the codec interface. When the end of the sample is read (either because of the pitch shift operation or simply because the address reaches the memory depth), we transition to the DONE_READ state, where we wait for a new audio sample from the software userspace. It should be noted that during every state of the second phase, we can transition back to the STORE state when a new sample is passed to the hardware by the userspace. This ensures that even when a note is playing, we can play a new note.

A concern we had was that the transfer from userspace to hardware would cause a noticeable delay during the recording phase when pressing a key, but it turned out that the software was sufficiently fast to handle this.



Driver Interface FSM State Transition Diagram

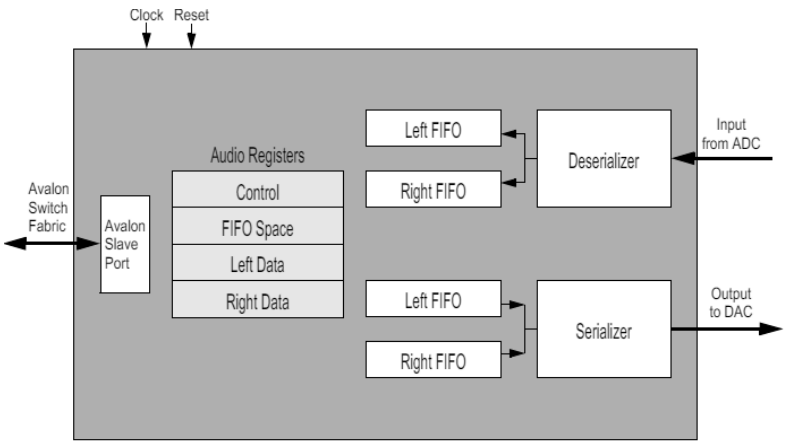
II.d: Codec Driver

For the codec driver that drives the on-board Wolfson WM8731, we simply used a wrapper around the Intel University Program IP Audio (see Figure) and Audio configuration blocks. This wrapper provides an output *advance* signal each time an ADC or signal is ready (not needed in this project) and each time a DAC signal is required for the left and right channels of the codec, i.e. when the IP block's FIFO buffers were not fully filled. The MM-interface is not dire

The codec was automatically configured over I2C so that it takes samples at 48kHz, MSB first of each 24 bits (maximum resolution of the codec). A diagram of the configuration registers are shown below:

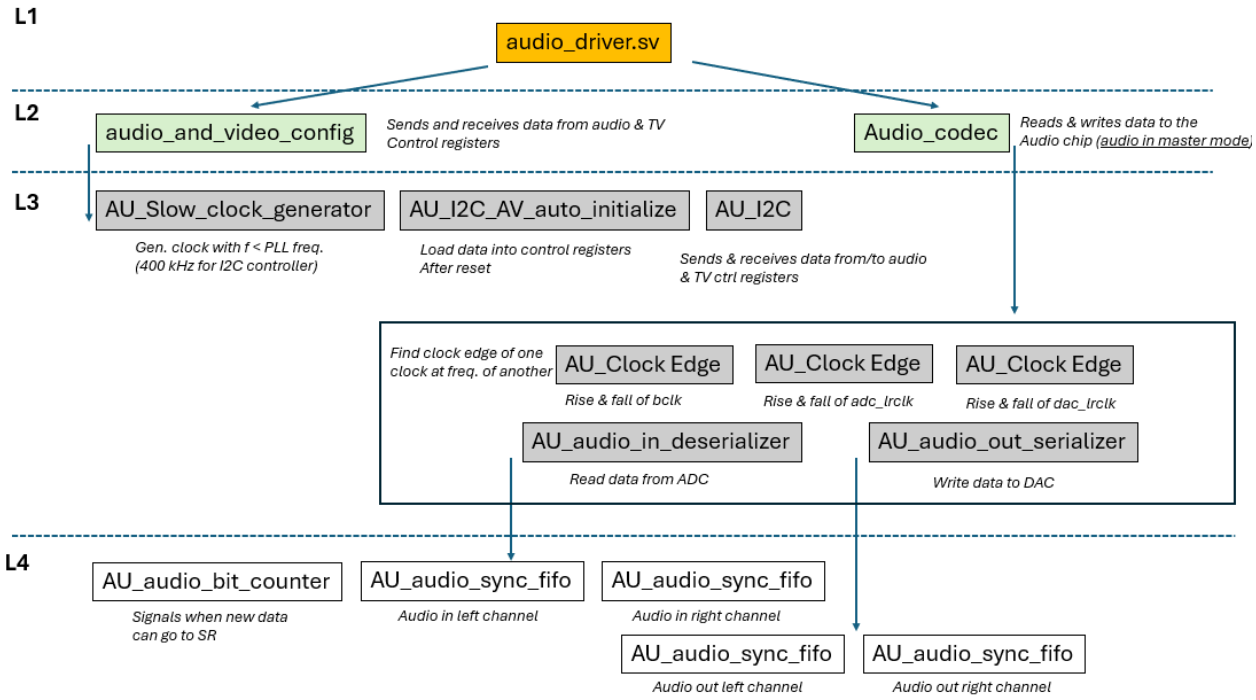
REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	0	0	1	0	0	0	SIDEATT		SIDETONE	DAC SEL	BY PASS	INSEL	MUTE MIC	MIC BOOST
R5 (0Ah)	0	0	0	0	1	0	1	0	0	0	0	HPOR	DAC MU	DEEMPH		ADC HPD
R6 (0Ch)	0	0	0	0	1	1	0	0	PWR OFF	CLK OUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD
R7 (0Eh)	0	0	0	0	1	1	1	0	BCLK INV	MS	LR SWAP	LRP	IWL		FORMAT	
R8 (10h)	0	0	0	1	0	0	0	0	CLKO DIV2	CLKI DIV2	SR				BOSR	USB/ NORM
R9 (12h)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	ACTIVE
R15(1Eh)	0	0	0	1	1	1	1	RESET								
	ADDRESS							DATA								

WM8731 Full Register Map



Intel Audio Core IP (MM-interface not used in this project)

For better understanding of the audio block we used, we made this diagram to show what is going on inside the driver module:



Reverse engineering the audio module we used in this project

III. Pitch Shifting algorithm description

The pitch shifting was done within the driver interface state machine which sends samples to the codec interface as an output. The userspace program sends a value from 0-11, which determines how many semitones to shift the original sample by, allowing us to use the same sample within each octave. In the READ state, instead of adding one to the value being used to

index the samples in the RAM, we kept track of a separate “fractional” index which allowed us to skip certain samples and pitch the note up. The fractional values that were added correspond to the equal temperament ratios used for standardized tunings. Just intonation was considered as it provided much simpler ratios, but this would limit the use of the sequencer as each note would be tuned to the base note, allowing for only sequences in the key of C. This method of pitch shifting also speeds up the sample and introduces harmonics which adds distortion to the sound, but because the sounds we used are short, these effects were barely noticeable, if at all.

Note/Interval		Just Intervals	CENTS	"Pythagorean" (True intervals)	CENTS	Equal Temperament	CENTS
Tonic	C	1	0.00	1	0.00	1	0.00
Minor 2nd	c#	16/15	111.73	256/243	90.22	$2^{1/12}$	100.00
Major 2nd	D	10/9	182.40	9/8	203.91	$2^{1/6}$	200.00
Minor 3rd	e#	6/5	315.64	32/27	294.13	$2^{1/4}$	300.00
Major 3rd	E	5/4	386.31	81/64	407.82	$2^{1/3}$	400.00
Perfect 4th	F	4/3	498.04	4/3	498.04	$2^{5/12}$	500.00
Augmented 4th	f#	45/32	590.22	729/512	611.73	√2	600.00
Diminished 5th	Gb	64/45	609.78	1024/729	588.27		
Perfect 5th	G	3/2	701.96	3/2	701.96	$2^{7/12}$	700.00
Minor 6th	g#	8/5	813.69	128/81	792.18	$2^{2/3}$	800.00
Major 6th	A	5/3	884.36	27/16	905.87	$2^{3/4}$	900.00
Minor 7th	a#	9/5	1017.60	16/9	996.09	$2^{5/6}$	1000.00
Major 7th	B	15/8	1088.27	243/128	1109.78	$2^{11/12}$	1100.00
Octave	C'	2	1200.00	2	1200.00	2	1200.00

IV. References

[1] Wolfson WM8731/WM8731L Audio CODEC
[2] UToronto DE1-SoC tutorial
[3]https://midi.org/basic-of-usb#:~:text=In%201999%2C%20the%20MIDI%20specification,says%20USB%2DAudio%20devices%20connected.
[4] https://www.usb.org/sites/default/files/midi10.pdf
[5] https://cmtext.indiana.edu/MIDI/chapter3_channel_voice_messages.php
[6] Parsing a WAV file in C – Truelogic Blog
[7] Courses 2016–17 - ECAD and Architecture Practical Classes. University of Cambridge.
[8] UToronto Audio Driver

V. Code

V.a. Hardware

driver_interface.sv (contains pitch shifter & RAM module)

```
/*
 * Avalon memory-mapped peripheral that acts as RAM for an audio sample
 * Team EmbeddedSequencer - Columbia University
 *
 ================ TODO ================
 - should be able to return to the store state so that a new sample can be
loaded
 - Add some signals for debuggin

 */

module driver_interface #(parameter MEM_DEPTH = 48000)
(
    // system interface
    input logic clk,
        input logic reset,

    // avalon interface
            input logic [31:0]  writedata,
            input logic write,
            input chipselect,
            input logic address,

    // codec interface
    input logic advance,
    output logic [23:0] leftSample,
    output logic [23:0] rightSample
);

// internal signals
logic [31:0] control;               // register that contains the control
registers
logic [15:0] mem_out;
logic [15:0] w_r_address;           // Need 16 bits to access every memory
address
logic mem_we;
```

```
logic [15:-16] fract_index;
logic [15:-16] fract_index_sum;
logic [15:-16] pitch_shift;
logic [3:0] semitone;

enum {RST, WAIT_STORE, STORE, WAIT_READ, READ, DONE_READ} state;

// Internal Modules
ram #( .MEM_DEPTH(MEM_DEPTH) ) audio_ram
(
    .clk(clk),
    .we(mem_we),
    .addr(w_r_address),
    .d_in(writedata[15:0]),
    .d_out(mem_out)
);

pitch_shifter p1(.semitone(semitone), .pitch_shift(pitch_shift));



// state logic
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;
            mem_we <= 1'b0;
        end else begin
            state <= STORE;
            mem_we <= 1'b1;
        end
      end
    WAIT_READ:
        if (advance) state <= READ;
        else if (write && chipselect) begin // arrival of a new wav file
            w_r_address <= 16'b0;
            fract_index <= 32'b0;
            mem_we <= 1'b1;
            state <= STORE;
        end else begin
            state <= WAIT_READ;
        end
    READ:
        if (!advance) begin      // wait for falling edge
            //if (w_r_address == MEM_DEPTH-1) w_r_address <= 16'b0; //
wraparound
            //else w_r_address <= w_r_address + 16'b1;
            if (w_r_address == MEM_DEPTH - 1) begin fract_index <= 32'b0;
w_r_address <= 16'b0; state <= DONE_READ; end
            else if (fract_index_sum[15:0] > MEM_DEPTH - 1) begin
fract_index <= 32'b0; w_r_address <= 16'b0; state <= DONE_READ; end
            else begin fract_index <= fract_index_sum; w_r_address <=
fract_index_sum[15:0]; state <= WAIT_READ; end
        end else if (write && chipselect) begin // arrival of a new wav
file
            w_r_address <= 16'b0;
            fract_index <= 32'b0;
            mem_we <= 1'b1;
            state <= STORE;
        end else begin
            state <= READ;
        end

    DONE_READ:
```

```
        if (write && chipselect) begin // arrival of a new wav file
            w_r_address <= 16'b0;
            fract_index <= 32'b0;
            mem_we <= 1'b1;
            state <= STORE;
        end else begin
            state <= DONE_READ;
        end
  endcase

// 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

module pitch_shifter(input logic [3:0] semitone,
                     output logic [15:-16] pitch_shift);
always_comb begin
    case(semitone)
        4'd0:       pitch_shift = 32'h0001_0000;
```

```
        4'd1:       pitch_shift = 32'h0001_0f3b;
        4'd2:       pitch_shift = 32'h0001_1f5c;
        4'd3:       pitch_shift = 32'h0001_306f;
        4'd4:       pitch_shift = 32'h0001_4289;
        4'd5:       pitch_shift = 32'h0001_55b5;
        4'd6:       pitch_shift = 32'h0001_6a09;
        4'd7:       pitch_shift = 32'h0001_7f91;
        4'd8:       pitch_shift = 32'h0001_9660;
        4'd9:       pitch_shift = 32'h0001_ae8a;
        4'd10:      pitch_shift = 32'h0001_c824;
        4'd11:      pitch_shift = 32'h0001_e3c3;
        default:    pitch_shift = 32'h0001_0000;
    endcase
end
endmodule
```

ram.sv

```
/*
* Super simple ram module written with the goal of being synthesizable as
* a RAM block
* Team EmbeddedSequencer - Columbia University
*/
module ram #(parameter MEM_DEPTH = 8192)
(
    input logic clk,
    input we,
    input logic [15:0] addr,
    input logic [15:0] d_in,
    output logic [15:0] d_out
);

logic [15:0] mem [MEM_DEPTH-1:0];

always_ff @(posedge clk) begin
    if(we) begin
        mem[addr] <= d_in;
        d_out <= d_in;
    end else d_out <= mem[addr];
```

```
end

endmodule
```

user_interface.sv

```
module button_press (input logic[25:0] horp_time,
                     input logic[25:0] holdup_time,
                     input logic button,
                     input logic clk, //50 MHz Clock input
                     output logic hold,
                     output logic holdup,
                     output logic letgo
                    );

    logic [25:0] cnt;
    enum logic [1:0] {letgo_s, init, horp, hold_s} state;

    always_ff@(posedge clk) begin
        case (state)
            init: begin state <= letgo_s; end

            horp: begin
                if (button == 1'b1) begin
                    state <= letgo_s;
                    cnt <= 0;
                end
                else if (button == 1'b0 && cnt >= horp_time) begin
                    state <= hold_s;
                    cnt <= 0;
                end
                else begin
                    state <= horp;
                    cnt <= cnt + 1'b1;
                end
            end

            hold_s: begin
```

```
                if (button == 1'b1) begin
                    state <= letgo_s;
                    cnt <= 0;
                end
                else if (button == 1'b0 && cnt >= holdup_time) begin
                    state <= hold_s;
                    cnt <= 0;
                end
                else begin
                    cnt <= cnt + 1'b1;
                    state <= hold_s;
                end
            end

            letgo_s: begin
                if (button == 1'b0) begin
                    state <= horp;
                    cnt <= 0;
                end
                else  state <= letgo_s;
            end
            default: state <= letgo_s;
        endcase
    end

    always_comb begin
        case (state)
            init: begin hold = 0; holdup = 0; letgo = 0; end
            horp: begin
                if (cnt >= horp_time) begin hold = 1; holdup = 0; letgo = 0; end
                else begin hold = 0; holdup = 0; letgo = 0;  end
            end
            hold_s: begin
                if (cnt >= holdup_time) begin hold = 1; holdup = 1; letgo = 0;
end
                else begin hold = 1; holdup = 0; letgo = 0;  end
            end
            letgo_s: begin hold = 0; holdup = 0; letgo = 1; end
            default: begin hold = 0; holdup = 0; letgo = 1; end
```

```
        endcase
    end

endmodule


module switch_db    (input logic[25:0] db_time,
                     input logic switch,
                     input logic clk, //50 MHz Clock input
                     output logic sw
                     );

    logic [25:0] cnt;
    logic prev_switch;
    enum logic {letgo_s, debounce} state;


    always_ff@(posedge clk) begin
        case (state)
            debounce: begin
                if (cnt == db_time) begin
                    state <= letgo_s;
                    cnt <= 0;
                end
                else begin
                    cnt <= cnt + 1'b1;
                end
            end

            letgo_s: begin
                if (switch != prev_switch) begin
                    state <= debounce;
                    prev_switch <= switch;
                    cnt <= 0;
                end
                else  begin state <= letgo_s; prev_switch <= switch; end
            end
            default: state <= letgo_s;
        endcase
```

```
    end

    always_comb begin
        case (state)
            debounce: begin
                if (cnt == db_time) begin sw = prev_switch; end
                else begin  sw = ~prev_switch;  end
            end
            letgo_s: begin   sw = prev_switch; end
            default: begin sw = prev_switch; end
        endcase
    end

endmodule




module hex7seg(input logic  [11:0] a,
               output logic [6:0] x,
               output logic [6:0] y,
               output logic [6:0] z
               );
            logic [11:0] tens;
            logic blank;

            hex7seg_tens tens_place(.a(tens), .blank(blank), .y(y), .z(z));

             always_comb begin
                if (a == 12'd300) begin
                    x = 7'b011_0000;
                    tens = a - 12'd300;
                    blank =1'b0;
                end
                else if (a >= 12'd200 && a < 12'd300) begin
                    x = 7'b010_0100;
                    tens = a - 12'd200;
                    blank = 1'b0;
```

```
                end
                else if (a >= 12'd100 && a < 12'd200) begin
                    x = 7'b111_1001;
                    tens = a - 12'd100;
                    blank = 1'b0;
                end
                else begin
                    x = 7'b111_1111;
                    tens = a;
                    blank = 1'b1;
                end
            end
endmodule



module hex7seg_tens(input logic  [11:0] a,
               input logic blank,
               output logic [6:0] y,
               output logic [6:0] z
               );
            logic [11:0] ones;

            hex7seg_ones ones_place(.a(ones), .z(z));

            always_comb begin
                if (a >= 12'd90 && a < 12'd100) begin
                    y = 7'b001_0000;
                    ones = a - 12'd90;
                end
                else if (a >= 12'd80 && a < 12'd90) begin
                    y = 7'b000_0000;
                    ones = a - 12'd80;
                end
                else if (a >= 12'd70 && a < 12'd80) begin
                    y = 7'b111_1000;
                    ones = a - 12'd70;
                end
                else if (a >= 12'd60 && a < 12'd70) begin
```

```
                y = 7'b000_0010;
                ones = a - 12'd60;
            end
            else if (a >= 12'd50 && a < 12'd60) begin
                y = 7'b001_0010;
                ones = a - 12'd50;
            end
            else if (a >= 12'd40 && a < 12'd50) begin
                y = 7'b001_1001;
                ones = a - 12'd40;
            end
            else if (a >= 12'd30 && a < 12'd40) begin
                y = 7'b011_0000;
                ones = a - 12'd30;
            end
            else if (a >= 12'd20 && a < 12'd30) begin
                y = 7'b010_0100;
                ones = a - 12'd20;
            end
            else if (a >= 12'd10 && a < 12'd20) begin
                y = 7'b111_1001;
                ones = a - 12'd10;
            end
            else begin
                ones = a;
                if (blank) begin
                    y = 7'b111_1111;
                end
                else begin
                    y = 7'b100_0000;
                end
            end
        end
endmodule

module hex7seg_ones(input logic  [11:0] a,
            output logic [6:0] z
            );
```

```
            always_comb begin
                case(a)
                  12'd0:    z = 7'b100_0000;
                  12'd1:    z = 7'b111_1001;
                  12'd2:    z = 7'b010_0100;
                  12'd3:    z = 7'b011_0000;
                  12'd4:    z = 7'b001_1001;
                  12'd5:    z = 7'b001_0010;
                  12'd6:    z = 7'b000_0010;
                  12'd7:    z = 7'b111_1000;
                  12'd8:    z = 7'b000_0000;
                  12'd9:    z = 7'b001_0000;
                  default: z = 7'b111_1111;
                endcase
            end

endmodule



module user_interface(
        input logic        CLOCK_50,  // 50 MHz Clock input

        input logic [3:0]  KEY, // Pushbuttons; KEY[0] is rightmost

        input logic [9:0]  SW, // Switches; SW[0] is rightmost

        input logic        reset,

        input logic        address,

        input logic        read,

        input logic        chipselect,

        // 7-segment LED displays; HEX0 is rightmost
        output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,

        output logic [9:0] LEDR, // LEDs above the switches; LED[0] on right
```

```
      output logic [15:0] readdata
      );

logic           clk;
logic [25:0] db_time;
logic [25:0] horp_time;
logic [25:0] holdup_time;
logic hold0, holdup0, letgo0;
logic hold1, holdup1, letgo1;
logic hold2, holdup2, letgo2;
logic hold3, holdup3, letgo3;
logic sw9, sw8;
assign db_time = 26'd1_000_000; //20 ms
assign horp_time = 26'd50_000_000; //1 s
assign holdup_time = 26'd2_500_000; //50 ms
assign clk = CLOCK_50;

button_press b3(.horp_time(horp_time), .holdup_time(holdup_time),
                .button(KEY[3]),
                .clk(CLOCK_50),
                .hold(hold3), .holdup(holdup3), .letgo(letgo3)
                );
button_press b2(.horp_time(horp_time), .holdup_time(holdup_time),
                .button(KEY[2]),
                .clk(CLOCK_50),
                .hold(hold2), .holdup(holdup2), .letgo(letgo2)
                );

button_press b1(.horp_time(horp_time), .holdup_time(holdup_time),
                .button(KEY[1]),
                .clk(CLOCK_50),
                .hold(hold1), .holdup(holdup1), .letgo(letgo1)
                );

button_press b0(.horp_time(horp_time), .holdup_time(holdup_time),
                .button(KEY[0]),
                .clk(CLOCK_50),
                .hold(hold0), .holdup(holdup0), .letgo(letgo0)
```

```
                    );
    switch_db s9(.db_time(db_time),
                 .switch(SW[9]),
                 .clk(CLOCK_50),
                 .sw(sw9)
                 );
    switch_db s8(.db_time(db_time),
                 .switch(SW[8]),
                 .clk(CLOCK_50),
                 .sw(sw8)
                 );

    enum logic [2:0] {init, playback, bpm, step, button_step, button_bpm,
button_track, dummy} state, prev_state;
    logic [15:0] bpm_count;
    logic [7:0]  step_count;
    logic [7:0]  track_count;
    logic        playback_mode;
    logic [11:0] left_hex;
    logic [11:0] right_hex;
    logic add;

    assign LEDR[4] = state == button_step;
    assign LEDR[5] = state == button_bpm;
    assign LEDR[6] = state == button_track;
    assign LEDR[7] = state == dummy;


    hex7seg bpm_press  (.a(left_hex),  .x(HEX5), .y(HEX4), .z(HEX3));
    hex7seg step_press (.a(right_hex), .x(HEX2), .y(HEX1), .z(HEX0));

    always_ff @(posedge clk)
    begin
        if (reset) begin
            bpm_count <= 16'd100;
            step_count <= 8'd1;
            track_count <= 8'd1;
            state <= init;
            prev_state <= init;
```

```
        end

        case (state)
        init: begin
            bpm_count <= 16'd100; step_count <= 8'd1; track_count <= 8'd1;
            if (sw9) begin state <= playback; end
            else if (!sw9 && sw8) state <= bpm;
            else if (!sw9 && !sw8) state <= step;
            else state <= step;
        end
        playback: begin
            if (sw9) state <= playback;
            else if (!sw9 && sw8) state <= bpm;
            else if (!sw9 && !sw8) state <= step;
            else state <= playback;
        end
        bpm: begin
            if (KEY[1] == 1'b0) begin state <= button_bpm; prev_state <= bpm; add
= 1; end
            else if (KEY[0] == 1'b0) begin state <= button_bpm; prev_state <=
bpm; add = 0; end
            else if (sw9) begin state <= playback; end
            else if (!sw9 && sw8) state <= bpm;
            else if (!sw9 && !sw8) state <= step;
            else state <= bpm;
        end
        step: begin
            if (KEY[3] == 1'b0) begin state <= button_step; prev_state <= step;
add <= 1; end
            else if (KEY[2] == 1'b0) begin state <= button_step; prev_state <=
step; add <= 0; end
            else if (KEY[1] == 1'b0) begin state <= button_track; prev_state <=
step; add <= 1; end
            else if (KEY[0] == 1'b0) begin state <= button_track; prev_state <=
step; add <= 0; end
            else if (sw9) state <= playback;
            else if (!sw9 && sw8) state <= bpm;
            else if (!sw9 && !sw8) state <= step;
            else state <= step;
```

```
        end
        button_step: begin
            //add or sub only once (check if prev state is step)
            if (prev_state == step) begin
                prev_state <= button_step;
                state <= button_step;

                if (add == 1) begin
                    if (step_count == 8'd16) step_count <= 8'd1;
                    else                     step_count <= step_count + 8'd1;
                end
                else begin
                    if (step_count == 8'd1)   step_count <= 8'd16;
                    else                      step_count <= step_count - 8'd1;
                end
            end
            else begin
                prev_state <= button_step;
                state <= button_step;
                if ((letgo3 && add == 1) || (letgo2 && add == 0)) begin
                    state <= step;
                end
            end

        end
        button_bpm: begin
            //if prev state was bpm (just once before holding) or hold and
            //holdup and prev state == button_bpm, add or sub
            if ((prev_state == bpm) ||
                (hold1 && holdup1 && prev_state == button_bpm && add == 1) ||
                (hold0 && holdup0 && prev_state == button_bpm && add == 0))
begin

                prev_state <= button_bpm;
                state <= button_bpm;

                if (add == 1) begin
                    if (bpm_count == 16'd300) bpm_count <= 16'd40;
                    else                      bpm_count <= bpm_count + 16'd1;
```

```
            end
            else begin
                if (bpm_count == 16'd40) bpm_count <= 16'd300;
                else                     bpm_count <= bpm_count - 16'd1;
            end
        end
        else begin
            prev_state <= button_bpm;
            state <= button_bpm;
            if ((letgo1 && add == 1) || (letgo0 && add == 0)) begin
                state <= bpm;
            end
        end
    end

    button_track: begin
        //add or sub only once (check if prev state is step)
        if (prev_state == step) begin
            prev_state <= button_track;
            state <= button_track;

            if (add == 1) begin
                if (track_count == 8'd4)  track_count <= 8'd1;
                else                      track_count <= track_count + 8'd1;
            end
            else begin
                if (track_count == 8'd1)   track_count <= 8'd4;
                else                       track_count <= track_count - 8'd1;
            end
        end
        else begin
            prev_state <= button_track;
            state <= button_track;
            if ((letgo1 && add == 1) || (letgo0 && add == 0)) begin
                state <= step;
            end
        end
    end
```

```
        dummy:   state <= init;
        default: state <= init;
        endcase
    end

    always_comb
    begin
        if (reset) begin
            readdata[15:0] = 16'b0;
            playback_mode = 1'b0;
            left_hex = {4'b0, step_count[7:0]};
            right_hex = {4'b0, track_count[7:0]};
        end

        else if (chipselect && read) begin
            case (address)
                1'b0: begin readdata[15:0] = {playback_mode, bpm_count[14:0]};
end
                1'b1: begin readdata[15:0] = {track_count, step_count}; end
                default: readdata[15:0] = 16'b0;
            endcase

            case (state)
                init:           begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                playback:       begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b1; end
                bpm:            begin left_hex = 12'd400;
right_hex = bpm_count[11:0];          playback_mode = 1'b0; end
                step:           begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                button_step:    begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                button_bpm:     begin left_hex = 12'd400;
right_hex = bpm_count[11:0];          playback_mode = 1'b0; end
                button_track:   begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                dummy:          begin left_hex = 12'd400;
right_hex = 12'd400;                  playback_mode = 1'b0; end
```

```
                default:        begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
            endcase
        end

        else begin
            readdata[15:0] = 16'b0;
            case (state)
                init:           begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                playback:       begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b1; end
                bpm:            begin left_hex = 12'd400;
right_hex = bpm_count[11:0];          playback_mode = 1'b0; end
                step:           begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                button_step:    begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                button_bpm:     begin left_hex = 12'd400;
right_hex = bpm_count[11:0];          playback_mode = 1'b0; end
                button_track:   begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
                dummy:          begin left_hex = 12'd400;
right_hex = 12'd400;                  playback_mode = 1'b0; end
                default:        begin left_hex = {4'b0, step_count[7:0]};
right_hex = {4'b0, track_count[7:0]}; playback_mode = 1'b0; end
            endcase
        end
    end

endmodule
```

V.b. Software

Wav_handler.h

```
/*
* Team EmbeddedSequencer - Columbia University
* Handles loading of .wav files
* source:
https://truelogic.org/wordpress/2015/09/04/parsing-a-wav-file-in-c/
*/
#ifndef _WAV_HANDLER_H
#define _WAV_HANDLER_H

// WAVE file header format
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 byterate;                              // 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 FLLR
string
      unsigned int data_size;                             // NumSamples *
NumChannels * BitsPerSample/8 - size of the next chunk that will be read
};


struct HEADER header;
```

```
int read_wav(int**data, char* filename, int verbose);

#endif
```

wav_handler.c

```
/*
* Team EmbeddedSequencer - Columbia University
* Handles loading of .wav files (Only the first channel is read if multiple
channels are detected)
* adapted from:
https://truelogic.org/wordpress/2015/09/04/parsing-a-wav-file-in-c/
*/

#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "wav_handler.h"
#define TRUE 1
#define FALSE 0


int read_wav(int**data, char* filename, int verbose);
char* seconds_to_time(float seconds);

FILE *ptr;
struct HEADER header;

// struct HEADER header;

/*
// make this file callable for debugging purposes
int main(int argc, char** argv){
      int** data;
      int num_samples;

      if(argc != 2){
            printf("Invalid number of input arguments. Exiting...");
            exit(1);
      }
```

```
	num_samples = read_wav(data, argv[1], 1);
	free(*data);

	return 0;
}
*/

// returns size of allocated data (number of int)
int read_wav(int** data, char* filename, int verbose) {
	unsigned char buffer4[4];
	unsigned char buffer2[2];



	if (verbose) printf("Opening  file: %s\n", filename);

	ptr = fopen(filename, "rb");
	if (ptr == NULL) {
		printf("Error opening file \n");
		exit(1);
	}

	int read = 0;

	// read header parts
	read = fread(header.riff, sizeof(header.riff), 1, ptr);
	if (verbose) printf("(1-4): %s \n", header.riff);

	read = fread(buffer4, sizeof(buffer4), 1, ptr);
	if (verbose) printf("%u %u %u %u\n", buffer4[0], buffer4[1],
buffer4[2], buffer4[3]);

	// convert little endian to big endian 4 byte int
	header.overall_size  = buffer4[0] |
	                       (buffer4[1]<<8) |
	                       (buffer4[2]<<16) |
	                       (buffer4[3]<<24);

	if (verbose) printf("(5-8) Overall size: bytes:%u, Kb:%u\n",
header.overall_size, header.overall_size/1024);

	read = fread(header.wave, sizeof(header.wave), 1, ptr);
```

```
      if (verbose) printf("(9-12) Wave marker: %s\n", header.wave);

      read = fread(header.fmt_chunk_marker,
sizeof(header.fmt_chunk_marker), 1, ptr);
      if (verbose) printf("(13-16) Fmt marker: %s\n",
header.fmt_chunk_marker);

      read = fread(buffer4, sizeof(buffer4), 1, ptr);
      if (verbose) printf("%u %u %u %u \n", buffer4[0], buffer4[1],
buffer4[2], buffer4[3]);

      // convert little endian to big endian 4 byte integer
      header.length_of_fmt = buffer4[0] |
                                             (buffer4[1] << 8) |
                                             (buffer4[2] << 16) |
                                             (buffer4[3] << 24);
      if (verbose) printf("(17-20) Length of Fmt header: %u \n",
header.length_of_fmt);

      read = fread(buffer2, sizeof(buffer2), 1, ptr);
      if (verbose) printf("%u %u \n", buffer2[0], buffer2[1]);

      header.format_type = buffer2[0] | (buffer2[1] << 8);
      char format_name[10] = "";
      if (header.format_type == 1)
            strcpy(format_name,"PCM");
      else if (header.format_type == 6)
            strcpy(format_name, "A-law");
      else if (header.format_type == 7)
            strcpy(format_name, "Mu-law");

      if (verbose) printf("(21-22) Format type: %u %s \n",
header.format_type, format_name);

      read = fread(buffer2, sizeof(buffer2), 1, ptr);
      if (verbose) printf("%u %u \n", buffer2[0], buffer2[1]);

      header.channels = buffer2[0] | (buffer2[1] << 8);
      if (verbose) printf("(23-24) Channels: %u \n", header.channels);

      read = fread(buffer4, sizeof(buffer4), 1, ptr);
      if (verbose)  printf("%u %u %u %u\n", buffer4[0], buffer4[1],
buffer4[2], buffer4[3]);
```

```
      header.sample_rate = buffer4[0] |
                                        (buffer4[1] << 8) |
                                        (buffer4[2] << 16) |
                                        (buffer4[3] << 24);

      if (verbose) printf("(25-28) Sample rate: %u\n", header.sample_rate);

      read = fread(buffer4, sizeof(buffer4), 1, ptr);
      if (verbose) printf("%u %u %u %u\n", buffer4[0], buffer4[1],
buffer4[2], buffer4[3]);

      header.byterate  = buffer4[0] |
                                        (buffer4[1] << 8) |
                                        (buffer4[2] << 16) |
                                        (buffer4[3] << 24);
      if (verbose) printf("(29-32) Byte Rate: %u , Bit Rate:%u\n",
header.byterate, header.byterate*8);

      read = fread(buffer2, sizeof(buffer2), 1, ptr);
      if (verbose) printf("%u %u \n", buffer2[0], buffer2[1]);

      header.block_align = buffer2[0] |
                                    (buffer2[1] << 8);
      if (verbose) printf("(33-34) Block Alignment: %u \n",
header.block_align);

      read = fread(buffer2, sizeof(buffer2), 1, ptr);
      if (verbose) printf("%u %u \n", buffer2[0], buffer2[1]);

      header.bits_per_sample = buffer2[0] |
                                    (buffer2[1] << 8);
      if (verbose) printf("(35-36) Bits per sample: %u \n",
header.bits_per_sample);

      read = fread(header.data_chunk_header,
sizeof(header.data_chunk_header), 1, ptr);
      if (verbose) printf("(37-40) Data Marker: %s \n",
header.data_chunk_header);

      read = fread(buffer4, sizeof(buffer4), 1, ptr);
      if (verbose) printf("%u %u %u %u\n", buffer4[0], buffer4[1],
buffer4[2], buffer4[3]);
```

```
      header.data_size = buffer4[0] |
                              (buffer4[1] << 8) |
                              (buffer4[2] << 16) |
                              (buffer4[3] << 24 );
      if (verbose) printf("(41-44) Size of data chunk: %u \n",
header.data_size);

      // calculate no.of samples
      long num_samples = (8 * header.data_size) / (header.channels *
header.bits_per_sample);
      if (verbose) printf("Number of samples: %lu \n", num_samples);

      long size_of_each_sample = (header.channels * header.bits_per_sample)
/ 8;
      if (verbose) printf("Size of each sample: %ld bytes\n",
size_of_each_sample);

      // calculate duration of file
      float duration_in_seconds = (float) header.overall_size /
header.byterate;
      if (verbose) printf("Approx.Duration in seconds = %f\n",
duration_in_seconds);
      if (verbose) printf("Approx.Duration in h:m:s = %s\n",
seconds_to_time(duration_in_seconds));

      // read each sample from data chunk if PCM
      if (header.format_type != 1) { // make sure data chunk is PCM
            printf("File is not PCM. Terminating Program... \n");
            exit(1);
      }
      long i =0;
      signed char data_buffer[size_of_each_sample];

      // make sure that the bytes-per-sample is completely divisible by
num.of channels
      long bytes_in_each_channel = (size_of_each_sample / header.channels);
      if ((bytes_in_each_channel  * header.channels) !=
size_of_each_sample) {
            printf("Error: %ld x %ud <> %ld\n", bytes_in_each_channel,
header.channels, size_of_each_sample);
            exit(1);
      }
```

```
	long low_limit = 0l;
	long high_limit = 0l;

	switch (header.bits_per_sample) {
		case 8:
			low_limit = -128;
			high_limit = 127;
			break;
		case 16:
			low_limit = -32768;
			high_limit = 32767;
			break;
		case 32:
			low_limit = -2147483648;
			high_limit = 2147483647;
			break;
	}
	if (verbose) printf("\nValid range for data values : %ld to %ld \n",
low_limit, high_limit);

	// allocate memory for storing the samples
	if (verbose) printf("Allocating heap memory for the samples. %ld
ints, each size %d \n", num_samples, sizeof(int));
	*data = malloc(sizeof(int) * num_samples);
	if (*data == NULL){
		printf("malloc for storing .wav samples failed. Exiting...");
		exit(1);
	}
	if (verbose) printf("Succesfully allocated heap memory \n");

	// parse samples
	for (i =1; i <= num_samples; i++) {
		// printf("==========Sample %ld / %ld=============\n", i,
num_samples);
		read = fread(data_buffer, sizeof(data_buffer), 1, ptr);
		if (read) {
			// dump the data read
			unsigned int  xchannels = 0;
			signed int data_in_channel = 0;
			int offset = 0; // move the offset for every iteration in
the loop below
			for (xchannels = 0; xchannels < header.channels;
```

```
xchannels ++ ) {
                        // printf("Channel#%d : ", (xchannels+1));
                        // convert data from little endian to big endian
based on bytes in each channel sample
                        if (bytes_in_each_channel == 4) {
                              printf("Warning: 32 bit sample detected. Is
something wrong?");
                              data_in_channel = (data_buffer[offset] &
0x00ff) |

((data_buffer[offset + 1] & 0x00ff) <<8) |

((data_buffer[offset + 2] & 0x00ff) <<16) |

(data_buffer[offset + 3]<<24);
                        }
                        else if (bytes_in_each_channel == 2) {
                              data_in_channel = (data_buffer[offset] &
0x00ff) | (data_buffer[offset + 1] << 8);
                              // printf("sample = %d, size of sample = %d
\n", data_in_channel, sizeof(data_in_channel));
                        }
                        else if (bytes_in_each_channel == 1) {
                              printf("Warning: 8 bit sample detected. Is
something wrong?");
                              data_in_channel = data_buffer[offset] &
0x00ff;
                              data_in_channel -= 128; //in wave, 8-bit are
unsigned, so shifting to signed
                        }

                        offset += bytes_in_each_channel;
                        (*data)[i] = data_in_channel;

                        // check if value was in range
                        if (data_in_channel < low_limit || data_in_channel
> high_limit)
                              printf("**warning: value out of range
detected\n");
                  }
            }
      }
```

```
      if (verbose) printf("Closing file..\n");
      fclose(ptr);

      return num_samples;
}

/**
 * Convert seconds into hh:mm:ss format
 * Params:
 *    seconds - seconds value
 * Returns: hms - formatted string
 **/
char* seconds_to_time(float raw_seconds) {
      char *hms;
      int hours, hours_residue, minutes, seconds, milliseconds;
      hms = (char*) malloc(100);

      sprintf(hms, "%f", raw_seconds);

      hours = (int) raw_seconds/3600;
      hours_residue = (int) raw_seconds % 3600;
      minutes = hours_residue/60;
      seconds = hours_residue % 60;
      milliseconds = 0;

      // get the decimal part of raw_seconds to get milliseconds
      char *pos;
      pos = strchr(hms, '.');
      int ipos = (int) (pos - hms);
      char decimalpart[15];
      memset(decimalpart, ' ', sizeof(decimalpart));
      strncpy(decimalpart, &hms[ipos+1], 3);
      milliseconds = atoi(decimalpart);

      sprintf(hms, "%d:%d:%d.%d", hours, minutes, seconds, milliseconds);
      return hms;
}
```

Audio_driver.h

```
\#ifndef _AUDIO_DRIVER_H
```

```
#define _AUDIO_DRIVER_H

#include <linux/ioctl.h>

typedef struct {
  int audio_sample; // write 32b word to the device over the Avalon bus
} audio_arg_t;

#define AUDIO_MAGIC 'q'

/* ioctls and their arguments */
#define AUDIO_WRITE _IOW(AUDIO_MAGIC, 1, audio_arg_t *)

#endif
```

Audio_driver.c

```
/* * Device driver for the WM8731 Audio Codec
 *
 * A Platform device implemented using the misc subsystem
 *
 * Team EmbeddedSequencer - Adapted from Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod audio_driver.ko
 *
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
```

```
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "audio_driver.h"


#define DRIVER_NAME "audio_driver"

#define CHANNEL_MASK 0x30000

/* Device registers */
#define REG_CONTROL(x) (x)
#define REG_AUDIO1(x) ((x)+4) // +8
#define REG_AUDIO2(x) ((x)+8) //+12
#define REG_AUDIO3(x) ((x)+12) //+16
#define REG_AUDIO4(x) ((x)+16) //+20

/*
 * Information about our device
 */
struct audio_dev {
      struct resource res; /* Resource: our registers */
      void __iomem *virtbase; /* Where registers can be accessed in memory
*/
      int current_sample;
} dev;

/*
 * Write single sample to the device
 * Assumes digit is in range and the device information has been set up
 */
static void write_sample(int* sample)
{
      unsigned int channel_number = (*sample & CHANNEL_MASK) >> 16;

      if(channel_number == 0x0) iowrite32(*sample,REG_AUDIO1(dev.virtbase)
);
      else if (channel_number == 0x1)
iowrite32(*sample,REG_AUDIO2(dev.virtbase) );
```

```
      else if (channel_number == 0x2)
iowrite32(*sample,REG_AUDIO3(dev.virtbase) );
        else {
        iowrite32(*sample,REG_AUDIO4(dev.virtbase)) ;

      }

//iowrite32(*sample, REG_AUDIO1(dev.virtbase) );
      dev.current_sample = *sample;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long audio_ioctl(struct file *f, unsigned int cmd, unsigned long
arg)
{
      audio_arg_t vla;

      switch (cmd) {
      // only have a write operation for the audio
      case AUDIO_WRITE:
            if (copy_from_user(&vla, (audio_arg_t *) arg,
                        sizeof(audio_arg_t)))
                  return -EACCES;
            write_sample(&(vla.audio_sample));
            break;

      default:
            return -EINVAL;
      }

      return 0;
}

/* The operations our device knows how to do */
static const struct file_operations audio_fops = {
      .owner            = THIS_MODULE,
      .unlocked_ioctl = audio_ioctl,
};
```

```
/* Information about our device for the "misc" framework -- like a char dev
*/
static struct miscdevice audio_misc_device = {
      .minor            = MISC_DYNAMIC_MINOR,
      .name       = DRIVER_NAME,
      .fops       = &audio_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init audio_probe(struct platform_device *pdev)
{
      int ret;

      /* Register ourselves as a misc device: creates /dev/audio */
      ret = misc_register(&audio_misc_device);

      /* Get the address of our registers from the device tree */
      ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
      if (ret) {
            ret = -ENOENT;
            goto out_deregister;
      }

      /* Make sure we can use these registers */
      if (request_mem_region(dev.res.start, resource_size(&dev.res),
                        DRIVER_NAME) == NULL) {
            ret = -EBUSY;
            goto out_deregister;
      }

      /* Arrange access to our registers */
      dev.virtbase = of_iomap(pdev->dev.of_node, 0);
      if (dev.virtbase == NULL) {
            ret = -ENOMEM;
            goto out_release_mem_region;
      }

      return 0;

out_release_mem_region:
```

```
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&audio_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int audio_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&audio_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id audio_of_match[] = {
	{ .compatible = "csee4840,driver_interface-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, audio_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver audio_driver = {
	.driver	= {
		.name = DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(audio_of_match),
	},
	.remove	= __exit_p(audio_remove),
};

/* Called when the module is loaded: set things up */
static int __init audio_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&audio_driver, audio_probe);
}

/* Calback when the module is unloaded: release resources */
static void __exit audio_exit(void)
```

```
{
	platform_driver_unregister(&audio_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(audio_init);
module_exit(audio_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Team EmbeddedSequencer & Stephen A. Edwards, Columbia
University");
MODULE_DESCRIPTION("Audio driver");
```

User_interface.h

```
#ifndef _USER_INTERFACE_H
#define _USER_INTERFACE_H

#include <linux/ioctl.h>
#include <linux/types.h>

typedef struct {
	unsigned short bpm;
	unsigned char step;
	unsigned char track;
	unsigned char playback;
} user_interface_props_t;

typedef struct {
  user_interface_props_t props;
} user_interface_arg_t;

#define USER_INTERFACE_MAGIC 'q'

/* ioctls and their arguments */
//#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t
*)
//#define VGA_BALL_READ_BACKGROUND  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t
*)
#define USER_INTERFACE_WRITE_PROPS _IOW(USER_INTERFACE_MAGIC, 1,
user_interface_props_t *)
```

```
#define USER_INTERFACE_READ_PROPS  _IOR(USER_INTERFACE_MAGIC, 2,
user_interface_props_t *)
#endif
```

User_interface.c

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>
#include "user_interface.h"
```

```
#define DRIVER_NAME "user_interface"

/* Device registers */
//#define BG_RED(x) (x)
//#define BG_GREEN(x) ((x)+1)
//#define BG_BLUE(x) ((x)+2)

#define UI_BPM_PLAYBACK(x) ((x)+20)
#define UI_STEP_TRACK(x) ((x)+22)


/*
 * Information about our device
 */
struct user_interface_dev {
      struct resource res; /* Resource: our registers */
      void __iomem *virtbase; /* Where registers can be accessed in memory
*/
      user_interface_props_t props;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
*/

/*
static void write_background(vga_ball_color_t *background)
{
      iowrite8(background->red, BG_RED(dev.virtbase) );
      iowrite8(background->green, BG_GREEN(dev.virtbase) );
      iowrite8(background->blue, BG_BLUE(dev.virtbase) );
      dev.background = *background;
}
*/

// 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 >> 8);
        props->bpm =  (unsigned short)(bpm_playback & 0x00007FFF);
      printk(KERN_INFO "Here: %hu", props->bpm);
      props->playback = (unsigned char) ((bpm_playback & 0x00008000) >>
15);
      dev.props = *props;
}


/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long user_interface_ioctl(struct file *f, unsigned int cmd, unsigned
long arg)
{
      //vga_ball_arg_t vla;
      user_interface_props_t props;

      switch (cmd) {
      /*case VGA_BALL_WRITE_BACKGROUND:
            if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                        sizeof(vga_ball_arg_t)))
                  return -EACCES;
            write_background(&vla.background);
            break;

      case VGA_BALL_READ_BACKGROUND:
            vla.background = dev.background;
            if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                        sizeof(vga_ball_arg_t)))
                  return -EACCES;
            break;
      // NEW CASES HERE: */
      case USER_INTERFACE_WRITE_PROPS:
            if (copy_from_user(&props,(user_interface_props_t * ) arg,
sizeof(user_interface_props_t )))
                  return -EACCES;
            break;

      case USER_INTERFACE_READ_PROPS:
            read_props(&props);
```

```
            if (copy_to_user((user_interface_props_t *) arg, &props,
sizeof(user_interface_props_t)))
                  return -EACCES;
            break;


      default:
            return -EINVAL;
      }

      return 0;
}

/* The operations our device knows how to do */
static const struct file_operations user_interface_fops = {
      .owner            = THIS_MODULE,
      .unlocked_ioctl = user_interface_ioctl,



};

/* Information about our device for the "misc" framework -- like a char dev
*/
static struct miscdevice user_interface_misc_device = {
      .minor            = MISC_DYNAMIC_MINOR,
      .name       = DRIVER_NAME,
      .fops       = &user_interface_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init user_interface_probe(struct platform_device *pdev)
{
      int ret;

      /* Register ourselves as a misc device: creates /dev/vga_ball */
      ret = misc_register(&user_interface_misc_device);

      /* Get the address of our registers from the device tree */
      ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
```

```
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&user_interface_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int user_interface_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&user_interface_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id user_interface_of_match[] = {
	{ .compatible = "csee4840,user_interface-1.0" },
	{},
};
```

```
MODULE_DEVICE_TABLE(of, user_interface_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver user_interface_driver = {
	.driver	= {
		.name = DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(user_interface_of_match),
	},
	.remove	= __exit_p(user_interface_remove),
};

/* Called when the module is loaded: set things up */
static int __init user_interface_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&user_interface_driver,
user_interface_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit user_interface_exit(void)
{
	platform_driver_unregister(&user_interface_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(user_interface_init);
module_exit(user_interface_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Brandon Cruz, Columbia University");
MODULE_DESCRIPTION("Sequencer UI driver");
```

Usbmidi.h

```
#ifndef _USBMIDI_H
#define _USBMIDI_H

#include <libusb-1.0/libusb.h>
```

```
#define USB_HID_MIDI_PROTOCOL 1

/* Modifier bits */



struct usb_midi_packet {
  uint8_t status;
  uint8_t keycode[3];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
   device was found. */
extern struct libusb_device_handle *openmidi(uint8_t *);

#endif
```

Usbmidi.c

```
#include "usbmidi.h"

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

/* References on libusb 1.0 and the USB HID/keyboard protocol
 *
 * http://libusb.org
 *
https://web.archive.org/web/20210302095553/https://www.dreamincode.net/foru
ms/topic/148707-introduction-to-using-libusb-10/
 *
 * https://www.usb.org/sites/default/files/documents/hid1_11.pdf
 *
 * https://usb.org/sites/default/files/hut1_5.pdf
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 * The argument con
```

```
 *
 */
struct libusb_device_handle *openmidi(uint8_t *endpoint_address) {
  libusb_device **devs;
  struct libusb_device_handle *midi = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    printf("Device speed %d \n", libusb_get_device_speed(dev));

    if (desc.bDeviceClass == 0){ // &&  desc.bDeviceSubClass == 0x03 &&
desc.bDeviceProtocol == 0x00 ) {



      struct libusb_config_descriptor *config;

      libusb_get_config_descriptor(dev, 0, &config);
      for (i = 0 ; i < config->bNumInterfaces ; i++){
      printf("Device no. %d, interface no. %d \n", d, i);
```

```
      for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {

          const struct libusb_interface_descriptor *inter =
          config->interface[i].altsetting + k ;

          if ( inter->bInterfaceClass == 1 &&
             inter->bInterfaceProtocol == 0 &&
            inter->bInterfaceSubClass == 3) {
          int r;
            printf("selected interface = %d \n", i);

          if ((r = libusb_open(dev, &midi)) != 0) {
            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
            exit(1);
          }

          if (libusb_kernel_driver_active(midi,i))
            libusb_detach_kernel_driver(midi, i);

          libusb_set_auto_detach_kernel_driver(midi, i);

          if ((r = libusb_claim_interface(midi, i)) != 0) {
            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n",
r);
            exit(1);

          }
          printf("bmAttributes, %d \n ",inter->endpoint[1].bmAttributes);

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

 found:
  libusb_free_device_list(devs, 1);

  return midi;
```

```
}
```

Hello.c (contains main)

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stddef.h>
#include <math.h>
#include "usbmidi.h"
#include "user_interface.h"
#include "wav_handler.h"
#include "audio_driver.h"

int audio_driver_fd;
int user_interface_fd;
int MIN_KEY_CODE = 18;
int MAX_KEYCODE = 60;
struct libusb_device_handle *midi;
uint8_t endpoint_address;

void read_props(user_interface_props_t *props)
{
  user_interface_arg_t vla;

  if (ioctl(user_interface_fd, USER_INTERFACE_READ_PROPS, props)) {
        perror("ioctl(USER_INTERFACE_READ_PROPS) failed");
        return;
  }

// props->props.bpm = vla.props.bpm;
// props->props.step = vla.props.step;
// props->props.track = vla.props.track;
// props->props.playback = vla.props.playback;
}
```

```
typedef struct NoteInfo {
        int noteVal;
        int octave;
        int noteIndex;
} NoteInfo;



NoteInfo mapCodeToNote(int num) {
   NoteInfo result;
   if(num != 0){
       num = num-6; //bias
       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;
 };



// #playback mode or live mode: 1 bit (live 0 , playbak 1)
// #active channels: 4 bits (channel 3  on/off, channel 2 on/off, channel 1
on/off, channel 0 on/off)
// shift: 4 bits #0 -11
// velocty: 3 bits
// channel: 2 bits (channel we are writing on)
// sample: 16 bits
// TOTAL: 30 bits / 32 available
//
```

```
//
// writedata[15:0] - audio data
// writedata[18:16] - ram channel
// writedata[21:19] - velocity
// writedata[25:22] - shift amount
// writedata[29:26] -active channels
// writedata[31:30] - live/playback mode
//

void write_single_sample(const int sample, int channel, int velocity, int
shift, int active_channels, int playback ){
  const int combo = (playback & 0x1) << 30 |(active_channels & 0x4) << 29
|(shift & 0x4) << 25 | (velocity & 0x3) << 19 | (channel &&
 0x3) << 16 | (sample & 0xFFFF);

//  printf("%u\n ", (combo &00000000000001110000000000000000 ) >> 16 );
  audio_arg_t vla;
  vla.audio_sample = combo;
  if (ioctl(audio_driver_fd, AUDIO_WRITE, &vla)) {
      perror("ioctl(AUDIO_WRITE) failed");
      return;
  }
}



void write_full_sample(int num_samples, int* data, int channel, int
velocity, int shift, int active_channels, int playback){
 //prepares velocity
 unsigned int vel_adj = (velocity & 0b1110000) >> 4 ;
  // TODO: samples are 16 bits, so you could fit 2 samples in 1 write
operation
  for(int i = 0; i < num_samples; i++){
    write_single_sample(data[i], channel, vel_adj, shift, active_channels,
playback);
  }
//  printf("%d samples written to the driver", i);
};

int get_octave(int noteIndex){
        int octave_index = 0;

        if(noteIndex <= 31) octave_index = 0;
```

```
        else if(noteIndex <= 55) octave_index = 1;
        else octave_index = 2;

  return octave_index;
}


int main(){

  //set file names:
        char* sample_name[4][3] = {

          {"bassoonC2.wav","bassoonC3.wav","bassoonC4.wav"} ,
          {"cello2C2.wav","cello2C3.wav","cello2C4.wav" } ,
          {"pianoC2.wav","pianoC3.wav", "pianoC4.wav" } ,
          {"synthbrassC2.wav","synthbrassC3.wav","synthbrassC4.wav"}

        };




//MIDI decoding info
  int c_note = 7;       //TODO: adjust so that this is the note for c
  int shift_amount = 0; //number of half steps
  int octave_index = 0; //min of 2 max of 4
  struct usb_midi_packet packet;
  int transferred;
  char keystate[12];

   //for getting audio sample
  char* prefix = "../../res/samples_cut/";
  size_t prefix_len = strlen(prefix);

//user interface
  user_interface_arg_t vla;
  user_interface_props_t props = {(unsigned short) 100, (unsigned char)1,
(unsigned char)1, (unsigned char)1 };
  static const char filename[] = "/dev/user_interface";


//Sequencer controllers
  int* wav_data;
```

```
  int num_samples = 0 ;
  int active_chan[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  int control_velocity[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}
  };


  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}
  };
  int usleep_interval;
  double beat_duration;
  NoteInfo note;

printf("Sequencer Interface Userspace program started\n");

  // find audio driver
  static const char fname[] = "/dev/audio_driver";
  if ( (audio_driver_fd = open(fname, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", fname);
    return -1;
  }


// find user interface driver
  if ( (user_interface_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }

  /* Open the keyboard */
  if ( (midi = openmidi(&endpoint_address)) == NULL ) {
    fprintf(stderr, "Did not find a keyboard\n");
    exit(1);
  }
```

```
  //Initalize sequencer settings
  read_props(&props);



  //escape loop by playing highest note
  while(note.noteIndex != 73 ) {
        if(props.playback == 1){ //playback mode
       //calculate wait interval based on bpm
          beat_duration = 60.0 / props.bpm;
          usleep_interval = (int)((beat_duration) * 1000000);
          int cur_octave = 0 ;

        printf("BPM %u \n", props.bpm);
  for (int i = 0 ; i<16; i++){
                printf("%u\d" , i);
                for(int j = 0; j<4;j++){
                        int cur = control_notes[j][i];
                        int cur_velocity = control_velocity[j][i];
                        int cur_active = active_chan[i];

                        if(cur = !0) {
                        cur_octave = get_octave(cur);


                          //get file name
                          char* selectedTrack =
sample_name[j][octave_index];
                         size_t selectedTrack_len = strlen(selectedTrack);
                         size_t total_len = prefix_len + selectedTrack_len
+ 1; // +1 for the null terminator

                        //assemble file path
                         char* desiredPath = malloc(total_len);
                         strcpy(desiredPath, prefix);
                         strcat(desiredPath, selectedTrack);


                         //Get info for write operation
                         num_samples = read_wav(&wav_data, desiredPath,
0);
                         shift_amount = (cur % 12) - c_note;
```

```
                        //num_samples, sample, channel, velocity, shfit,
active_chan, playback
                        write_full_sample(num_samples, wav_data, j,
cur_velocity, shift_amount, active_chan[i] , 1);
                        free(wav_data);
                        free(desiredPath);
                       }



                                    //write the original value back to the
arrays
                  control_notes[j][i] = cur;
                  control_velocity[j][i] = cur_velocity;
                  active_chan[i] = cur_active;




                } // end of track loop

                usleep(usleep_interval);
        } //end of step loop


        read_props(&props);
        }//end of props.playback ==1

   else { //record mode

     libusb_bulk_transfer(midi , endpoint_address, (unsigned char*)
&packet, sizeof(packet), &transferred , 0);
        if (transferred == sizeof(packet)) {

        //Get UI info and decode MIDI
        read_props(&props);
        note = mapCodeToNote(packet.keycode[1]);


        //Todo: figure out if shift amount is right and adjust for writing
to hw
        shift_amount = note.noteVal - c_note;
```

```
        //adjust octave
        octave_index = get_octave(note.noteIndex);

// printf("BPM: %u, Step: %u, Track: %u, Playback; %u, Note: %d, Speed:
%d\n", props.bpm, props.step, props.track, props.playback, note,
packet.keycode[2]);

//printf("status: %u, kc1: %u, kc2: %u, kc3: %u \n" ,packet.status,
packet.keycode[0],packet.keycode[1],packet.keycodee[2]);


              //get the track and file name
        int track = props.track - 1;
        char* selectedTrack = sample_name[track][octave_index];
        size_t selectedTrack_len = strlen(selectedTrack);
        size_t total_len = prefix_len + selectedTrack_len + 1; // +1 for
the null terminator
        char* desiredPath = malloc(total_len);
        strcpy(desiredPath, prefix);
        strcat(desiredPath, selectedTrack);


        //printf("file to be opened: %s\n",desiredPath);

        int step_on = props.step;

        // write audio file
         num_samples = read_wav(&wav_data, desiredPath, 0);



         if( note.noteIndex != 0 && packet.status != 8){
            //calculate active channels
                int curr_active = 1;
                for (int i = 0; i < track ; i++){
                        curr_active *= 2;
                }
         //write sample
           write_full_sample(num_samples,wav_data, track,
packet.keycode[2], shift_amount, curr_active, 0);
           control_notes[track][step_on] = note.noteIndex;
           control_velocity[track][step_on] = packet.keycode[2];
```

```
            active_chan[step_on] += curr_active;

          }
           else {
           num_samples = 0;

          } //end of if else

           free(wav_data);
           free(desiredPath);


           } // end of if(transferred == sizeof)

              usleep(20000);

       } //end of props.playbacj


   } //end of for(;;;)



    printf("Audio Userspace program terminating\n");
    return 0;
}
```

Process_samples.py

```
"""

* Script used to preprocess audio samples and cut them to 2 seconds
* Team EmbeddedSequencer - Columbia University


"""

import scipy.io
import scipy.signal
from os import listdir
```

```
import numpy as np

dir_original = './samples_original/'
dir_processed = './samples_cut/'

len_procesed = 44100 * 1 # cut length of audio samples to 1s
sr_new = 48000

if __name__ == "__main__":
   path_original_list = listdir(dir_original)
   print(path_original_list)

   for f in path_original_list:
      sr, data = scipy.io.wavfile.read(dir_original + f)
      if sr != 44100:
         print("Sample rate != 44.1kHz, exiting program...")
         raise Exception

      if len(data) < len_procesed:
         print("Samples shorter than desired length detected, exiting
program...")
         raise Exception

      # cut to prespecified length
      data_processed = data[:len_procesed]

      # resample audio
      data_processed = scipy.signal.resample(data_processed, sr_new)
      data_processed = data_processed.astype(np.int16)

      # todo: use a tapering envelope to avoid clicking noise

      # write
      scipy.io.wavfile.write(dir_processed + f, sr_new, data_processed)
```