No Caller ID: Design Document

Because sometimes it's good to be a little incognito.

The secret agents behind the next big thing in spy-tech:

Noah Silverstein (nis2116), Max Acebal (mra2180), Marian Abuhazi (ma4107), and Rebekah Kim (rmk2160).

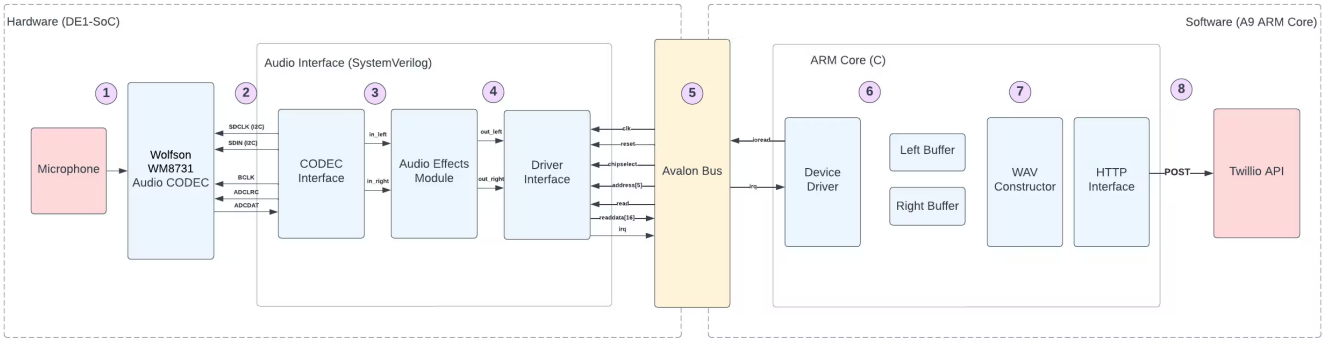
Abstract

Team No Caller ID is proud to present our design for incognito phone calls. By sampling a microphone input through the *Mic in* port on the *Terasic FPGA* and converting the audio signal to a digital format, we will apply pitch-shift effects to the audio through software. This distorted audio will be saved as *.wav* file and sent as a pre-recorded phone using the *Twilio* API.

The real-time audio effect will allow you to remain completely anonymous and can be used to throw somebody off your scent, allow for more privacy, or even have some fun.

System Block Diagram

The full system from microphone input to API call output is detailed in the following system block diagram. All relevant connections are listed between functional units. The left side of the diagram indicates the FPGA hardware, and the right side indicates the ARM Core software. The Avalon Bus connects between the two in the center. In the following section, we will walk through, in detail, each module and the full data flow through our system.



Modules and Data Flow

Below, we will discuss each data transition depicted in the system block diagram. Each number corresponds to the label within the system diagram and the data flow and modification is described accordingly.

1. Microphone to Board

The DE1-SoC houses the Wolfson WM8731 audio CODEC, which in turn, contains an Analog to digital converter (ADC) and an associated mic in port. An external, unpowered, microphone will be connected to this port as our audio source.

2. Audio CODEC to CODEC Interface

The Wolfson audio CODEC is configured via an I2C connection to our Audio Interface. The CODEC Interface (a small controller written in SystemVerilog) puts the CODEC into slave mode and configures its operations via a series of programmable registers set via the I2C interface (the SDCLCK and SDIN lines). The CODEC contains the following 16 registers mapped to the listed addresses.

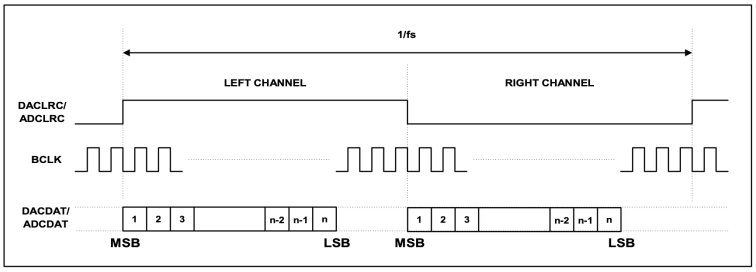
REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	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								

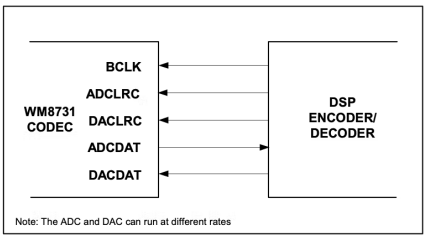
Our CODEC Interface has a simple I2C serial interface which loads the values listed above into the CODEC configuration registers. The relevant registers to set are R4 (bit 2 = 1 to enable mic In) and R7 (bit 1:0 = 01 for MSB-First, left-justified, bit 3:2 = 11 for 32 bits, bit 6 = 0 to enable slave mode). The following values are determined via the CODEC manual, and the relevant configuration options are listed below.

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000111 Digital Audio Interface Format	1:0	FORMAT[1:0]	10	Audio Data Format Select 11 = DSP Mode, frame sync + 2 data packed words 10 = I^2S Format, MSB-First left-1 justified 01 = MSB-First, left justified 00 = MSB-First, right justified
	3:2	IWL[1:0]	10	Input Audio Data Bit Length Select 11 = 32 bits 10 = 24 bits 01 = 20 bits 00 = 16 bits
	4	LRP	0	DACLRC phase control (in left, right or I^2S modes) 1 = Right Channel DAC data when DACLRC high 0 = Right Channel DAC data when DACLRC low (opposite phasing in I^2S mode) or DSP mode A/B select (in DSP mode only) 1 = MSB is available on 2nd BCLK rising edge after DACLRC rising edge 0 = MSB is available on 1st BCLK rising edge after DACLRC rising edge
	5	LRSWAP	0	DAC Left Right Clock Swap 1 = Right Channel DAC Data Left 0 = Right Channel DAC Data Right
	6	MS	0	Master Slave Mode Control 1 = Enable Master Mode 0 = Enable Slave Mode
	7	BCLKINV	0	Bit Clock Invert 1 = Invert BCLK 0 = Don't invert BCLK

0000100 Analogue Audio Path Control	0	MICBOOST	0	Microphone Input Level Boost 1 = Enable Boost 0 = Disable Boost
	1	MUTEMIC	1	Mic Input Mute to ADC 1 = Enable Mute 0 = Disable Mute
	2	INSEL	0	Microphone/Line Input Select to ADC 1 = Microphone Input Select to ADC 0 = Line Input Select to ADC
	3	BYPASS	1	Bypass Switch 1 = Enable Bypass 0 = Disable Bypass
	4	DACSEL	0	DAC Select 1 =Select DAC 0 = Don't select DAC
	5	SIDETONE	0	Side Tone Switch 1 = Enable Side Tone 0 = Disable Side Tone
	7:6	SIDEATT[1:0]	00	Side Tone Attenuation 11 = -15dB 10 = -12dB 01 = -9dB 00 = -6dB

Once configured, the CODEC reports data, when requested, via a simple serial interface to our CODEC Interface. In short, the CODEC Interface pulls ADCLRCK high or low to receive 32 bits of either right or left channel data (ADCDAT) with the MSB sent first. With a 50MHz master clock, 48kHz sampling rate is trivially created via a clock divisor. This data will be passed on to the Audio Effects Module.





3. CODEC Interface to Audio Effects Module

The CODEC Interface has two 32 bit registers to store a single left sample and a single right sample. As it pulses the BCLK signal and receives data from the CODEC, it writes into these registers. Then, the CODEC Interface, passes the left stereo data via in_left and right stereo data via in_right to the Audio Effects Module where they are stored in circular buffers.

4. Audio Effects Module to Driver Interface

The Audio Interface implements the octave shifting algorithm described later in the Algorithm's section, resulting in two 32 bit sample registers (left and right) which are updated at the same rate that new values are read in. They are passed to the driver interface via out_left and out_right wires.

5. Driver Interface to ARM Core

The Driver Interface reads left and right samples from the Audio Effects Module. At the same rate as the ADC samples (48kHz), it raises the irq (interrupt request) telling the ARM Core that it has a sample ready to read. Then, the device driver will handle the interrupt via reading from the Driver Interface via the Avalon bus. The Driver Interface will send 64 bits of data over the readdata line of the Avalon interface.

6. Device Driver to User Memory Space

The device driver on the software side will receive data via ioctl calls (ioread), prompted via an interrupt raised by the driver interface over the Avalon bus, from a device driver running on the ARM Core. Individual 32 bit samples, which have been octave shifted, will be passed across the Avalon bus, and the device driver will send the data from kernel space to user space so that it can be processed.

7. Sample Accumulation and File Formatting

The C program will accumulate audio samples in two arrays (left and right channels).

Once a certain number of samples (length of recording) is stored, the data will be formatted as a .wav file as described in the Algorithm's section below.

8. API Request

The .wav file will be uploaded to the Twillio API via a POST HTTP request. Then, it will be sent as a phone call to a designated number via another HTTP POST request. These steps are detailed under the Algorithm's section below.

Algorithms

Audio Effects Module

Introduction

Pitch-shifting is the process of altering the pitch of an audio signal without reducing nor increasing its duration. Pitch-shifting can be performed in the frequency domain using the Fast Fourier Transform (FFT) algorithm, manipulating the frequencies of the signal, and reverting to a time-domain signal using an inverse FFT. However, this method is mathematically intensive and complicated to implement in hardware.

In this project we will apply audio effects on a voice signal captured on the mic line to the DE1-SOC board. We will offer two types of audio effects: the ability to shift the signal to a higher pitch (”Chipmunk” voice), and to a lower pitch (”Darth Vader” voice) in real time.

We have implemented the algorithm described below to perform the pitch-shifting entirely in the time domain. The algorithm has been adapted from PitchShifter by Sean Carroll, Gulnar Mirza, and James Talmage from Cornell University.

Overview

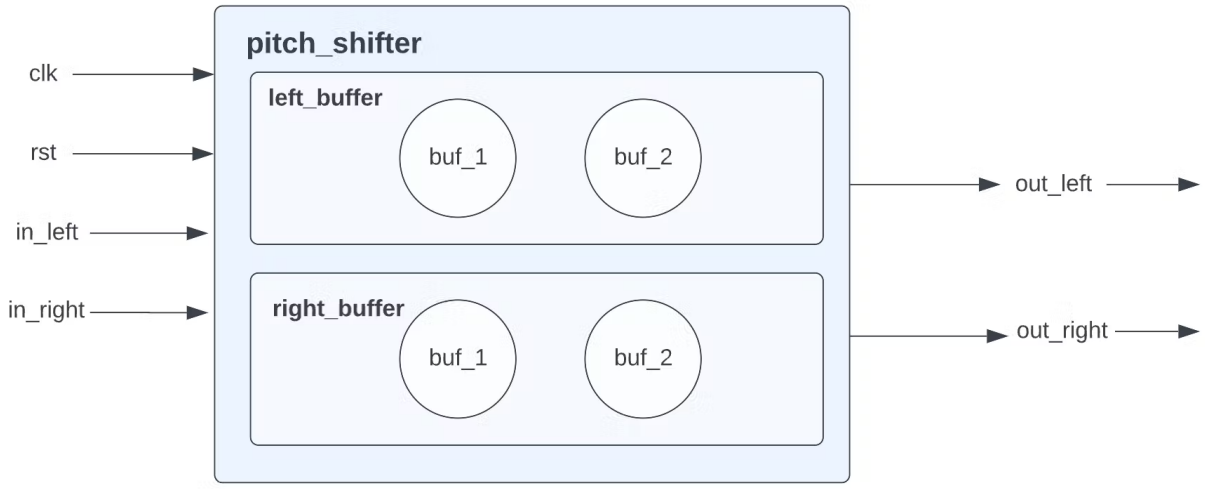
The Audio Effects Module consists of a child module, the *pitch-shifter*. The *pitch-shifter* module shifts the signal to the desired octave. The pitching of the signal is determined by a shift factor, which we have pre-set to 0.8 for a “Darth Vader” effect, and 1.65 for the “Chipmunk” alterations. These factors have been determined from extensive simulation.

Architecture

The Audio Effects Module takes in the following signals: a clock (*clk*), a reset (*rst*), a 32-bit left speaker data sample (*in_left*), and a 32-bit right speaker data sample (*in_right*). The module outputs 2 signals: *out_left* and *out_right*, which correspond to the outputs of the shifter.

The *in_left* and *in_right* data samples are passed into *pitch_shifter* from the CODEC interface.

Audio Effects Module



Algorithm

On every rising edge of *clk*, *pitch_shifter* reads the *in_left* sample and passes the value read to the *left_buffer* module. *Left_buffer* is implemented using two circular buffers of length 1024. *Buf_1* and *buf_2* have two signals each called *w* and

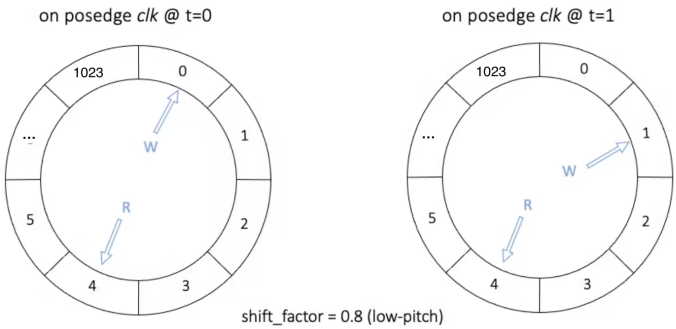
r that serve as pointers for writing and reading data. All pointers are initialized to 0 upon *rst*.

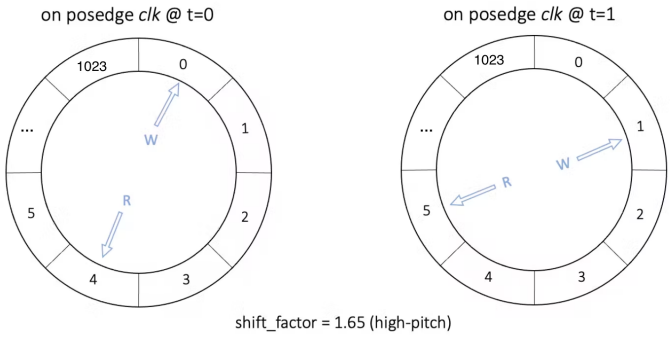
Writing to buffers

Once the sample is passed into *left_buffer*, it is written to *buf_1* at address *w_1*, then *w_1* is incremented by 1. Next, this same sample *in_left* is written to *buf_2* at *w_2*, a location 180° from *w_1*. Finally, *w_2* is advanced by 1. The same process is repeated for *in_right* data with its respective *right_buffer*.

Reading from buffers

Now that the new samples are written to the left and right buffers, they need to be read at the pitch-shifted location. This is done by offsetting the read pointer by the integer portion of the shift-factor. Hence, if the shift-factor is 0.8, we will read at a location 0 cells away from the current read pointer. However, if the shift-factor is 1.65 we will read from a location 1 cell away. This resembles the essence of pitch-shifting as it “skips” some samples for low-pitch shifting and “repeats” other samples for the high-pitch shifting.





Averaging values read from buf_1 and buf_2

A dual-buffer design allows us to improve the quality of the output audio signal. Essentially, by having a second buffer whose read pointer is always 180° shifted from the main buffer we minimize the likelihood that the buffer will encounter and output an “older” sample that has already been written to the bus. This eliminates “clicking” sounds.

MATLAB Simulation

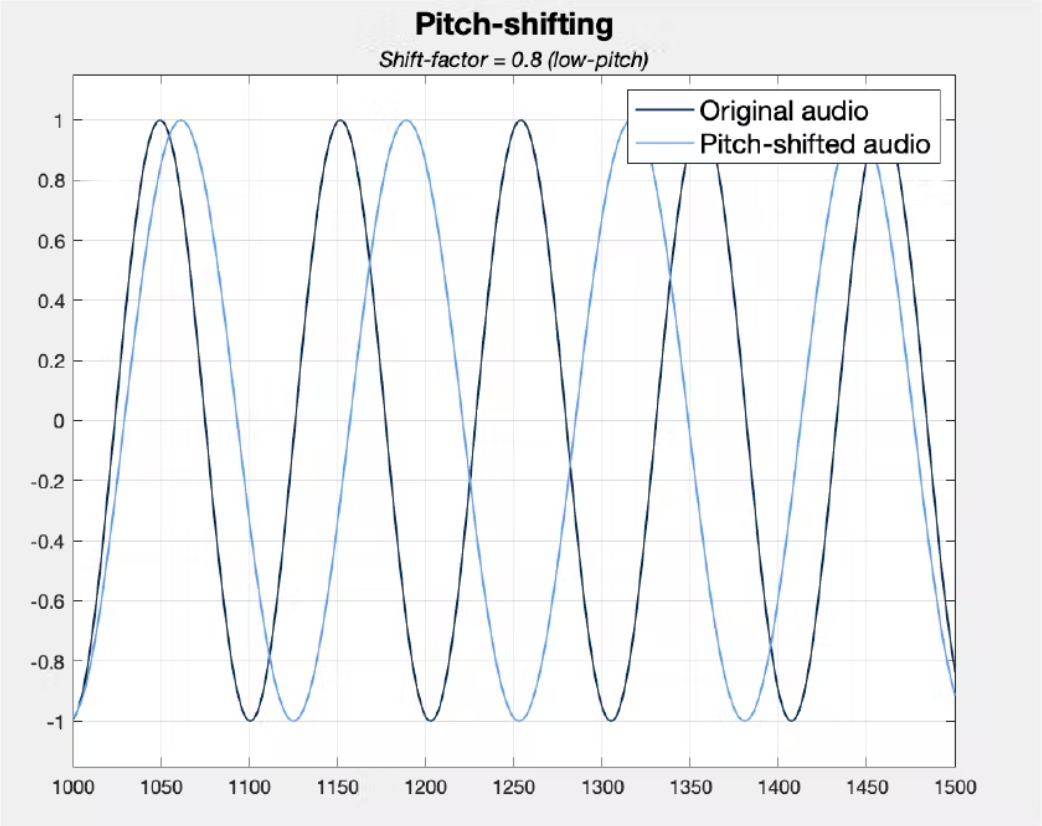
```
% ------------- SETUP ------------- %
% Circular buffer length
I = 1024;

% Shifting factor
```

```
% 0.85 for low pitch
% 1 for normal voice
% 1.65 for high pitch
shift_factor = 0.8;

% Read .wav file
% Test audio file is a recitation of The Gettysburg Address
filename = 'gettysburg.wav';
% [samples,fs] = audioread(filename);

% Test sine wave to better visualize effect of shifting & filtering
fs = 45000;
x = linspace(0,1,fs);
samples = sin(2*pi*440*x);

% ------------- BUFFERS AND POINTERS ------------- %
% Create 2 buffers for left data, and two buffers for right data
% Initialize buffers to zero
l_buf_1 = zeros(1, l);
l_buf_2 = zeros(1, l);
r_buf_1 = zeros(1, l);
r_buf_2 = zeros(1, l);

% Output array of shifted samples
% In hardware implementation, samples will be streamed to Avalon bus

% Output
out_left = zeros(length(samples),1);
out_right = zeros(length(samples),1);

% Read and write pointers for l_buf_1, l_buf_2, r_buf_1, and r_buf_2
% Initialize w_1 to 1. MATLAB arrays start at 1.
% Initialize w_2 to 180 deg shifted cell.

% l_buf write
l_w_1 = 1;
l_w_2 = floor(l/2);

% l_buf read
l_r_1 = 1;
l_r_2 = 1;

% r_buf write
r_w_1 = 1;
r_w_2 = floor(l/2);

% r_buf read
r_r_1 = 1;
r_r_2 = 1;
```

```
% Indexing for read pointers
l_i_1= 0;
l_i_2= 0;
r_i_1= 0;
r_i_2= 0;

% ------------- ALGORITHM ------------- %
% Read one sample at a time from the array of samples
% Simulate reading one sample from a register in hardware
for i=1:length(samples)

    % LEFT BUFFER
    % Add the sample to l_buf_1 at address l_w_1
    % Advance l_w_1
    l_buf_1(l_w_1) = samples(i);
    l_w_1 = mod(i, l) + 1;

    % Add the sample to l_buf_2
    % Advance l_w_2
    l_buf_2(l_w_2) = samples(i);
    l_w_2 = mod(i+floor(l/2), l) + 1;

    % Read from l_buf_1 at address l_r_1
    l_i_1= l_i_1 + shift_factor;
    l_r_1 = mod(floor(l_i_1), l) + 1;

    % Read from l_buf_2 at address l_r_2
    l_i_2= l_i_2 + shift_factor;
    l_r_2 = mod(floor(l_i_2), l) + 1;


    % RIGHT BUFFER
    % Add the sample to l_buf_1 at address l_w_1
    % Advance l_w_1
    r_buf_1(r_w_1) = samples(i);
    r_w_1 = mod(i, l) + 1;

    % Add the sample to l_buf_2
    % Advance l_w_2
    r_buf_2(r_w_2) = samples(i);
    r_w_2 = mod(i + floor(l/2), l) + 1;

    % Read from l_buf_1 at address l_r_1
    r_i_1= r_i_1 + shift_factor;
    r_r_1 = mod(floor(r_i_1), l) + 1;

    % Read from l_buf_2 at address l_r_2
    r_i_2= r_i_2 + shift_factor;
```

```
    r_r_2 = mod(floor(r_i_2), l) + 1;

    out_left(i) = 0.5*(l_buf_1(l_r_1) + l_buf_2(l_r_2));
    out_right(i) = 0.5*(r_buf_1(r_r_1) + r_buf_2(r_r_2));

end

% ------------- WAVEFORM VIZUALIZATION & SOUND------------- %
% Listen to the original audio
% sound(samples, fs);

% ------------- VISUALIZE PITCH-SHIFT ------------- %
% Filter only in the human voice range using 4th order Butterworth filter
fc = 3500;
[b,a] = butter(4, fc/(fs/2));
out_filtered = filter(b, a, out_left);

% Plot the original audio waveform
figure
plot(samples, 'LineWidth', 1, 'Color', '#0b2852');
hold on;

% Plot the time-shifted, filtered audio waveform
plot(out_filtered, 'LineWidth', 1, 'Color', '#629df5')

% Style plot
[t,s] = title("Pitch-shifting", 'Shift-factor = 0.8 (low-pitch)');
grid;
t.FontSize = 16;
s.FontAngle = 'italic';
lgd = legend('Original audio','Pitch-shifted audio');
lgd.FontSize = 14;

% Visualize a given domain of the sine wave
xlim([1000, 1500]);
ylim([-1.15, 1.15]);
```

Simulation results



WAV File Construction

To generate a WAV file we must take the streamed input and write it to a file with the correct WAV file headers. From there, we will write the unsigned integer input to the file line by line. Below is a standard example for how this code will be written to construct the WAV file from an array of samples (source).

```
/* make_wav.c
 * Creates a WAV file from an array of ints.
 * Output is monophonic, signed 16-bit samples
 * copyright
 * Fri Jun 18 16:36:23 PDT 2010 Kevin Karplus
 * Creative Commons license Attribution-NonCommercial
 *  http://creativecommons.org/licenses/by-nc/3.0/
 */

#include <stdio.h>
#include <assert.h>

#include "make_wav.h"
```

```
void write_little_endian(unsigned int word, int num_bytes, FILE *wav_file)
{
   unsigned buf;
   while(num_bytes>0)
   {   buf = word & 0xff;
      fwrite(&buf, 1,1, wav_file);
      num_bytes--;
   word >>= 8;
   }
}

/* information about the WAV file format from
   http://ccrma.stanford.edu/courses/422/projects/WaveFormat/
 */

void write_wav(char * filename, unsigned long num_samples, short int * data, int s_rate)
{
   FILE* wav_file;
   unsigned int sample_rate;
   unsigned int num_channels;
   unsigned int bytes_per_sample;
   unsigned int byte_rate;
   unsigned long i;    /* counter for samples */

   num_channels = 1;   /* monoaural */
   bytes_per_sample = 2;

   if (s_rate<=0) sample_rate = 44100;
   else sample_rate = (unsigned int) s_rate;

   byte_rate = sample_rate*num_channels*bytes_per_sample;

   wav_file = fopen(filename, "w");
   assert(wav_file);   /* make sure it opened */

   /* write RIFF header */
   fwrite("RIFF", 1, 4, wav_file);
   write_little_endian(36 + bytes_per_sample* num_samples*num_channels, 4, wav_file);
   fwrite("WAVE", 1, 4, wav_file);

   /* write fmt  subchunk */
   fwrite("fmt ", 1, 4, wav_file);
   write_little_endian(16, 4, wav_file);   /* SubChunk1Size is 16 */
   write_little_endian(1, 2, wav_file);    /* PCM is format 1 */
   write_little_endian(num_channels, 2, wav_file);
   write_little_endian(sample_rate, 4, wav_file);
   write_little_endian(byte_rate, 4, wav_file);
   write_little_endian(num_channels*bytes_per_sample, 2, wav_file);  /* block align */
   write_little_endian(8*bytes_per_sample, 2, wav_file);  /* bits/sample */
```

```
    /* write data subchunk */
    fwrite("data", 1, 4, wav_file);
    write_little_endian(bytes_per_sample* num_samples*num_channels, 4, wav_file);
    for (i=0; i< num_samples; i++)
    {   write_little_endian((unsigned int)(data[i]),bytes_per_sample, wav_file);
    }

    fclose(wav_file);
}
```

For our purposes our sample rate will be **48 kHz**, we will have **2 channels** (left and right), we will have **8 bytes (32 bits)** per sample**.** This is because we will be using our 32 bit input modified input signal buffer.

API Access

In order to send audio files as a phone call we need to use an API that allows WAV files to be exported. **Twilio** allows us to do just this by uploading our saved WAV file as an asset on their static file hosting service. From there, we can make a call playing the saved audio by using the Twilio's programmable voice functionality.

Uploading Assets:

To upload our saved asset we can use the Twilio command line plugin via a simple bash script which look like

```
#!/bin/bash
twillio assets:upload path/to/file
```

Making the Call:

We can use curl to use Twilio's REST API to authenticate and create a call object. This will make a phone call with the uploaded distorted voice .wav file to the specified phone number.

```
curl -X POST "https://api.twilio.com/2010-04-01/Accounts/$TWILIO_ACCOUNT_SID/Calls.json" \
--data-urlencode "Url=http://demo.twilio.com/docs/voice.xml" \
--data-urlencode "To=+14155551212" \
--data-urlencode "From=+15017122661" \
-u $TWILIO_ACCOUNT_SID:$TWILIO_AUTH_TOKEN
```