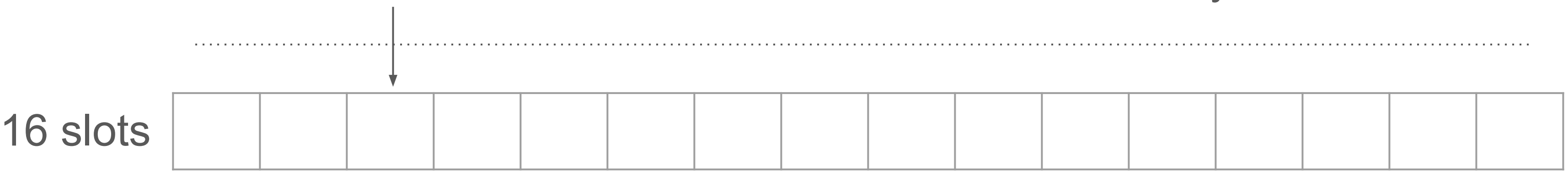
Embedded Sequencer

Brandon Cruz, Adrian Florea & Alexander Ranschaert

Specifications/Desired behavior

2 Modes:

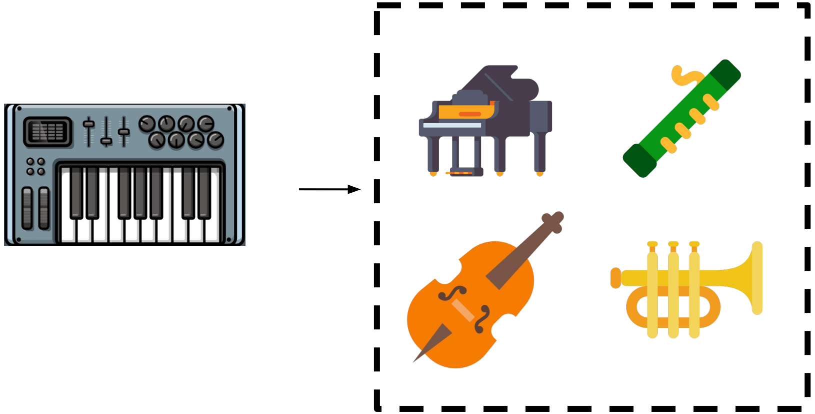
- Record
- Playback; choose BPM

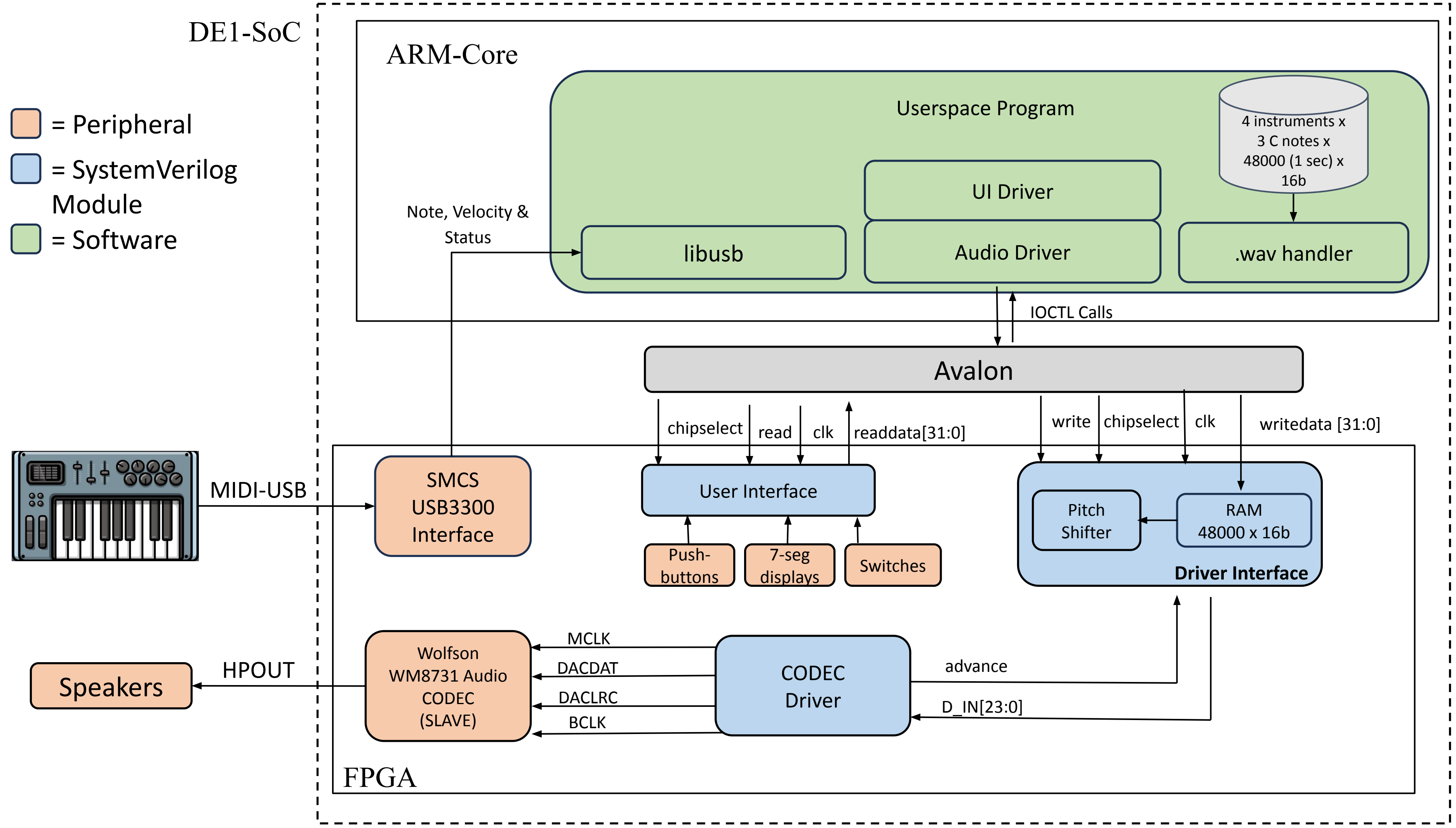


Select slot with button

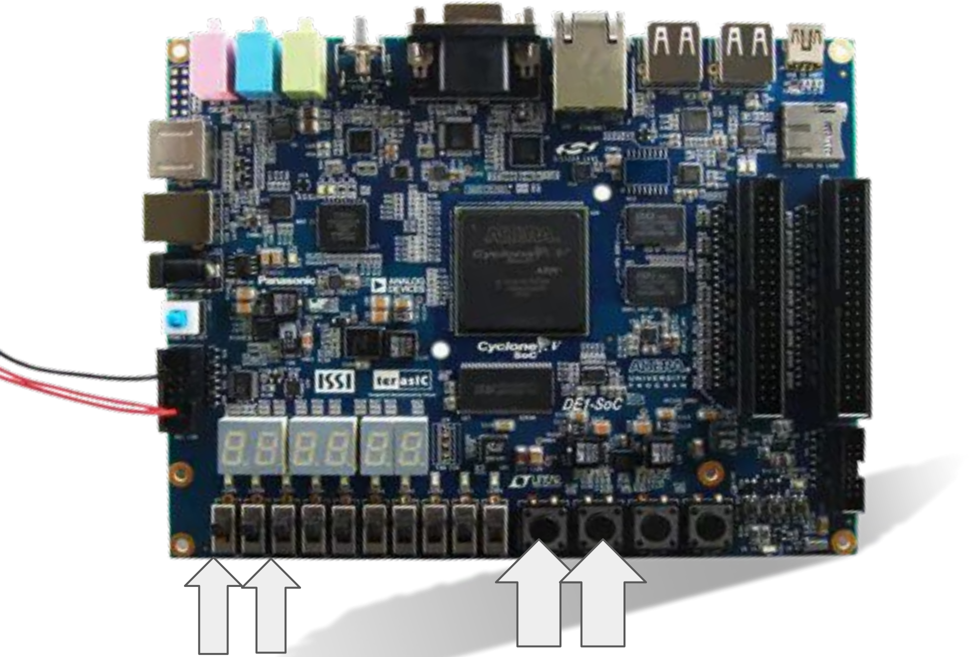
16 slots

4 Instruments;
Controlled with
MIDI keyboard




DE1-SoC
= Peripheral
= SystemVerilog Module
= Software
ARM-Core
Userspace Program
4 instruments x 3 C notes x 48000 (1 sec) x 16b
UI Driver
libusb
Audio Driver
.wav handler
Note, Velocity & Status
IOCTL Calls
Avalon
chipselect
read
clk
readdata[31:0]
write
chipselect
clk
writedata [31:0]
MIDI-USB
SMCS USB3300 Interface
User Interface
Push-buttons
7-seg displays
Switches
Pitch Shifter
RAM 48000 x 16b
Driver Interface
HPOUT
Speakers
Wolfson WM8731 Audio CODEC (SLAVE)
MCLK
DACDAT
DACLRC
BCLK
CODEC Driver
advance
D_IN[23:0]
FPGA

User Interface



playback

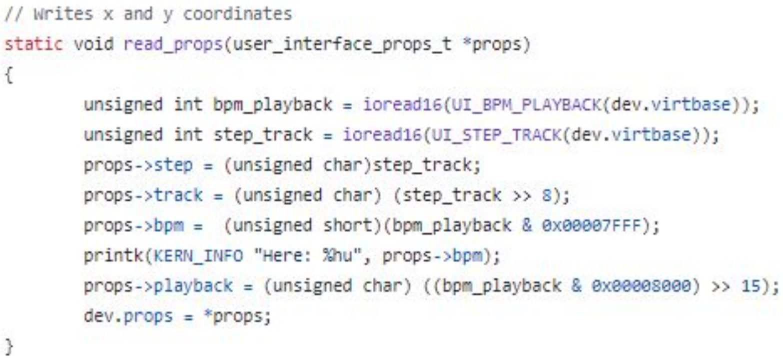
bpm

change step

change track/bpm

- Memory mapped messages to userspace:

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

8bits    8bits 1bit   15bits
```

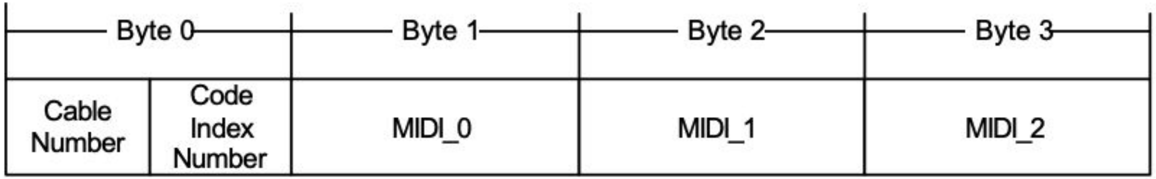


```c
// Writes x and y coordinates
static void read_props(user_interface_props_t *props)
{
        unsigned int bpm_playback = ioread16(UI_BPM_PLAYBACK(dev.virtbase));
        unsigned int step_track = ioread16(UI_STEP_TRACK(dev.virtbase));
        props->step = (unsigned char)step_track;
        props->track = (unsigned char) (step_track >> 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;
}
```

Decoding USB-MIDI

Handled via USB Bulk Transfer:

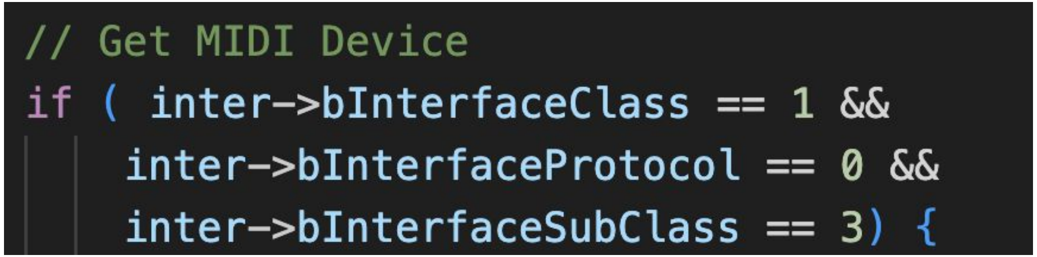
USB-MIDI Packets



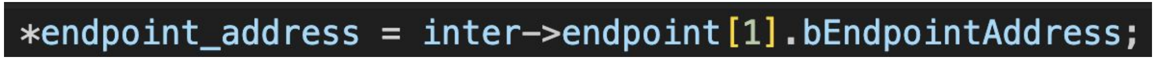
Byte 0		Byte 1	Byte 2	Byte 3
Cable Number	Code Index Number	MIDI_0	MIDI_1	MIDI_2

Note on or off   Note velocity   Key Number   Zero padded

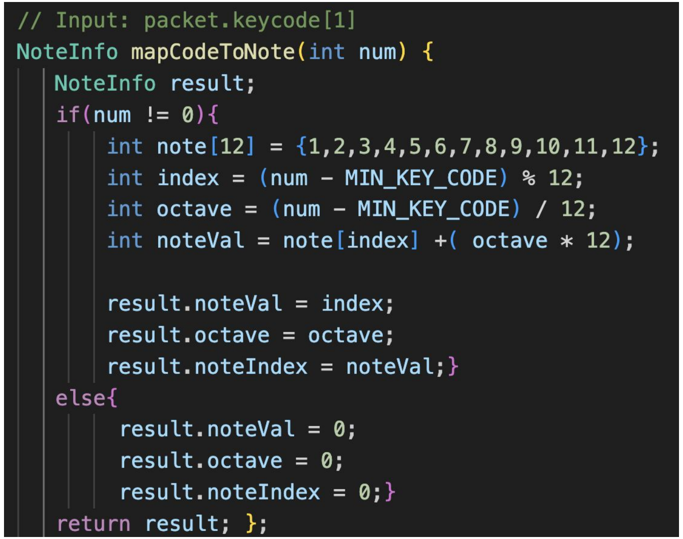
Libusb Device Handle:



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



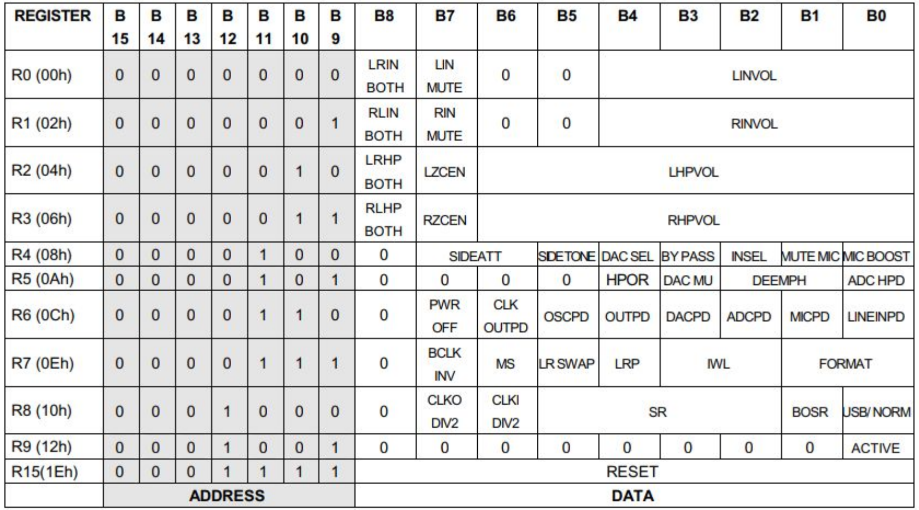
```
*endpoint_address = inter->endpoint[1].bEndpointAddress;
```



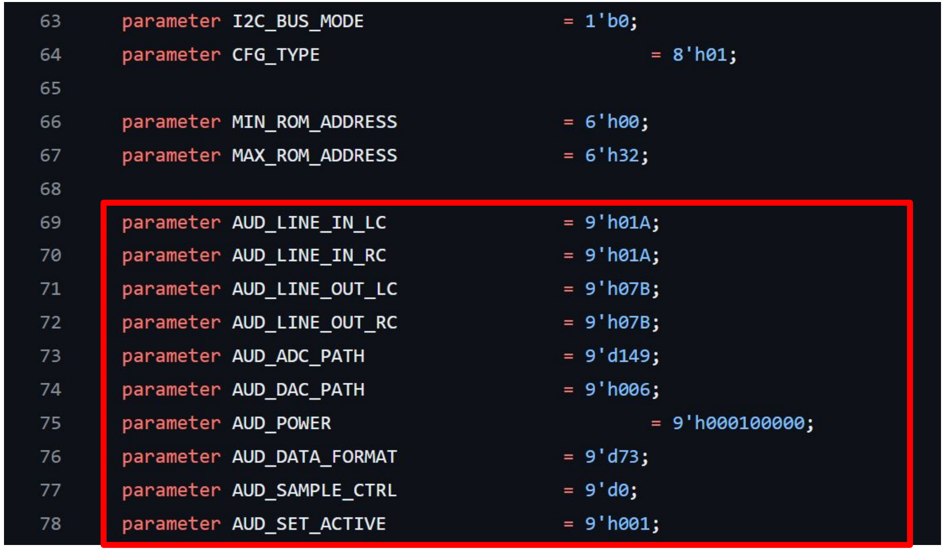
```
// Input: packet.keycode[1]
NoteInfo mapCodeToNote(int num) {
   NoteInfo result;
   if(num != 0){
       int note[12] = {1,2,3,4,5,6,7,8,9,10,11,12};
       int index = (num - MIN_KEY_CODE) % 12;
       int octave = (num - MIN_KEY_CODE) / 12;
       int noteVal = note[index] +( octave * 12);

       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



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								



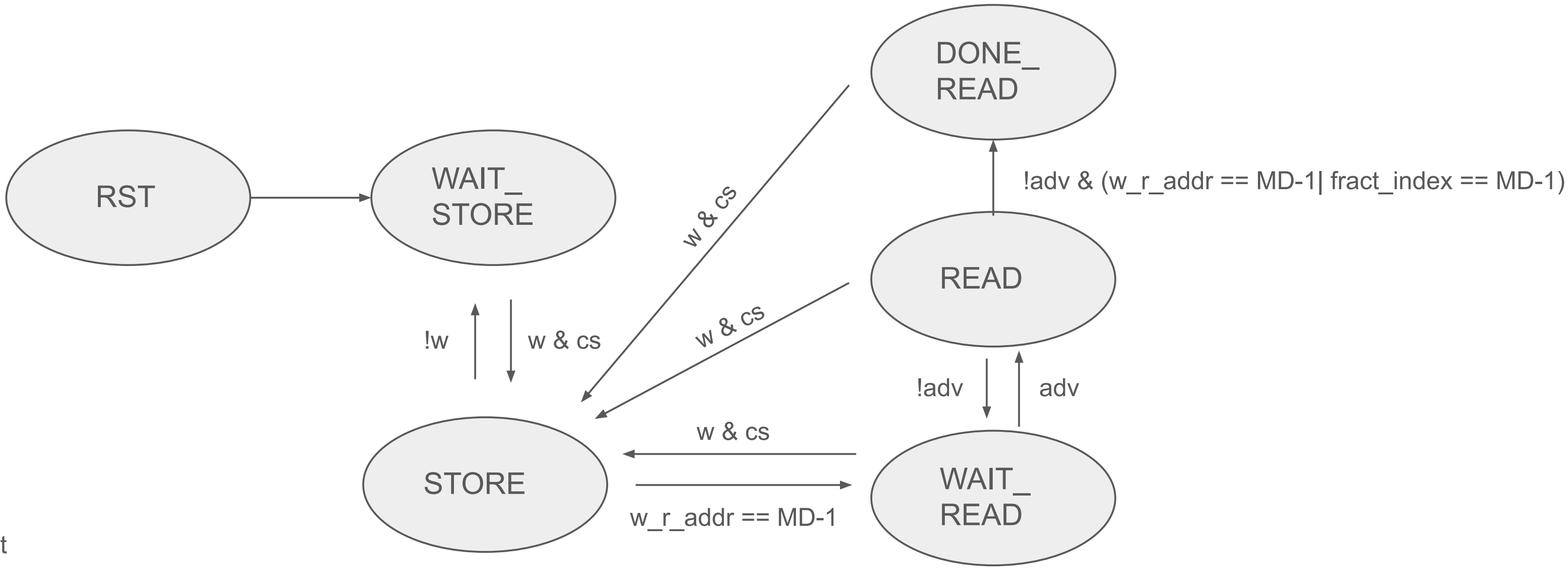
```
    parameter I2C_BUS_MODE                  = 1'b0;
    parameter CFG_TYPE                              = 8'h01;

    parameter MIN_ROM_ADDRESS               = 6'h00;
    parameter MAX_ROM_ADDRESS               = 6'h32;

    parameter AUD_LINE_IN_LC                = 9'h01A;
    parameter AUD_LINE_IN_RC                = 9'h01A;
    parameter AUD_LINE_OUT_LC               = 9'h07B;
    parameter AUD_LINE_OUT_RC               = 9'h07B;
    parameter AUD_ADC_PATH                  = 9'd149;
    parameter AUD_DAC_PATH                  = 9'h006;
    parameter AUD_POWER                             = 9'h000100000;
    parameter AUD_DATA_FORMAT               = 9'd73;
    parameter AUD_SAMPLE_CTRL               = 9'd0;
    parameter AUD_SET_ACTIVE                = 9'h001;
```

(driver we used was a wrapper around preconfigured Intel IP)

Driver Interface (Moore FSM)

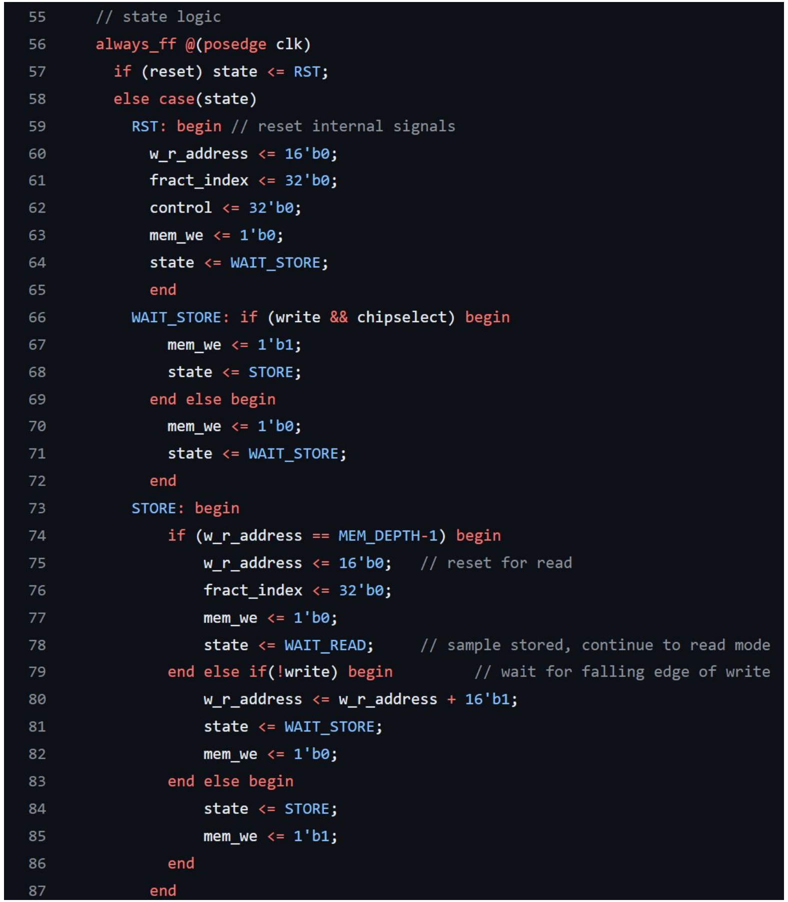
1. Store sample from userspace to RAM

2. Shift Pitch & Pass samples to CODEC

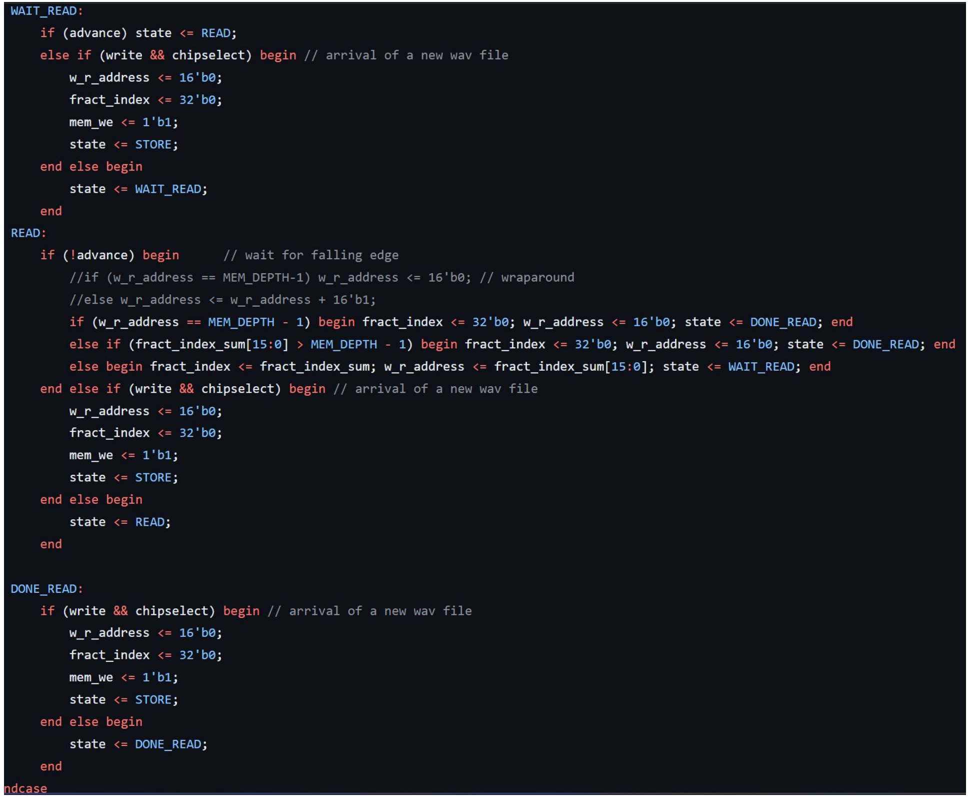


w : write
cs : chipselect
adv : advance
MD : Memory Depth (48000)

Driver Interface (Moore FSM, State Transitions)

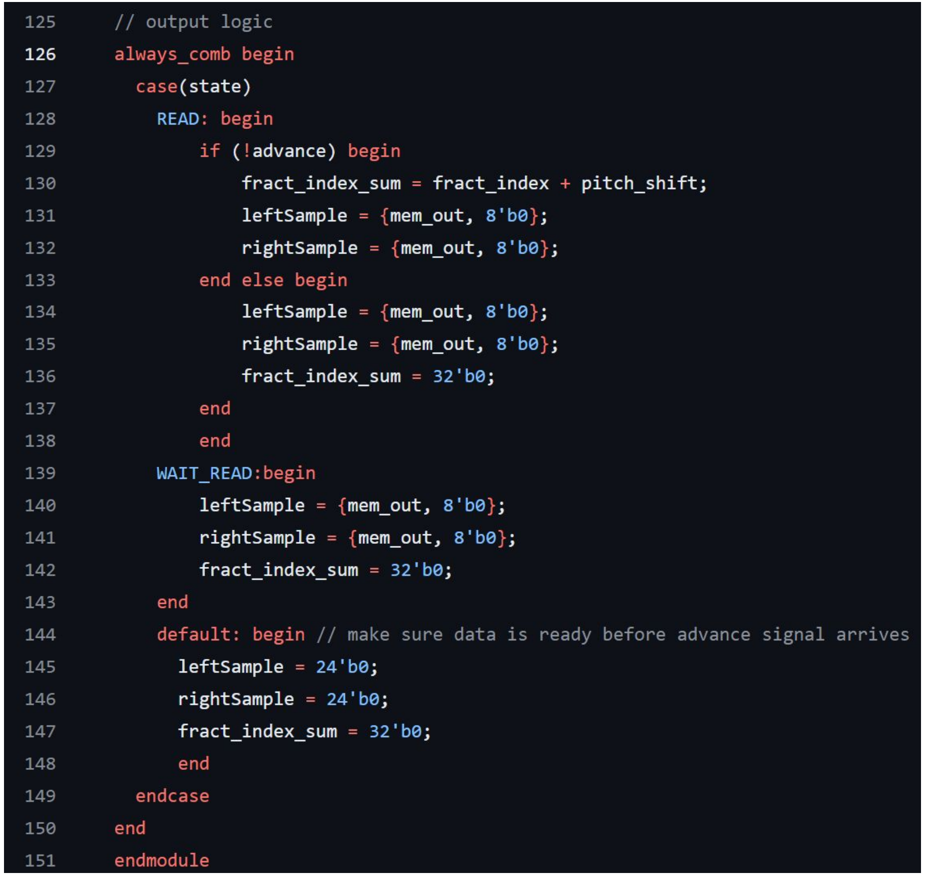


```
  // state logic
  always_ff @(posedge clk)
    if (reset) state <= RST;
    else case(state)
      RST: begin // reset internal signals
        w_r_address <= 16'b0;
        fract_index <= 32'b0;
        control <= 32'b0;
        mem_we <= 1'b0;
        state <= WAIT_STORE;
        end
      WAIT_STORE: 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
ndcase
```

Driver Interface (Moore FSM, Output Logic)



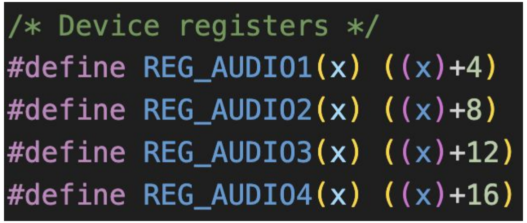
```
// output logic
always_comb begin
  case(state)
    READ: begin
        if (!advance) begin
            fract_index_sum = fract_index + pitch_shift;
            leftSample = {mem_out, 8'b0};
            rightSample = {mem_out, 8'b0};
        end else begin
            leftSample = {mem_out, 8'b0};
            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

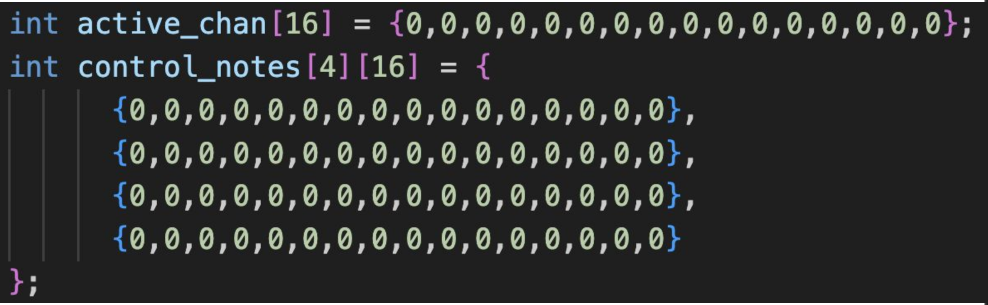
- Memory mapped messages:

[playbackmode	active channels	pitch_shift	note_velocity	channel	audio_sample]
1 bit	4 bits	4 bits	3 bits	2 bits	16 bits

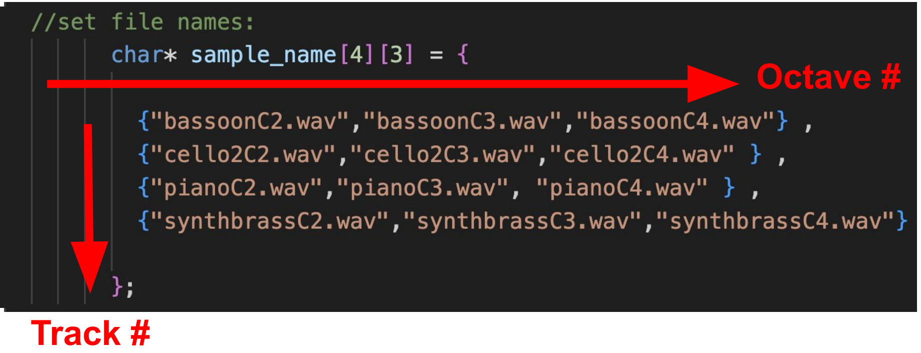
- Audio driver with 4 device registers (1 per track) for memory mapped write



```
/* 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}
};
```

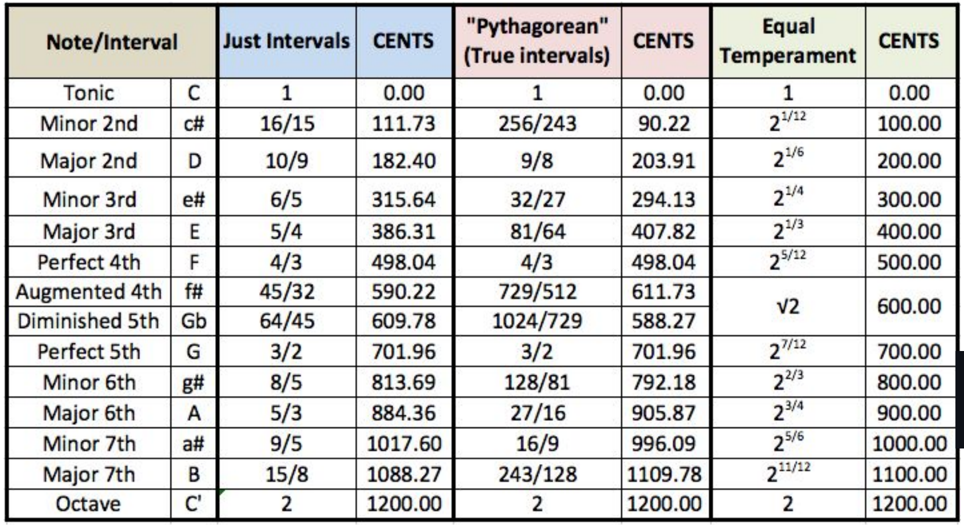


```
//set file names:
        char* sample_name[4][3] = {

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

        };
```

Octave #

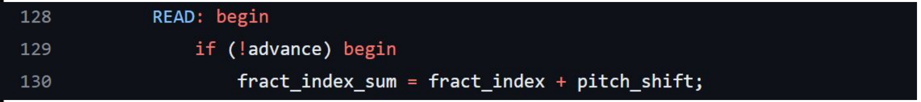
Track #

Pitch Shifter

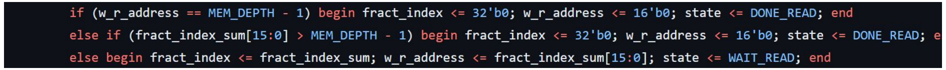


Note/Interval		Just Intervals	CENTS	"Pythagorean" (True intervals)	CENTS	Equal Temperament	CENTS
Tonic	C	1	0.00	1	0.00	1	0.00
Minor 2nd	c#	16/15	111.73	256/243	90.22	$2^{1/12}$	100.00
Major 2nd	D	10/9	182.40	9/8	203.91	$2^{1/6}$	200.00
Minor 3rd	e#	6/5	315.64	32/27	294.13	$2^{1/4}$	300.00
Major 3rd	E	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

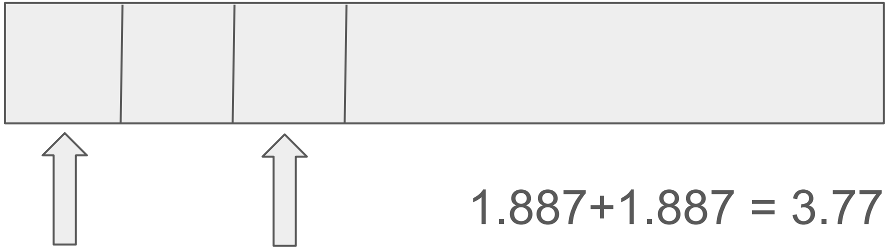
- Just intonation vs Equal Temperament
- Implemented using fixed point numbers and skipping samples



```
        READ: begin
            if (!advance) begin
                fract_index_sum = fract_index + pitch_shift;
```

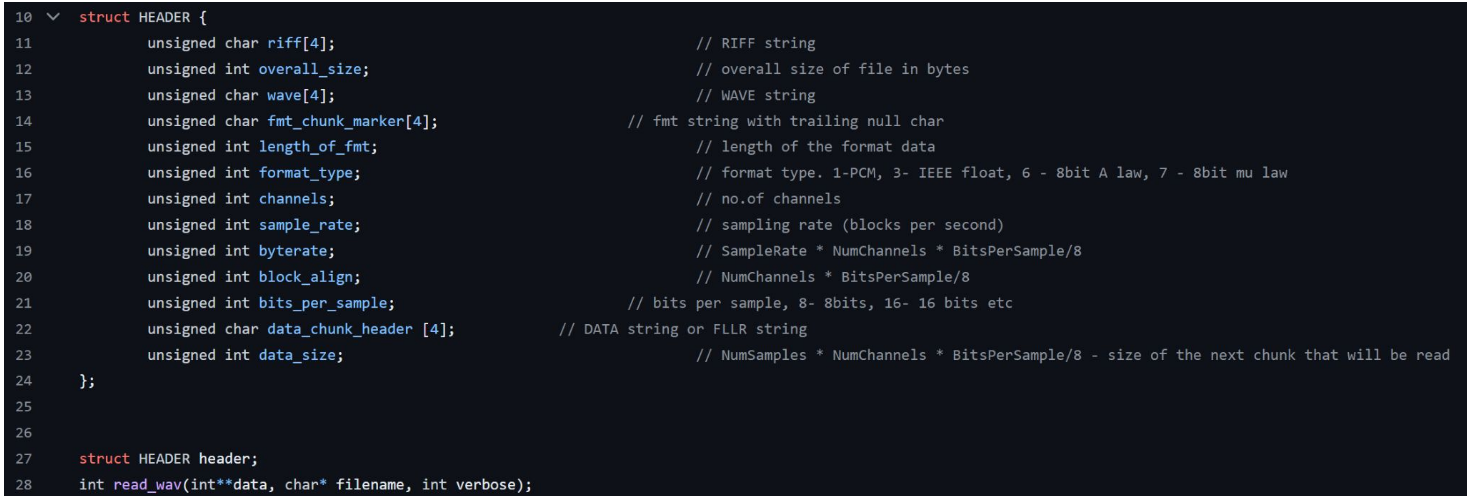


```
    if (w_r_address == MEM_DEPTH - 1) begin fract_index <= 32'b0; w_r_address <= 16'b0; state <= DONE_READ; end
    else if (fract_index_sum[15:0] > MEM_DEPTH - 1) begin fract_index <= 32'b0; w_r_address <= 16'b0; state <= DONE_READ; e
    else begin fract_index <= fract_index_sum; w_r_address <= fract_index_sum[15:0]; state <= WAIT_READ; end
```



1.887+1.887 = 3.77 = 3

wav_handler.c



```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 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);
```

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.