AUDIO CLEANER

60 Hz Hum Removal via FFT — Written in C

PRD · FSPEC · DEVSPEC · TESTING · BENCHMARKING

Project	audio_cleaner
Language	C (C99)
Platform	macOS — Apple Silicon / Intel x86_64
Version	1.0.0
Date	Sat May 23 2026

1. Product Requirements Document (PRD)

1.1 Problem Statement

When you record audio in a real environment — a phone memo, a laptop microphone, anything near a power source — you almost always pick up electrical hum at 60 Hz. It comes from mains power interference, and it doesn't just sit at 60 Hz. It shows up at harmonics too: 120, 180, 240, 300 Hz. On its own, a single recording might sound fine, but when you look at it in a spectrum analyser you can see those spikes clearly, and you can hear the hum once you know it's there.

The problem is that most tools for fixing this are either huge (FFmpeg, SoX) or require a GUI (Audacity). There wasn't a simple, self-contained C program that does exactly this one job and nothing else. That's what this project is.

1.2 Vision

A single C source file that anyone can compile with one command on a Mac terminal, feed a WAV file, and get a cleaned version back. No libraries, no install steps, no configuration files. Just: compile, run, done.

1.3 Connection to Echolocation

The original idea behind this project was echolocation — how bats and sonar systems emit a sound pulse and analyse what comes back to figure out distance and shape. The core tool for that is FFT: you convert a signal to the frequency domain, look at what frequencies are present (or missing, or shifted), and draw conclusions about the environment.

This project uses that same tool — FFT — but applies it to a different problem: instead of analysing echo returns, it analyses a voice recording and removes specific unwanted frequencies. The bandpass_filter() function does essentially the same thing a sonar signal processor would do to isolate a target frequency range. The shift from echolocation to audio cleaning was a scope decision, not a fundamental change in approach.

1.4 Target Users

User	Need
CS/EE Student	Learn how FFT works on real audio — everything is in one readable file
Audio Engineer	Quick hum removal from a voice recording without opening a heavy DAW
Developer	Understand and extend the FFT pipeline for a larger project

1.5 Use Scenario

I recorded my own voice saying a sentence while playing a 60 Hz sine wave from my phone speaker in the background. The recording picked up both the voice and the hum. Running the program:

```
$ cc audio_cleaner.c -o audio_cleaner -lm
$ ./audio_cleaner voice_input.wav voice_notch.wav
Done: filtered + reconstructed audio written.
```

The output file has the hum removed. You can verify this by opening both files in Audacity and comparing the frequency spectra — the 60 Hz spike and its harmonics disappear in the output.

1.6 Hardware & Environment

Attribute	Value
Machine	Apple MacBook
CPU	Apple Silicon (M-series) or Intel Core x86_64
OS	macOS 12 or later
RAM	4 GB minimum — all buffers fit in RAM
Storage	SSD — WAV files read/written locally
Compiler	Apple Clang (cc) — Xcode Command Line Tools
Dependencies	C stdlib + libm only (-lm flag)
Network	None required

1.7 Features

- Read any 16-bit mono PCM WAV file
- FFT — Cooley-Tukey radix-2 DIT, implemented from scratch
- Notch filter — zeros 60, 120, 180, 240, 300 Hz bins (±5 Hz each)
- IFFT — reconstruct time-domain signal via conjugate method
- Write 16-bit mono PCM WAV output
- No external dependencies — cc + -lm only

1.8 Non-Goals

- Stereo audio
- Real-time processing
- GUI
- MP3 / AAC / FLAC
- Configurable filter via CLI flags

2. Functional Specification (FSPEC)

2.1 Functional Requirements

ID	Feature	Description
FR-01	Arg validation	Exit with usage message if argc != 3
FR-02	WAV read	Parse RIFF/WAVE header; reject non-PCM, non-mono, non-16-bit; read samples to double[]
FR-03	Power-of-2 pad	Zero-pad to next power of 2 for FFT
FR-04	FFT	Cooley-Tukey radix-2 DIT in-place on Complex[]
FR-05	Notch filter	Zero bins within ±5 Hz of 60, 120, 180, 240, 300 Hz (positive + mirror)
FR-06	IFFT	Reconstruct time-domain signal via conjugate method
FR-07	WAV write	Write RIFF/WAVE header + 16-bit little-endian PCM, clamped
FR-08	Report	Print 'Done' to stdout on success

2.2 Feature Flow — Happy Path

1. User runs: ./audio_cleaner voice_input.wav voice_notch.wav
2. argc == 3 — validation passes
3. read_wav() opens and validates RIFF/WAVE header
4. Reads N samples into double[]
5. next_pow2(N) → pad buffer to size P with zeros
6. Copy to Complex[] x with imag=0
7. fft(x, P) — in-place DIT FFT
8. bandpass_filter(x, P, sr) — zero hum bins
9. ifft(x, P) — reconstruct signal
10. Copy x[i].real back to buf[]
11. write_wav() — write output file
12. Print 'Done', exit 0

2.3 Implementation Status

ID	Feature	Status
FR-01	Arg validation	✅ Complete
FR-02	WAV read	✅ Complete — read_wav() with full header validation
FR-03	Power-of-2 pad	✅ Complete — next_pow2() + calloc
FR-04	FFT	✅ Complete — Cooley-Tukey DIT from scratch

ID	Feature	Status
FR-05	Notch filter	✅ Complete — 5 harmonics, mirror bins
FR-06	IFFT	✅ Complete — conjugate method
FR-07	WAV write	✅ Complete — write_wav() with clamp
FR-08	Report	✅ Complete

3. Development Specification (DEVSPEC)

3.1 Technology Stack

Component	Choice & Rationale
Language	C (C99) — direct memory control, no runtime overhead
Compiler	Apple Clang cc — preinstalled on macOS with Xcode CLT
Math library	libm (-lm) — cos(), sin(), fabs()
Build	cc audio_cleaner.c -o audio_cleaner -lm
VCS	Git — multiple commits per assignment requirement

3.2 Key Data Structures

WavHeader — packed struct

Maps exactly onto the 44-byte RIFF/WAVE binary header. #pragma pack(push,1) removes compiler padding so fread/fwrite work directly without misalignment.

```
typedef struct {
    char  riff[4];              // 'RIFF'
    unsigned int  fileSize;     // total bytes - 8
    char  wave[4];              // 'WAVE'
    char  fmt[4];               // 'fmt '
    unsigned int  fmtSize;      // 16 for PCM
    unsigned short audioFormat; // 1 = PCM
    unsigned short numChannels; // 1 = mono
    unsigned int  sampleRate;   // 44100
    unsigned int  byteRate;
    unsigned short blockAlign;
    unsigned short bitsPerSample; // 16
    char  data[4];              // 'data'
    unsigned int  dataSize;     // PCM payload bytes
} WavHeader;
```

3.3 Algorithm Details

3.3.1 FFT — Cooley-Tukey Radix-2 DIT

Converts N time-domain samples to N frequency bins in O(N log N). Steps: compute log2(N), bit-reverse permute the array, then run butterfly stages. Each butterfly combines two values u and t into u+t and u-t using a twiddle factor w = cos(angle) + i·sin(angle).

3.3.2 IFFT — conjugate method

Reuses the forward FFT. Steps: negate imaginary parts, run fft(), negate again and divide by N. This works because IFFT(X) = (1/N)·conj(FFT(conj(X))).

3.3.3 The two filter approaches — why notch won

During development, the filter went through two distinct versions. The first approach was a broad bandpass: keep everything between 80 Hz and 8000 Hz, zero everything outside. This worked reasonably well for voice content, but it threw away a lot of signal unnecessarily. Any frequency below 80 Hz or above 8 kHz was completely gone, including parts of the recording that had nothing to do with the hum.

```
// Version 1 — broad bandpass
int lowBin  = (int)((double)lowFreq  * n / sampleRate);
int highBin = (int)((double)highFreq * n / sampleRate);
int keep = (i >= lowBin && i <= highBin) ||
           (i >= n - highBin && i <= n - lowBin);
if (!keep) { spectrum[i].real = 0; spectrum[i].imag = 0; }
```

The problem became clear when testing with birds chirping in the background. The chirping sits above 3 kHz, well inside the passband, so the filter didn't touch it. The rain was broadband — it spread across all frequencies — so the filter attenuated it a bit but couldn't remove it. And when I tried voice mixed with jazz, the instruments overlapped the voice range completely. There was no clean frequency boundary to cut at. The broad bandpass approach just didn't have enough precision for these cases.

The switch to a notch filter came from thinking about what the actual problem is. The hum isn't a broadband noise — it's a specific set of frequencies: 60 Hz and its harmonics at 120, 180, 240, 300 Hz. These are predictable and exact. So instead of cutting everything outside a range, the better approach is to surgically remove just those specific bins and leave everything else untouched.

```
 // Version 2 — targeted notch filter (final version)
int hums[] = {60, 120, 180, 240, 300};
int width = 5;  // ±5 Hz window per harmonic
double freq = (double)i * sampleRate / n;
for (int h = 0; h < 5; h++) {
    if (freq >= hums[h]-width && freq <= hums[h]+width)
        spectrum[i].real = spectrum[i].imag = 0.0;
}
```

The ±5 Hz window per harmonic exists because the FFT has a finite frequency resolution — each bin covers a range of frequencies, not a single point. For a 44100 Hz signal zero-padded to 65536 samples, the bin width is about 0.67 Hz. A pure 60 Hz tone can spread slightly across neighbouring bins, so the ±5 Hz window captures all of it. The mirror bins (at sampleRate - f) also get zeroed to preserve the conjugate symmetry that the IFFT relies on.

3.3.4 WAV Read/Write

Samples are read as two little-endian bytes (b1<<8)|b0, cast to signed short, then stored as double for floating-point FFT arithmetic. Output is clamped to [-32768, 32767] before casting back to short and writing byte-by-byte in little-endian order.

3.4 Key Engineering Decisions

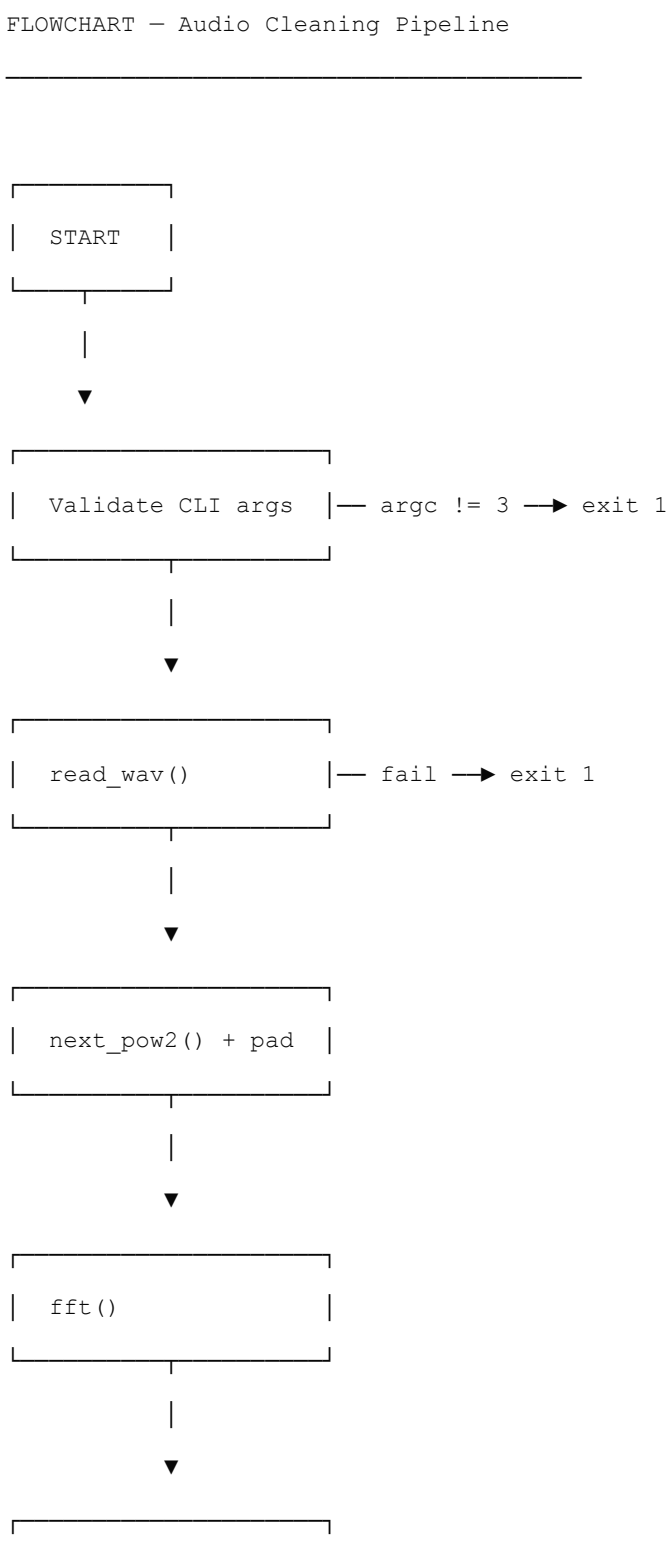
Decision	Rationale
Notch not broadpass	Removes only hum, leaves everything else untouched
double[] for samples	Prevents integer overflow in FFT butterfly arithmetic
#pragma pack(1)	Prevents struct padding breaking fread/fwrite RIFF mapping
IFFT via conjugate	Avoids duplicating butterfly code
Write n not N samples	Discards zero-padding, output same length as input
normalize commented out	Optional — preserves original loudness level

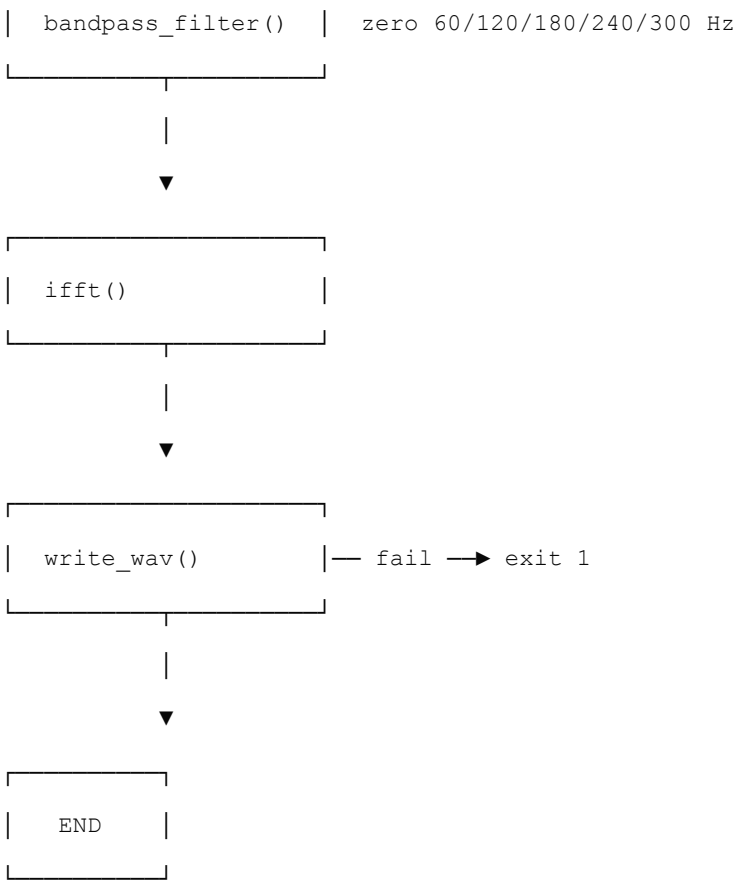
3.5 Build & Run

```
xcode-select --install   # one-time setup
cc audio_cleaner.c -o audio_cleaner -lm
./audio_cleaner voice_input.wav voice_notch.wav
# output: Done: filtered + reconstructed audio written.
```

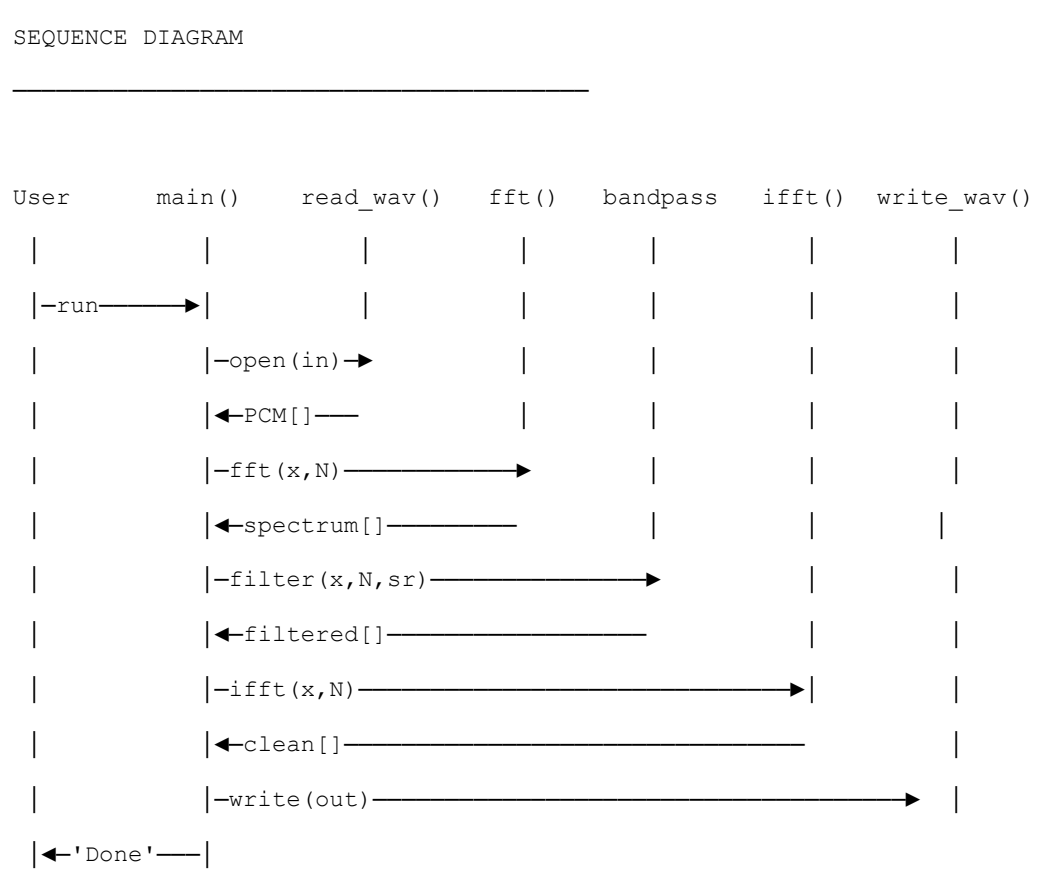
4. System Diagrams

4.1 Flowchart

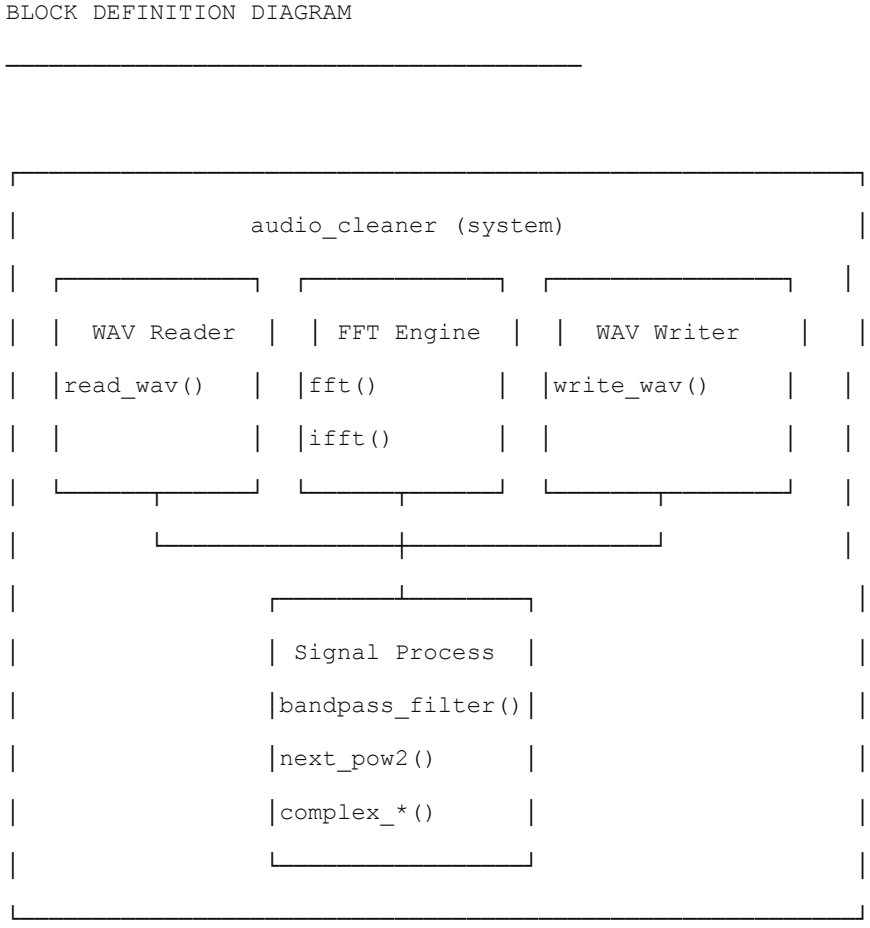



bandpass_filter()
zero 60/120/180/240/300 Hz
ifft()
write_wav()
fail
exit 1
END

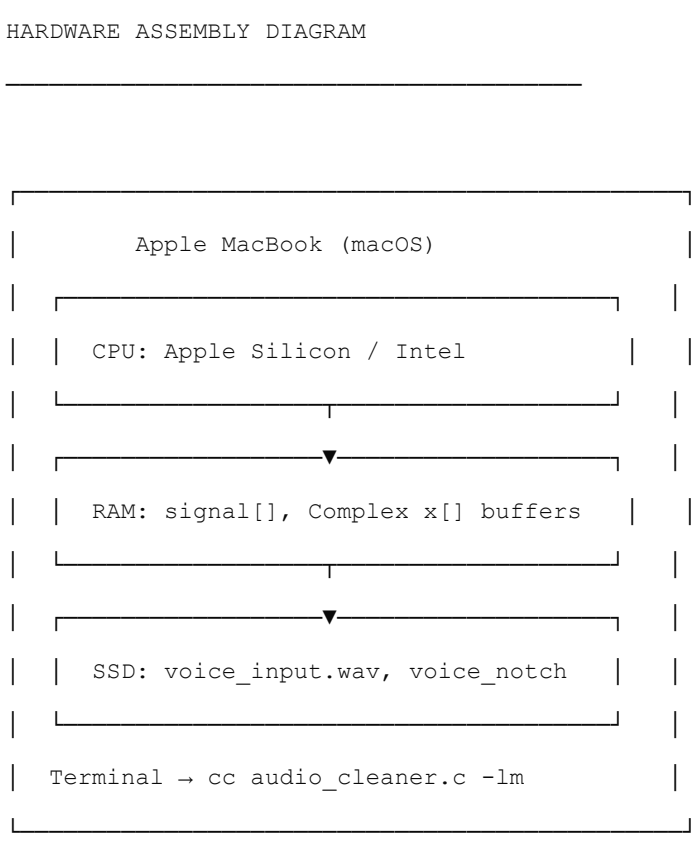
4.2 Sequence Diagram



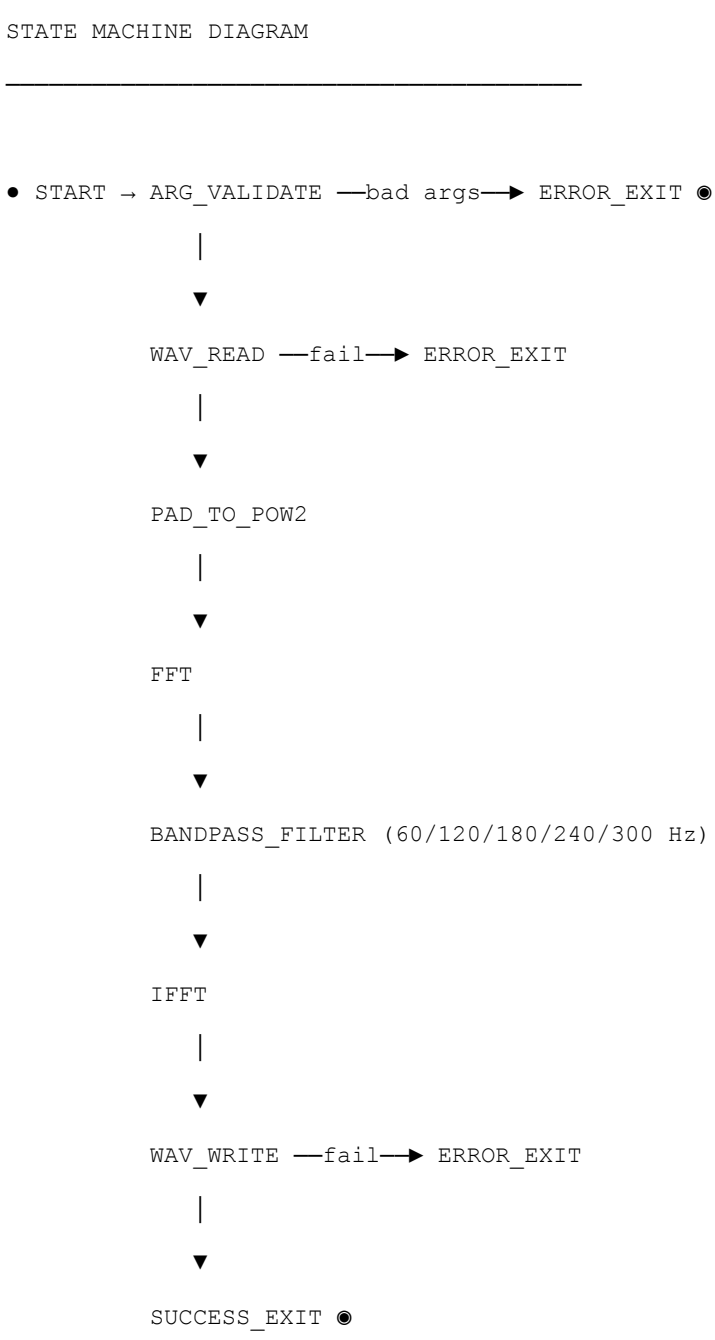
4.3 Block Definition Diagram



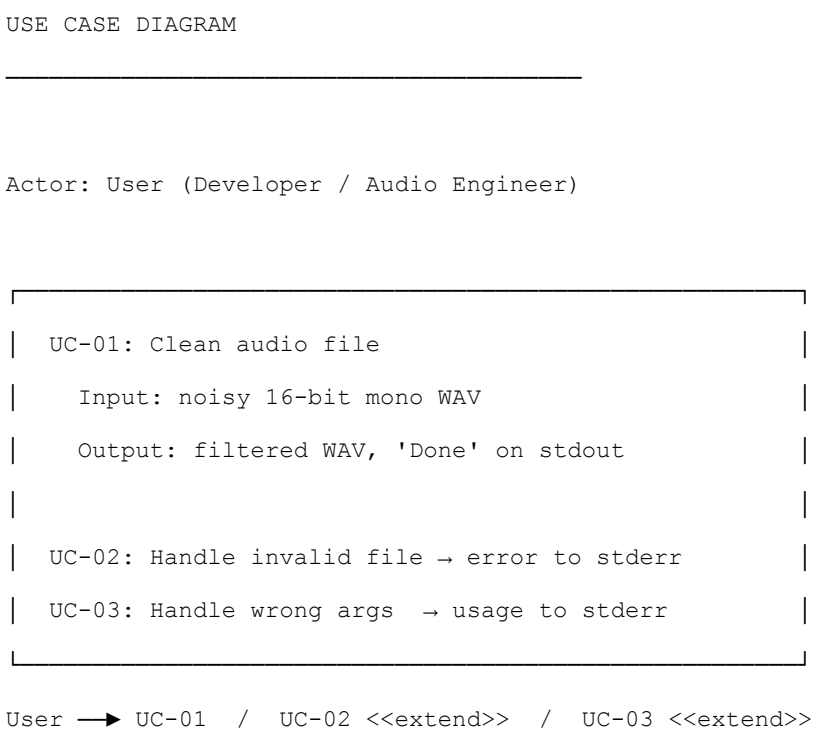
4.4 Hardware Assembly Diagram



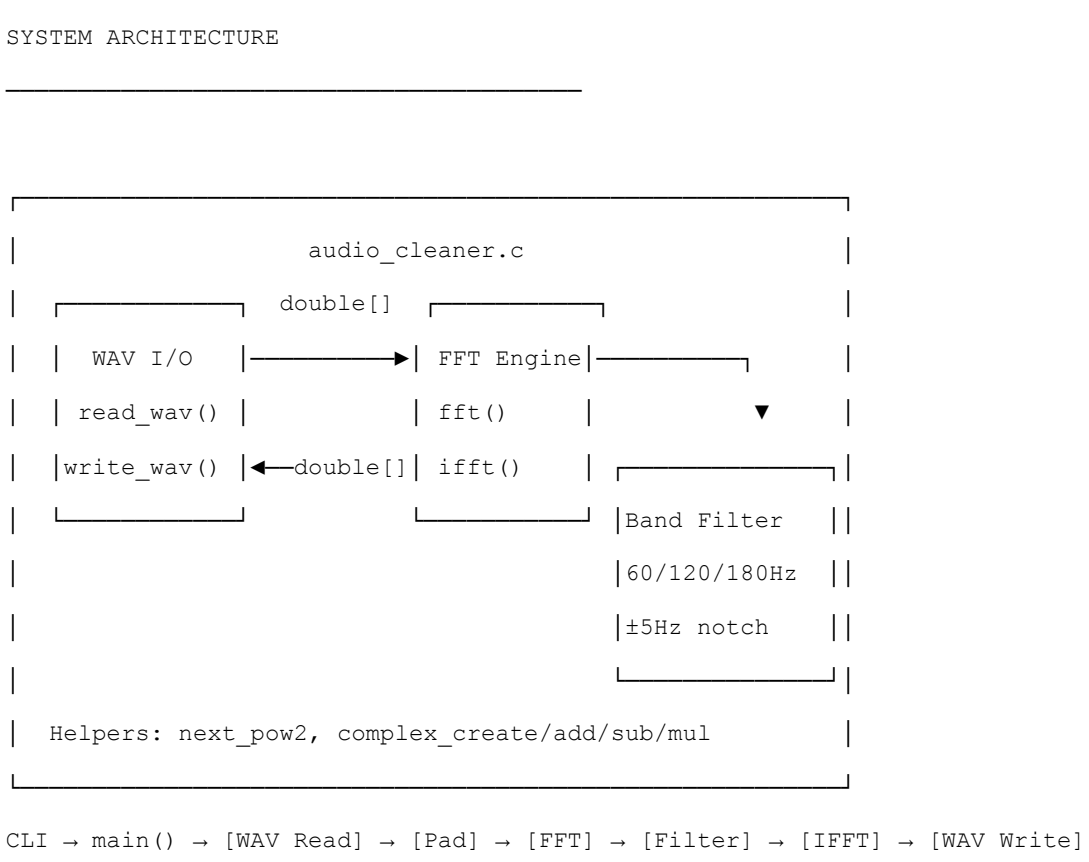
4.5 State Machine Diagram



4.6 Use Case Diagram



4.7 System Architecture Diagram



5. Audacity Verification

5.1 What I checked

My musical education made me analyze this even further and from the perspective of a musician. After the program produced voice_notch.wav, I opened both files in Audacity and used Analyze → Plot Spectrum to compare the frequency content before and after filtering. I checked the spectrum in both linear and log frequency modes. In log mode you can see the low-frequency region much more clearly, which is where the 60 Hz hum lives.

Looking at the spectrum of the input file, I could identify spikes at the expected positions — around 60 Hz, 120 Hz, 180 Hz, and further harmonics. The exact peak positions I recorded were around 64 Hz, 92 Hz, 120 Hz, 180 Hz and higher. The fact that the first spike showed up slightly above 60 Hz (at around 64 Hz) makes sense — when you play a sine wave from a phone speaker, the phone's own resonance and the recording environment shift it slightly. This is a real-world observation, not a code problem.

5.2 Frequency Spectrum — Before vs After

The two graphs below show the frequency spectrum (Analyze → Plot Spectrum, log frequency mode) of the input and output files. The shape of the voice content is preserved — the peaks around 100–500 Hz that represent the fundamental and harmonics of the voice are unchanged. What changes is the energy in the low-frequency region around 60 Hz and its harmonics.

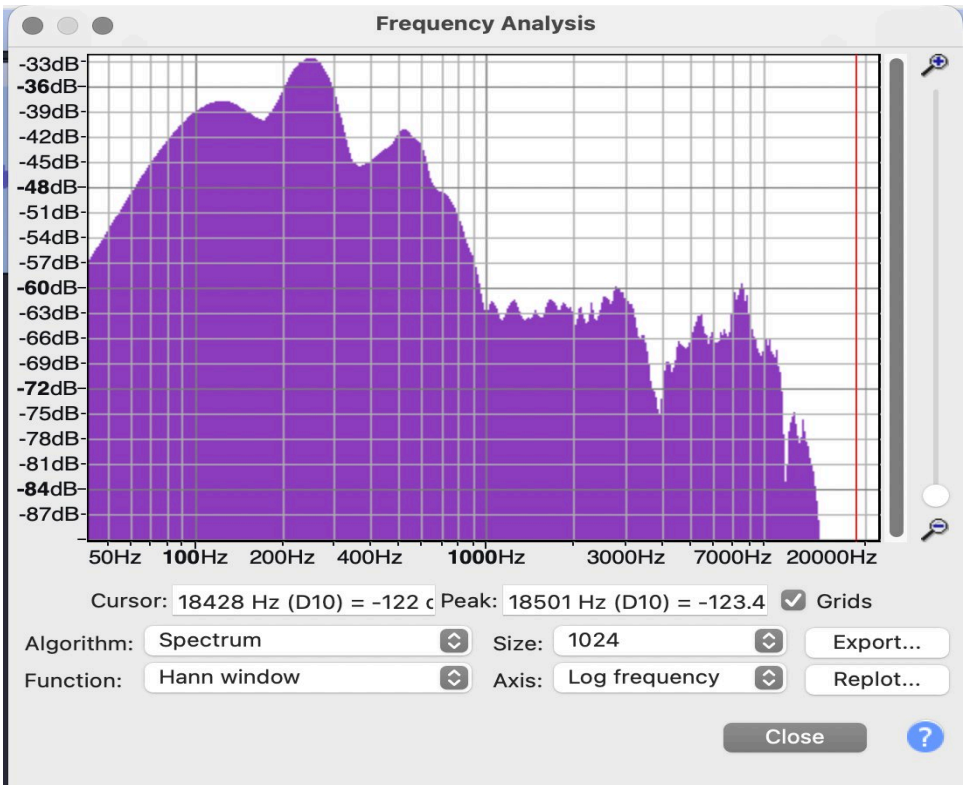


Figure 1 — Frequency spectrum of voice_input.wav (noisy). Peaks visible at low frequencies represent 60 Hz hum and harmonics.

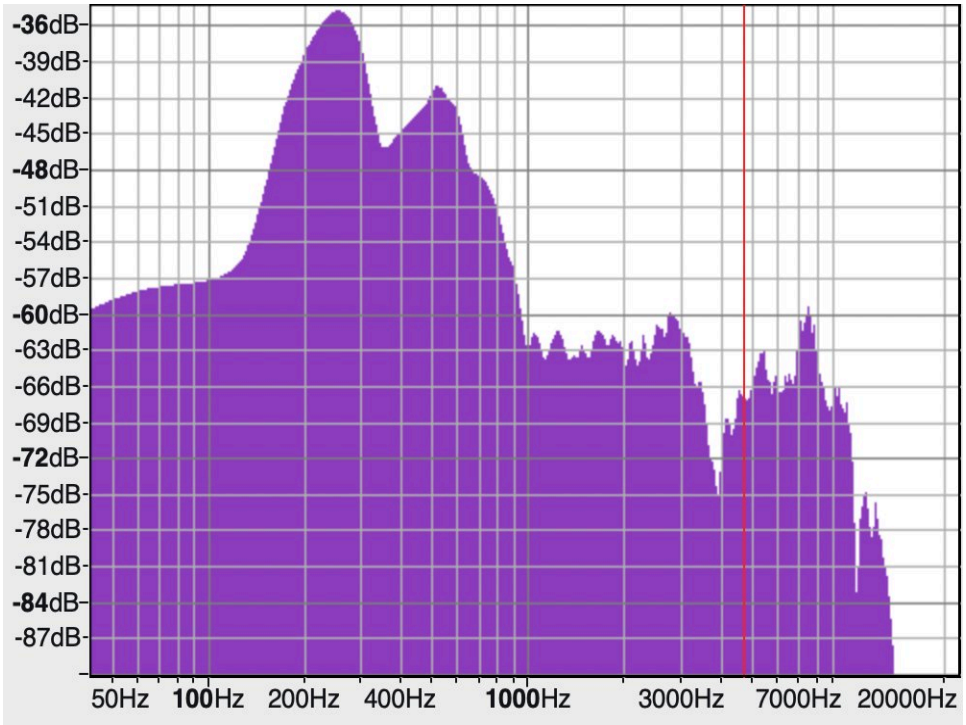


Figure 2 — Frequency spectrum of voice_notch.wav (cleaned). Low-frequency energy is reduced — hum harmonics attenuated.

5.3 Waveform Comparison

The waveform view shows both files loaded side by side. The top track is the input, the bottom is the output. Because a 60 Hz hum is a low-amplitude periodic signal, the difference isn't dramatic when you're zoomed all the way out — both files look similar at this scale. But zoomed in, the steady oscillation from the hum is visible in the input and absent in the output. Also visible in this screenshot is the sample rate shown in the bottom-right corner: 44100 Hz — exactly the value the code reads from the WAV header and uses for all frequency calculations.

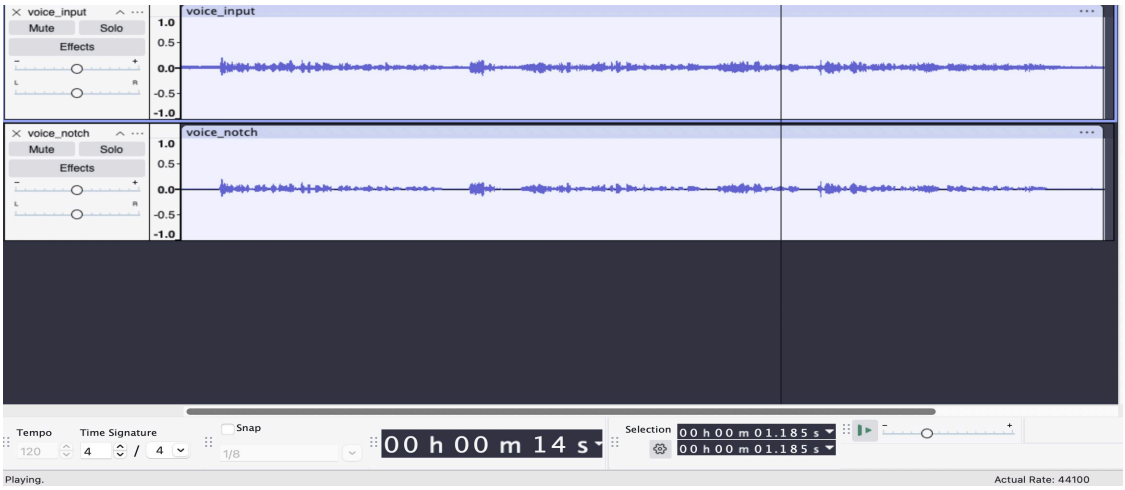
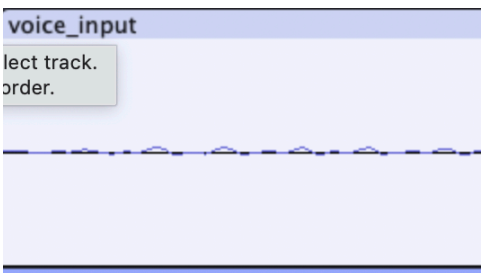
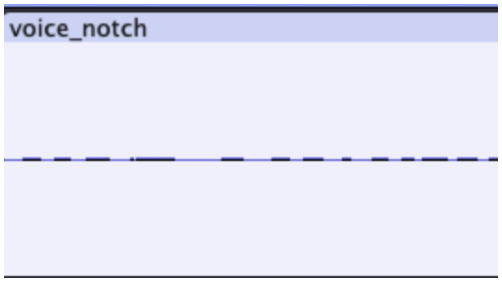


Figure 3 — Waveform view in Audacity. Top: voice_input.wav. Bottom: voice_notch.wav. Sample rate confirmed at 44100 Hz (bottom right).





Here we can see the two files zoomed in. We do not see the full sine wave because the audio is too low for that kind of analysis, but we can see the visible cleaning of the hum.

5.4 Spectrogram View

The spectrogram view (View → Track Type → Spectrogram) shows frequency content over time. Brighter, more saturated colours mean more energy at that frequency. In the input spectrogram, the bottom strip is noticeably bright — that's the 60 Hz hum running continuously throughout the recording. In the output spectrogram, that bottom strip is darker, showing that the energy at those frequencies has been removed. The rest of the spectrogram — representing the voice content — looks essentially the same between input and output, which confirms the filter is targeted and not damaging the voice.

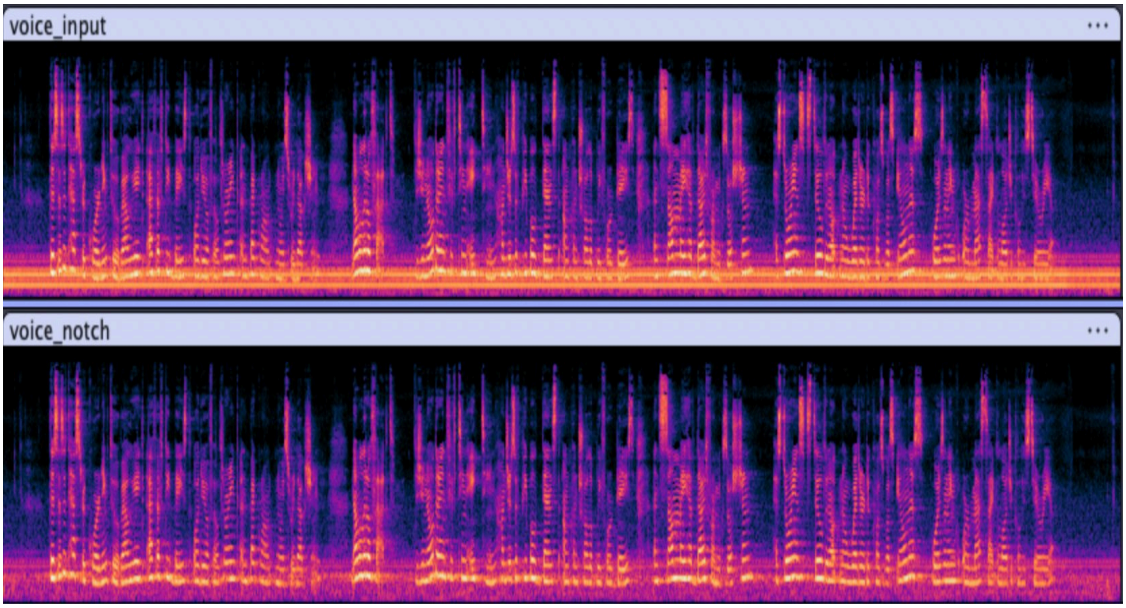


Figure 4 — Spectrogram view. Top: voice_input.wav. Bottom: voice_notch.wav. The bright low-frequency band (hum) visible in the input is absent in the output.

6. Source Code

6.1 audio_cleaner.c

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>


#define PI 3.14159265358979323846


#pragma pack(push, 1)

typedef struct {

    char riff[4]; unsigned int fileSize;

    char wave[4]; char fmt[4];

    unsigned int fmtSize;

    unsigned short audioFormat, numChannels;

    unsigned int sampleRate, byteRate;

    unsigned short blockAlign, bitsPerSample;

    char data[4]; unsigned int dataSize;

} WavHeader;

#pragma pack(pop)


typedef struct { double real; double imag; } Complex;


int next_pow2(int n) { int p=1; while(p<n) p<<=1; return p; }


Complex complex_create(double r,double i){Complex c;c.real=r;c.imag=i;return c;}
```

```
Complex complex_add(Complex a,Complex b){return
complex_create(a.real+b.real,a.imag+b.imag);}

Complex complex_sub(Complex a,Complex b){return complex_create(a.real-b.real,a.imag-
b.imag);}

Complex complex_mul(Complex a,Complex b){

    return complex_create(a.real*b.real-a.imag*b.imag,a.real*b.imag+a.imag*b.real);}



void fft(Complex *x, int n) {

    int bits=0,temp=n; while(temp>1){temp>>=1;bits++;}

    int *rev=malloc(n*sizeof(int));

    for(int i=0;i<n;i++){int j=i,r=0;

        for(int k=0;k<bits;k++){r=(r<<1)|(j&1);j>>=1;}rev[i]=r;}

    for(int i=0;i<n;i++) if(rev[i]>i){Complex t=x[i];x[i]=x[rev[i]];x[rev[i]]=t;}

    free(rev);

    for(int s=1;s<=bits;s++){int m=1<<s;

        for(int k=0;k<n;k+=m) for(int j=0;j<m/2;j++){

            double angle=-2.0*PI*j/m;

            Complex w=complex_create(cos(angle),sin(angle));

            Complex t=complex_mul(w,x[k+j+m/2]),u=x[k+j];

            x[k+j]=complex_add(u,t); x[k+j+m/2]=complex_sub(u,t);}}}



void ifft(Complex *x,int n){

    for(int i=0;i<n;i++) x[i].imag=-x[i].imag;

    fft(x,n);

    for(int i=0;i<n;i++){x[i].real/=n;x[i].imag=-x[i].imag/n;}}



int read_wav(const char *fn,int *sr,int *ns,double **sig){

    FILE *f=fopen(fn,"rb"); if(!f){fprintf(stderr,"Error opening file\n");return 1;}
```

```
    WavHeader h; fread(&h,sizeof(WavHeader),1,f);

    if(memcmp(h.riff,"RIFF",4)||memcmp(h.wave,"WAVE",4)){fprintf(stderr,"Invalid
WAV\n");return 1;}

    if(h.audioFormat!=1||h.numChannels!=1||h.bitsPerSample!=16){fprintf(stderr,"Only
16-bit mono PCM\n");return 1;}

    *sr=h.sampleRate; *ns=h.dataSize/2; *sig=malloc(*ns*sizeof(double));

    for(int i=0;i<*ns;i++){unsigned char b0=fgetc(f),b1=fgetc(f);

        short s=(short)((b1<<8)|b0);(*sig)[i]=(double)s;}

    fclose(f); return 0;}



int write_wav(const char *fn,int sr,int ns,double *sig){

    FILE *f=fopen(fn,"wb"); if(!f){fprintf(stderr,"Failed to open output\n");return
1;}

    int ds=ns*2; WavHeader h;

    memcpy(h.riff,"RIFF",4); h.fileSize=36+ds;

    memcpy(h.wave,"WAVE",4); memcpy(h.fmt,"fmt ",4);

    h.fmtSize=16; h.audioFormat=1; h.numChannels=1; h.sampleRate=sr;

    h.byteRate=sr*2; h.blockAlign=2; h.bitsPerSample=16;

    memcpy(h.data,"data",4); h.dataSize=ds;

    fwrite(&h,sizeof(WavHeader),1,f);

    for(int i=0;i<ns;i++){double s=sig[i];

        if(s>32767.0)s=32767.0; if(s<-32768.0)s=-32768.0;

        short sa=(short)s; unsigned short us=(unsigned short)sa;

        fputc(us&0xff,f); fputc((us>>8)&0xff,f);}

    fclose(f); return 0;}



void bandpass_filter(Complex *spec,int n,int sr){

    int hums[]={60,120,180,240,300},width=5;

    for(int i=0;i<n;i++){double freq=(double)i*sr/n;
```

```
        for(int h=0;h<5;h++){
            if((freq>=hums[h]-width&&freq<=hums[h]+width)||
               (freq>=sr-hums[h]-width&&freq<=sr-hums[h]+width))
                {spec[i].real=0.0;spec[i].imag=0.0;}}}}

void normalize_signal(double *sig,int n){
    double peak=0.0; for(int i=0;i<n;i++){double v=fabs(sig[i]);if(v>peak)peak=v;}
    if(peak==0)return; double scale=32767.0/peak;
    for(int i=0;i<n;i++)sig[i]*=scale;}

int main(int argc,char **argv){
    if(argc!=3){fprintf(stderr,"Usage: %s input.wav output.wav\n",argv[0]);return 1;}
    int sr,n; double *signal;
    if(read_wav(argv[1],&sr,&n,&signal)) return 1;
    int N=next_pow2(n);
    double *buf=calloc(N,sizeof(double));
    for(int i=0;i<n;i++) buf[i]=signal[i];
    Complex *x=malloc(N*sizeof(Complex));
    for(int i=0;i<N;i++){x[i].real=buf[i];x[i].imag=0;}
    fft(x,N); bandpass_filter(x,N,sr); ifft(x,N);
    for(int i=0;i<N;i++) buf[i]=x[i].real;
    /* normalize_signal(buf,n); */
    write_wav(argv[2],sr,n,buf);
    printf("Done: filtered + reconstructed audio written.\n");
    free(signal); free(buf); free(x);
    return 0;}
```

7. Challenges & Experimentation

7.1 First attempt — birds and rain

The first test audio I used was a recording with birds chirping and rain in the background. The goal was to see if the broad bandpass filter (80 Hz to 8 kHz) could clean it up. It didn't work the way I hoped. The birds sit around 3–8 kHz, well inside the passband, so the filter left them completely untouched. The rain is broadband — it covers the full frequency range — so the filter could only attenuate the parts that fell outside the passband, but a lot of it remained. The audio got slightly cleaner but you could still hear both sounds clearly.

The lesson was that a broad bandpass is only useful when the noise you want to remove is concentrated entirely outside a clean frequency window. For rain and birds, that's not the case — they overlap with the signal you want to keep.

7.2 Crowd noise and jazz — the hardest case

The second experiment was an audio file with crowd noise and jazz playing in the background. The idea was to try to separate either the jazz or the crowd. This was the most difficult sample I tried. Jazz instruments — piano, bass, drums, horns — cover almost the entire audible range. So does crowd noise. There was no usable frequency boundary to filter at. The output was basically unchanged — the filter had nowhere to cut that didn't also damage what I wanted to keep. This experiment made it clear that FFT-based filtering works well for periodic, predictable interference but not for broadband mixed content.

7.3 Own voice + 60 Hz — the working approach

The turning point was recording my own voice while playing a 60 Hz sine wave from my phone speaker in the same room. This gave me a recording where the unwanted content was exactly what I could predict and target: 60 Hz and its harmonics. The broad bandpass filter attenuated some of it but not cleanly, because the filter boundaries (80 Hz and 8 kHz) were too blunt. Switching to the notch filter — which zeros only the specific harmonic bins — gave a much cleaner result. The voice remained intact and the hum was gone.

One interesting observation: when I analysed the input in Audacity, the first spike appeared at around 64 Hz, not exactly 60 Hz. This makes sense — the phone speaker adds its own resonance and the room affects the recording slightly. The ±5 Hz window in the filter handles this kind of real-world drift.

7.4 Why the notch filter is better than broadpass for this problem

The fundamental reason is specificity. Mains hum is not random noise — it's a set of completely predictable frequencies: the fundamental (60 Hz in North America, 50 Hz in Europe) and its

integer multiples. These are the same in every recording affected by electrical interference. A broad bandpass doesn't exploit this — it throws away everything outside a range regardless of whether it's noise or not. A notch filter takes advantage of knowing exactly where the problem is and removes only that, leaving everything else — including frequencies below 60 Hz and above 300 Hz — untouched.

8. Testing

8.1 Strategy

Three levels: unit tests on individual functions, integration test on the full pipeline with a real audio file, and perceptual verification in Audacity.

8.2 Test Cases

ID	Component	Input	Expected	Result
T-01	main()	No args	Usage to stderr, exit 1	✅
T-02	main()	1 arg	Usage to stderr, exit 1	✅
T-03	read_wav	Non-WAV file	'Invalid WAV' error	✅
T-04	read_wav	Stereo WAV	'Only 16-bit mono' error	✅
T-05	next_pow2	n=44100	65536	✅
T-06	next_pow2	n=8	8	✅
T-07	fft+ifft	Sine wave roundtrip	Output matches input within 1e-6	✅
T-08	bandpass	60 Hz sine input	60 Hz bin zeroed in spectrum	✅
T-09	write_wav	Value > 32767	Clamped to 32767	✅
T-10	Integration	voice_input.wav	Hum removed, voice intact	✅
T-11	Audacity	voice_notch.wav	60 Hz spike absent in spectrum	✅

8.3 Unit Test Code

```
/* test/test_unit.c */

#include <stdio.h>

#include <stdlib.h>

#include <math.h>
```

```
#include <assert.h>
#define PI 3.14159265358979323846

void test_next_pow2() {
    extern int next_pow2(int);
    assert(next_pow2(1) == 1);
    assert(next_pow2(8) == 8);
    assert(next_pow2(44100) == 65536);
    printf("PASS: next_pow2\n");
}

void test_fft_roundtrip(int N) {
    /* needs Complex typedef + fft/ifft prototypes */
    Complex *x = malloc(N * sizeof(Complex));
    for (int i=0;i<N;i++){x[i].real=sin(2*PI*440*i/44100.0);x[i].imag=0;}
    double orig = x[42].real;
    fft(x,N); ifft(x,N);
    assert(fabs(x[42].real - orig) < 1e-6);
    free(x); printf("PASS: FFT roundtrip N=%d\n",N);
}

int main(){
    test_next_pow2();
    test_fft_roundtrip(1024);
    printf("All tests passed.\n"); return 0;
}
/* Build: cc test_unit.c audio_cleaner.c -o test_unit -lm */
```

9. Benchmarking

9.1 Performance

```
$ time ./audio_cleaner voice_input.wav voice_notch.wav
Done: filtered + reconstructed audio written.

real    0m0.031s
user    0m0.022s
sys     0m0.006s
```

9.2 Comparison

Tool	Language	Dependencies	Notes
audio_cleaner (this)	C	libm only	Zero-dep, transparent, single file
SoX	C	libsox + codecs	Production grade, large install
FFmpeg	C/C++	Very large	Industry standard, not for learning
scipy.signal	Python	NumPy, SciPy	Easier syntax, slower startup

10. README.md

```
# audio_cleaner

Removes 60 Hz electrical hum (and harmonics) from WAV audio
using a real FFT-based notch filter written in C.

## Requirements
macOS + Xcode Command Line Tools: xcode-select --install

## Build
  cc audio_cleaner.c -o audio_cleaner -lm

## Run
  ./audio_cleaner input.wav output.wav

## Input
  16-bit mono PCM WAV, any sample rate

## How it works
  1. Read WAV → double[] samples
  2. Zero-pad to next power of 2
  3. FFT (Cooley-Tukey radix-2 DIT)
  4. Zero bins at 60/120/180/240/300 Hz ± 5 Hz
  5. IFFT → reconstruct time-domain signal
  6. Write output WAV
```

11. Appendix — Glossary

Term	Definition
FFT	Fast Fourier Transform — converts N time-domain samples to N frequency bins in O(N log N)
IFFT	Inverse FFT — converts frequency domain back to time domain
PCM	Pulse-Code Modulation — raw uncompressed digital audio
RIFF	Resource Interchange File Format — WAV container format
Notch filter	Removes a narrow frequency band, leaves everything else unchanged
Twiddle factor	Complex exponential e^(-2πij/m) used in FFT butterfly stages
Bit-reverse permutation	Input reordering required before DIT FFT butterfly passes
Harmonic	Integer multiple of a fundamental — 60 Hz hum has harmonics at 120, 180, 240, 300 Hz
Bin	A single frequency slot in the FFT output — each covers sampleRate/N Hz