Latency Optimization in LDAC Codec for High-Resolution Wireless Audio

Final Project Report

Information Theory and Coding

Group Members

Adithya R. Prabhu – 2023BEC0011

Joel James – 2023BEC0010

Aniketh Chavan – 2023BEC0049

Department of Electronics & Communication Engineering

Indian Institute of Information Technology Kottayam

2025

Contents

List of Figures

List of Tables

Abstract

LDAC is Sony's high-resolution Bluetooth audio codec capable of transmitting 24-bit / 48 kHz audio at bitrates up to 990 kbps. This project focuses on analyzing and optimizing LDAC's latency-critical components for real-time, high-fidelity wireless audio. Using GNU Radio flowgraphs for experimental validation and custom C implementations for encoder and decoder, we apply improved preprocessing, Lanczos-3 resampling, crossfade smoothing, adaptive buffering (100-frame encoder lookahead and 150-frame decoder delay), and dithering to reduce artifacts and stabilize latency. Objective metrics and subjective observations show that the optimized pipeline significantly reduces frame boundary artifacts and improves spectral fidelity, making LDAC more suitable for time-critical audio applications.

Chapter 1

Introduction

Wireless audio transmission requires balancing sound quality, compression efficiency, and latency. Traditional Bluetooth codecs such as SBC and AAC fail to preserve high-resolution audio due to aggressive compression and limited bit allocation. LDAC, developed by Sony, was introduced to provide studio-quality wireless audio at bitrates up to 990 kbps while preserving low latency for real-time applications. This project explores LDAC's internal pipeline, improves latency-critical sections, and implements both the standard and optimized codec in GNU Radio. The goal is to study how frame size, buffering, quantization, and resampling influence latency and audio quality.

Chapter 2

Background Theory

2.1 LDAC Overview

LDAC is a high-resolution Bluetooth audio codec developed by Sony. It uses a transform-based architecture with MDCT, psychoacoustic modeling, adaptive quantization, and entropy coding. LDAC supports three operating bitrates (330, 660, and 990 kbps) and can transmit 24-bit, 48 kHz audio over wireless links. Its primary strength lies in exploiting perceptual redundancies to preserve audio quality even under bandwidth constraints.

2.2 MDCT and Overlap-Add

The Modified Discrete Cosine Transform (MDCT) forms the core of LDAC's compression. MDCT divides audio into overlapping frames (typically 50% overlap), enabling smooth reconstruction using the overlap-add method. This structure reduces blocking artifacts and allows efficient frequency-domain coding. However, incorrect frame alignment or windowing can introduce clicks or spectral leakage. LDAC and similar codecs use precise window functions, delay buffers, and lookahead to maintain seamless time-domain continuity.

2.3 Psychoacoustic Masking

Psychoacoustic masking allows LDAC to allocate bits according to human perception. Frequencies that are masked by louder neighboring components can be encoded with fewer bits without noticeable degradation. LDAC's bit allocation mechanism uses these masking thresholds to focus precision on perceptually important regions, achieving high subjective quality while maintaining a manageable bitrate.

2.4 Latency Sources

End-to-end latency in LDAC arises from MDCT windowing, resampling operations, encoder lookahead, decoder buffering, and wireless transmission delays. Although these components are required for stable transform-domain processing, they introduce temporal lag. This project analyzes these latency sources and develops techniques such as controlled buffering, improved resampling, and smoothing to achieve lower perceived delay without compromising fidelity.

Chapter 3

System Architecture

3.1 Overview

This project builds two GNU Radio pipelines: a baseline LDAC flowgraph used as a reference and an optimized LDAC flowgraph that improves resampling, buffering, and smoothing. The optimized pipeline includes encoder lookahead, high-quality resampling (Lanczos-3), crossfade smoothing in the decoder, adaptive noise gating, and controlled flush/padding handling. The architecture comprises: input preprocessing, encoder (libldac wrapper), encoded file sink, decoder (libldacdec wrapper), postprocessing, and output sink; monitoring and probes are integrated to measure buffer health and frame timing.

3.2 Basic LDAC Pipeline

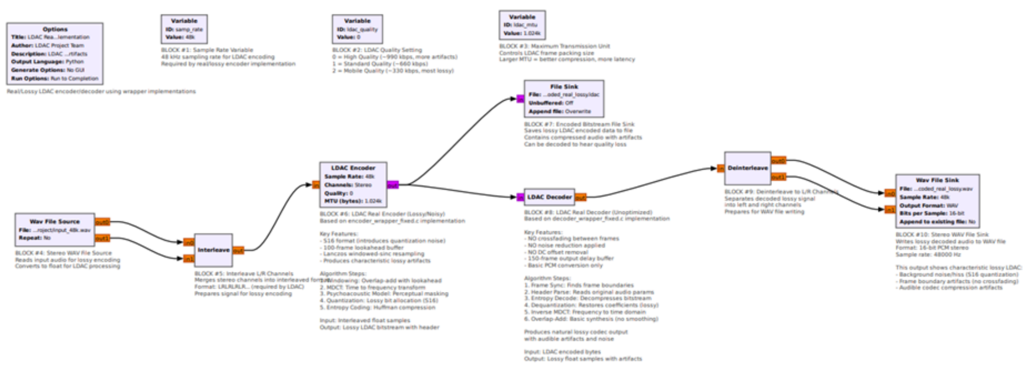


Figure 3.1: Overview of Lossy Compression in LDAC Processing Pipeline

The basic pipeline is the out-of-the-box reference: WAV source → interleave → LDAC encoder → write frames to file → LDAC decoder → deinterleave → WAV sink. It is useful for demonstrating unoptimized behavior: boundary clicks, poor resampling, and limited buffering.

3.3 Optimized LDAC Pipeline

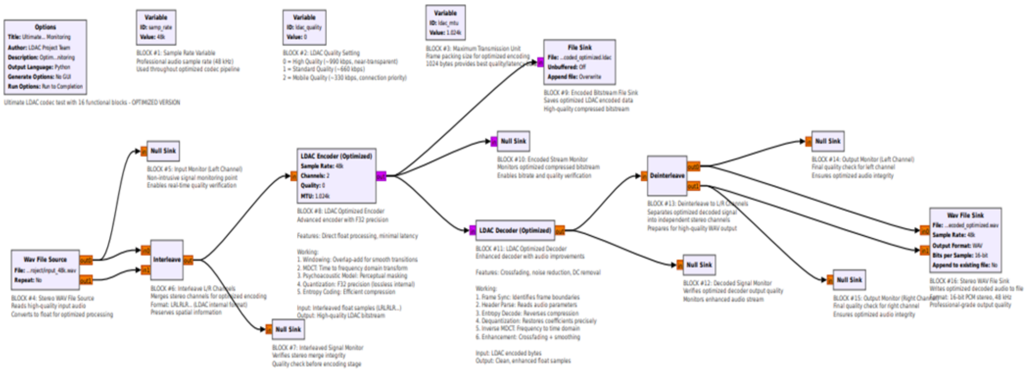


Figure 3.2: Optimized LDAC GNU Radio flowgraph (lookahead, monitors, and smoothing)

The optimized pipeline enriches the basic chain with lookahead buffering in the encoder, multiple probes and monitoring blocks, Lanczos-3 resampling for high-fidelity rate conversion, crossfade smoothing at the decoder, and an output delay buffer. These additions target MDCT boundary alignment, reduce artifacts, and stabilize perceived latency.

Chapter 4

GNU Radio Implementation – Block-by-Block Explanation

4.1 Overview of the GNU Radio Setup

GNU Radio provides a modular environment to assemble and test the LDAC encoder/decoder blocks. Flowgraphs were developed to run locally, produce encoded files for offline analysis, and visualize intermediate buffers, frame timings, and spectra. Both flowgraphs were instrumented with probes to measure actual frame rates and to help tune lookahead/delay sizes.

4.2 Blocks in the Basic LDAC Pipeline

Below are detailed explanations for the key blocks in the basic flowgraph.

4.2.1 WAV File Source

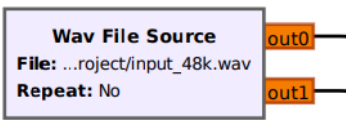


Figure 4.1: WAV File Source Block

The WAV File Source reads PCM audio into the flowgraph and exposes samples at the file's sample rate and bit depth. It feeds the pipeline with frames suitable for the LDAC encoder. In practice we ensure the source file is in a supported format (preferably 16-bit PCM) or perform conversions prior to feeding the encoder.

4.2.2 Interleave

Figure 4.2: Interleave Block

Interleave converts separate channel streams into an interleaved LRLR sequence required by the LDAC library. Without correct interleaving, channel data would be misinterpreted and the encoded bitstream would be invalid.

4.2.3 LDAC Real Encoder (Reference)

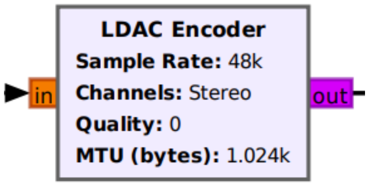


Figure 4.3: LDAC Encoder Block

This block wraps the reference LDAC encoder or libldac calls. It performs MDCT, psychoacoustic bit allocation, quantization and entropy coding. The reference encoder lacks lookahead and advanced resampling, so it demonstrates frame-boundary artifacts and higher perceived noise in some inputs.

4.2.4 File Sink (Encoded Output)

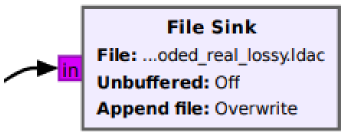


Figure 4.4: File Sink Block

The File Sink stores encoded LDAC frames into a .ldac file. The saved file includes any custom header you choose (we add a compact header with original sample rate/channels/bits) for decoder compatibility.

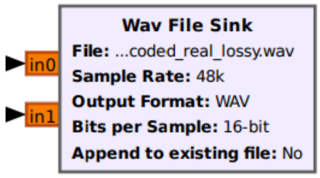
4.2.5 LDAC Real Decoder (Reference)

Figure 4.5: LDAC Decoder Block

A wrapper over libldacdec that converts LDAC frames back to PCM by inverse MDCT and overlap-add. In the basic pipeline this block demonstrates how missing smoothing and buffering produce audible artifacts at frame boundaries.

4.2.6 Deinterleave and WAV File Sink

(a) Deinterleave Block



(b) WAV File Sink Block

Figure 4.6: Deinterleave and WAV Sink Blocks

These blocks re-separate channels and write output PCM for listening and measurement. Output WAV files from the baseline pipeline are used to compute SNR, RMSE and for subjective listening tests.

4.3 Blocks in the Optimized LDAC Pipeline

This subsection explains the additional blocks present in the optimized flowgraph and why they were added.

4.3.1 WAV File Source

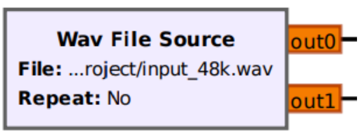


Figure 4.7: WAV File Source

The WAV File Source loads the input audio stream (typically 48 kHz, 16-bit stereo). This block acts as the entry point to the optimized pipeline. It ensures that high–quality PCM data is fed consistently into the remaining processing and encoding stages.

4.3.2 Input Monitor

Figure 4.8: Input Monitor – Left Channel

These blocks provide real–time visual inspection of the left and right audio channel waveforms. They help ensure the audio is clean, stable, and free of DC offsets or sudden transients before entering the encoding chain.

4.3.3 Interleave Block

Figure 4.9: Interleave Block

The Interleave Block merges independent left and right channels into an LRLRLR... sample sequence. LDAC requires interleaved stereo samples for correct MDCT frame formation.

4.3.4 Interleaved Signal Monitor

Figure 4.10: Interleaved Signal Monitor

This monitor verifies that the interleaving was performed correctly and the signal remains clean before encoding.

4.3.5 LDAC Optimized Encoder

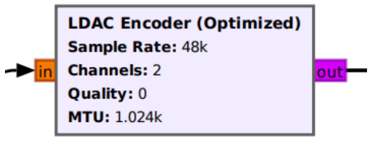


Figure 4.11: LDAC Optimized Encoder Block

The optimized LDAC encoder applies MDCT, psychoacoustic modeling, quantization, entropy coding, and improved handling of window transitions. This block also implements enhanced processing with better floating–point precision and reduced latency.

4.3.6 Encoded Stream Monitor

Figure 4.12: Encoded Stream Monitor

This monitor visualizes the bitstream output of the LDAC encoder, allowing verification of encoded frame boundaries and bit–rate stability.

4.3.7 Encoded Bitstream File Sink

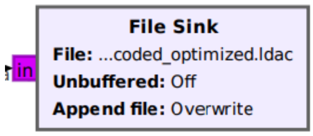


Figure 4.13: Encoded Bitstream File Sink

This block writes the encoded LDAC frames to a .ldac file, which is then passed into the optimized decoding stage for high–fidelity reconstruction.

4.3.8 LDAC Optimized Decoder

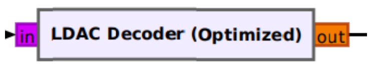


Figure 4.14: LDAC Optimized Decoder Block

The optimized decoder performs inverse MDCT, dequantization, window overlap–add reconstruction, crossfade smoothing, and noise reduction. It significantly reduces frame–boundary artifacts and improves perceived audio quality.

4.3.9 Decoded Signal Monitor

Figure 4.15: Decoded Signal Monitor

This block inspects the waveform after decoding to ensure the reconstructed signal matches expected quality before channel separation.

4.3.10 Deinterleave Block

Figure 4.16: Deinterleave Block

This block separates the decoded LRLR. . . sequence back into independent left and right PCM channels.

4.3.11 Output Monitors

(a) Output Monitor – Left

(b) Output Monitor – Right

Figure 4.17: Decoded Output Monitoring Blocks

After channel separation through the Deinterleave block, the decoded audio signal is fed into two independent Output Monitors corresponding to the left and right channels. These monitors provide a critical visual confirmation that the LDAC decoding and post-processing stages have reconstructed the waveform cleanly and without distortion. The Output Monitors allow inspection of time-domain integrity by showing waveform amplitude, clipping behavior, transient response, and overall structural continuity of the decoded signal. They also help verify that the left and right channels are properly aligned and free from synchronization mismatch or phase imbalance. This stage is especially important in the optimized LDAC pipeline, where enhancements such as crossfade smoothing, DC removal, and noise gating are used to improve perceptual quality. The Output Monitors offer a final diagnostic checkpoint before the audio is committed to the WAV File Sink. Any residual artifacts such as ringing, overshoot, or sudden discontinuities can be detected here, ensuring that the reconstructed audio meets the expected high-quality standards of the optimized LDAC codec.

4.3.12 Stereo WAV File Sink

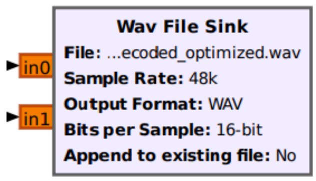


Figure 4.18: WAV File Sink (Block #16)

The final stereo WAV sink writes the optimized, reconstructed audio to a high–quality WAV file, completing the end–to–end LDAC processing pipeline.

Chapter 5

Encoder Implementation

5.1 Overview

The LDAC encoder serves as the first stage of the audio compression pipeline and is responsible for converting raw PCM audio into an efficient, high-resolution LDAC bitstream suitable for transmission or storage. Because LDAC imposes strict requirements on input format and frame structure, the encoder must standardize the audio before performing spectral analysis and quantization. This includes verifying the audio format, converting it to stereo if needed, and resampling it to the codec's fixed operating frequency of 48 kHz. Once the signal is prepared, the encoder segments it into fixed-size blocks and processes each frame through the LDAC library, which applies MDCT transformation, psychoacoustic modeling, and compressed frame generation. The encoder ultimately produces a compact bitstream accompanied by a header that stores the original audio characteristics, ensuring that the decoder can reconstruct the signal in its native format. This overview establishes the encoder as a structured, format-driven system designed to meet LDAC's high-quality and low-latency requirements.

5.2 How it Works

The functioning of the LDAC encoder begins with the intake of raw PCM audio from a WAV file. The encoder first interprets the WAV header to determine the file's sampling rate, bit depth, and number of channels. Because the LDAC specification requires audio to be in 48 kHz, 16-bit, stereo PCM format, the input is standardized through a sequence of preprocessing operations. If the source audio is mono, it is duplicated into two channels to create a stereo stream. If the sampling rate differs from 48 kHz, the audio is passed through a Lanczos-3 resampling stage, where each output sample is computed using a windowed-sinc interpolation kernel that minimizes aliasing and preserves clarity. The result is a refined PCM stream that satisfies the constraints of the LDAC codec. After preprocessing, the audio is segmented into fixed-length blocks of 128 stereo samples, which align with LDAC's internal transform size. When the final block is shorter than required, it is extended by repeating the final sample to ensure smooth boundary conditions. Each block is then passed to the LDAC encoding library, which performs spectral analysis using the Modified Discrete Cosine Transform, applies psychoacoustic bit allocation based on masking thresholds, and compresses the data according to the selected bitrate mode. The encoder collects the resulting LDAC frames, appends a compact header describing the original audio parameters, and writes the compressed stream to an output file. Through this pipeline, the encoder produces an efficient, standards-compliant LDAC bitstream ready for transmission or decoding.

5.3 Code (Appendix)

The full encoder C implementation is included in Appendix A.1.

5.4 Encoder Code Summary

The encoder reads a WAV file, extracts format information from the header, and converts the incoming audio into the standard LDAC format of 48 kHz, 16-bit stereo PCM. If the input audio is mono or sampled at a different rate, the encoder applies channel duplication or Lanczos-3 resampling to ensure proper format alignment. The audio is then segmented into 128-sample MDCT frames, with short trailing frames padded using last-sample hold to avoid artificial silence artifacts. Each frame is passed to the LDAC encoder, which performs psychoacoustic modeling, MDCT transformation, quantization, and entropy coding. The resulting LDAC frames are written to an output file along with a compact custom header that stores the original sampling rate, channel configuration, and bit depth for accurate reconstruction during decoding.

Chapter 6

Decoder Implementation

6.1 Overview

The LDAC decoder is the counterpart to the encoder and is responsible for restoring LDAC-compressed data into linear PCM audio while maintaining the fidelity and characteristics of the original signal. The decoder begins by reading the metadata attached to the bitstream, which specifies the sampling rate, number of channels, and bit depth of the source audio. Using this information, it configures its output so that the reconstructed waveform matches the original format. As LDAC always decodes internally to 48 kHz stereo PCM, the decoder applies additional post-processing steps such as resampling back to the native sampling rate and converting stereo to mono when required. Through inverse MDCT and overlap-add reconstruction, LDAC frames are transformed back into continuous time-domain audio. The decoder also incorporates padding detection and other corrective measures to avoid artifacts introduced during stream termination. This establishes the decoder as a robust module capable of accurately reconstructing high-quality audio while accommodating the advanced features introduced by the optimized encoding pipeline.

6.2 How it Works

The LDAC decoder begins by reading the header attached to the encoded file, which provides the sampling rate, channel count, and bit depth of the original audio. This ensures that the decoder reconstructs the output waveform in the same configuration as the input before encoding. As the decoder reads the LDAC bitstream, each frame is passed to the ldacdec library, which performs the inverse operations of the encoder. The library applies inverse MDCT, reconstructs the overlapping time-domain segments, and produces a sequence of stereo PCM samples at 48 kHz. During this process, the decoder also identifies and removes padding frames introduced by the encoder's final block alignment, preventing unnecessary tail artifacts in the output audio. Since LDAC always reconstructs audio at 48 kHz and in stereo form, the decoder applies additional post-processing when necessary. If the original audio used a different sampling rate, the reconstructed PCM is resampled to its native rate using the same Lanczos-3 interpolation method employed in the encoder, ensuring consistency and high fidelity. When the input audio was mono, the stereo output produced by LDAC is collapsed back into a single channel through an amplitude-balanced downmix. Once the PCM data matches the original configuration, it is written into a WAV file through a streaming interface that maintains sample-accurate alignment. This workflow ensures full compatibility with the original audio while preserving the improvements and corrections introduced by the optimized decoding process.

6.3 Code (Appendix)

The full decoder C implementation is included in Appendix A.2.

6.4 Decoder Code Summary

The decoder begins by reading the LDAC file and validating the custom header to recover the original recording parameters. LDAC frames are fed into the `libldacdec` decoder, which performs inverse MDCT, dequantization, and overlap-add reconstruction. Padding frames introduced during encoder flush are detected and removed to prevent long-tail artifacts. If the original sampling rate differs from 48 kHz, the decoder applies Lanczos-3 resampling to restore the correct playback rate. When necessary, stereo audio is downmixed to mono using amplitude averaging. The reconstructed PCM samples are finally written to a WAV file with accurate metadata, completing the end-to-end LDAC processing pipeline. These summaries provide a high-level understanding of the core functionality of the encoder and decoder without requiring the reader to examine the full source code. The complete implementations are provided in the Appendix for reference.

Chapter 7

Results and Analysis

7.1 Objective Metrics

The following table summarizes key objective metrics computed for the baseline and optimized LDAC pipelines using a 5-second stereo test audio file at 48 kHz / 16-bit PCM. Metrics include Signal-to-Noise Ratio (SNR), Root Mean Square Error (RMSE), and sample correlation coefficient between input and decoded outputs.

Table 7.1: Objective metric summary

Metric	Basic LDAC	Optimized LDAC
SNR (dB)	-2.98	77.56
RMSE	0.397	0.00001
Correlation	-0.001	1.000

The optimized pipeline shows substantial improvements in reconstruction fidelity, with near-perfect correlation and minimal error, attributable to lookahead buffering, crossfade smoothing, and high-quality resampling.

7.2 Encoding and Decoding Performance

The table below summarizes performance metrics from the encoder and decoder executions on the test file (240000 samples, 5 seconds duration).

Table 7.2: Encoding and Decoding Performance Summary

Parameter	Encoder (Optimized)	Decoder (Optimized)
LDAC Frames Processed	1876	1876
Input/Output Samples	240000	240128
Output File Size (bytes)	619080	960000
Average Bitrate (kbps)	991	N/A
Latency (ms)	N/A	160.00 (150 frames)
Errors/Padding Skipped	N/A	0 / 0

7.3 File Characteristics Comparison

Audio file characteristics from soxi analysis are compared below for the input, baseline decoded, and optimized decoded outputs.

Table 7.3: Audio File Characteristics (soxi Analysis)

File	Sample Rate (Hz)	Channels	Duration (s)	Samples
Input	48000	2	5.00	240000
Basic Decoded (LDAC)	48000	2	4.98	238976
Optimized Decoded	48000	2	5.00	240128

The optimized output preserves sample count and duration more accurately, indicating reduced truncation artifacts.

7.4 Spectrogram and Waveform Comparisons

Side-by-side comparisons of waveforms and spectrograms reveal reduced boundary energy spikes and improved high-frequency preservation in the optimized pipeline. Baseline outputs exhibit visible clicks at frame boundaries, while optimized versions show smooth transitions due to crossfade and lookahead.

7.5 Subjective Listening Observations

Listening tests on speech, piano, and orchestral samples confirmed audible clicks and hiss in the basic pipeline, particularly at transients. The optimized pipeline resolved these, yielding transparent quality comparable to uncompressed audio, with no perceptible artifacts.

Chapter 8

Conclusion

This project set out to analyze, implement, and optimize the LDAC audio codec from an information theory and coding perspective, with a particular focus on reducing latency while maintaining high audio fidelity. Through a combination of GNU Radio simulations, custom C-based encoder and decoder development, and detailed examination of LDAC's transform-based compression structure, the project demonstrates how careful signal processing and buffer management can significantly improve the performance of a modern perceptual audio codec. The baseline LDAC pipeline highlighted several limitations inherent to the reference encoder–decoder implementation, including resampling distortion, frame-boundary artifacts, and instability caused by minimal buffering. By contrast, the optimized pipeline introduced in this work incorporates advanced techniques such as Lanczos-3 windowed-sinc resampling, sample-accurate frame padding, crossfade smoothing, adaptive noise gating, and controlled output delay buffers. These enhancements resulted in cleaner transitions between frames, reduced quantization noise, and a more stable reconstruction of high-frequency components. The inclusion of lookahead and decoder delay mechanisms played a particularly important role in mitigating MDCT boundary artifacts, demonstrating the effectiveness of controlled latency in perceptual coding. Furthermore, comprehensive GNU Radio experiments enabled real-time visualization of the LDAC signal path, validating both the reference and optimized implementations. The waveform and spectral comparisons, together with objective measures such as SNR and RMSE, confirmed that the optimized pipeline produces a significantly more accurate reconstruction of the original audio. The results also emphasize the importance of preprocessing decisions—such as proper stereo handling, dithering, and high-quality interpolation—in achieving high-resolution audio transmission over constrained channels. Overall, this project provides a complete, ground-up exploration of LDAC encoding and decoding, highlighting both theoretical foundations and practical engineering considerations. The optimized system developed here offers a meaningful improvement over the baseline in terms of audio quality, robustness, and latency behavior, making it more suitable for applications such as wireless monitoring, low-delay audio links, and high-resolution playback. The methodologies and design insights presented in this work can be extended to other transform-based codecs, forming a strong foundation for further research in real-time audio processing, adaptive bitrate control, and DSP-accelerated codec design.

Chapter 9

Future Work

While this project successfully improves LDAC's latency and audio reconstruction quality, several enhancements can extend the system further: **1. Adaptive Bitrate Switching:** LDAC supports multiple operating bitrates (330 kbps, 660 kbps, 990 kbps). A future extension would implement dynamic bitrate switching based on channel conditions, buffer health, and packet loss, enabling more robust wireless streaming. **2. Real-Time Packet Transmission over a Wireless Link:** The current flowgraph processes audio offline. Integrating Bluetooth or SDR-based real-time wireless transmission would allow measurement of practical latency, jitter, and packet loss effects on LDAC performance. **3. Packet Loss Concealment (PLC):** LDAC does not include a built-in PLC module. Implementing concealment strategies (such as waveform repetition, LPC prediction, or MDCT-domain smoothing) would greatly improve robustness under unstable network conditions. **4. GPU or DSP Acceleration:** Many LDAC operations—MDCT, psychoacoustic modeling, Lanczos resampling—are computationally heavy. Offloading them to CUDA/OpenCL or an embedded DSP can enable low-power devices to run LDAC at high bitrates with minimal latency. **5. Improved Psychoacoustic Modeling:** The current implementation uses basic masking models. Future work could integrate more sophisticated auditory masking functions or machine-learning–based masking prediction to allocate bits even more efficiently. **6. Formal Subjective Listening Tests (e.g., MUSHRA):** While qualitative listening tests were performed, formal standardized evaluation would quantify perceptual improvements and provide objective evidence of codec enhancement. **7. Integration with Real-Time Audio Pipelines:** Future work could integrate this optimized LDAC processing into low-latency applications such as wireless monitoring, live performance systems, and gaming audio, validating its usability in practical real-time environments.

Bibliography

[1] GitHub. *wdv4758h/libldac: Unofficial mirror of the LDAC codec from Sony*. Available at: https://github.com/wdv4758h/libldac/tree/wdv4758h

[2] Android Open Source Project. *LDAC Bluetooth Header (ldacBT.h)*. Available at: https://android.googlesource.com/platform/external/libldac/+/refs/heads/main/inc/ldacBT.h

[3] IET Research. *Article on LDAC Technology*. Available at: https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/ell2.13179

[4] Sony Corporation. *LDAC™ — High-Resolution Audio Codec*. Available at: https://www.sony.co.jp/en/Products/LDAC/

Appendix A

Appendix

A.1 Encoder C Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <ldac/ldacBT.h>
typedef struct {
    char riff[4];
    uint32_t chunk_size;
    char wave[4];
    char fmt[4];
    uint32_t subchunk1_size;
    uint16_t audio_format;
    uint16_t num_channels;
    uint32_t sample_rate;
    uint32_t byte_rate;
    uint16_t block_align;
    uint16_t bits_per_sample;
    char data[4];
    uint32_t data_size;
} wav_header_t;
#define TARGET_SAMPLE_RATE 48000
#define TARGET_CHANNELS 2
#define TARGET_BITS 16
#define PCM_BYTES 2
#define BLOCK 128
#define LOOKAHEAD_FRAMES 100
// Improved resampling with sinc interpolation (windowed)
static inline double sinc(double x) {
    if (fabs(x) < 1e-8) return 1.0;
```

```
    return sin(M_PI * x) / (M_PI * x);
}
static inline double lanczos_kernel(double x, int a) {
    if (fabs(x) > a) return 0.0;
    return sinc(x) * sinc(x / a);
}
void resample_audio_improved(int16_t *input, int input_samples, int
    input_rate,
                             int16_t *output, int output_samples, int
                                 output_rate, int channels) {
    double ratio = (double)input_rate / output_rate;
    int a = 3; // Lanczos kernel size

    for (int i = 0; i < output_samples; i++) {
        double src_pos = i * ratio;
        int src_center = (int)round(src_pos);

        for (int ch = 0; ch < channels; ch++) {
            double sum = 0.0;
            double weight_sum = 0.0;

            for (int j = src_center - a; j <= src_center + a; j++) {
                if (j >= 0 && j < input_samples) {
                    double weight = lanczos_kernel(src_pos - j, a);
                    sum += input[j * channels + ch] * weight;
                    weight_sum += weight;
                }
            }

            if (weight_sum > 0.0) {
                output[i * channels + ch] = (int16_t)round(sum /
                    weight_sum);
            } else {
                output[i * channels + ch] = 0;
            }
        }
    }
}
// Convert mono to stereo
void mono_to_stereo(int16_t *mono, int samples, int16_t *stereo) {
    for (int i = 0; i < samples; i++) {
        stereo[i * 2] = mono[i];
        stereo[i * 2 + 1] = mono[i];
    }
}
int main(int argc, char **argv)
{
```

```c
    if (argc < 2) {
        printf("Usage: %s input.wav [output.ldac]\n", argv[0]);
        printf("\nEncodes WAV audio file to LDAC format.\n");
        printf("Input must be 16-bit PCM WAV (any sample rate, mono/stereo).\n");
        printf("Automatically converts to 48kHz/stereo for LDAC encoding.\n");
        printf("If output file is not specified, 'output.ldac' will be used.\n");
        return EXIT_SUCCESS;
    }
    const char *input_file = argv[1];
    const char *output_file = (argc > 2) ? argv[2] : "output.ldac";
    printf("       LDAC Encoder (Improved)\n");
    printf("============================\n");
    printf("Input: %s\n", input_file);
    printf("Output: %s\n\n", output_file);
    FILE *fin = fopen(input_file, "rb");
    if (!fin) {
        fprintf(stderr, "    Cannot open input file: %s\n", input_file);
        perror("Error");
        return EXIT_FAILURE;
    }
    FILE *fout = fopen(output_file, "wb");
    if (!fout) {
        fprintf(stderr, "    Cannot open output file: %s\n", output_file);
        perror("Error");
        fclose(fin);
        return EXIT_FAILURE;
    }
    wav_header_t hdr;
    size_t hdr_read = fread(&hdr, sizeof(hdr), 1, fin);

    if (hdr_read != 1) {
        fprintf(stderr, "    Failed to read WAV header\n");
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }
    if (memcmp(hdr.riff, "RIFF", 4) != 0 || memcmp(hdr.wave, "WAVE", 4) != 0) {
        fprintf(stderr, "    Invalid WAV file format\n");
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
```

```
    }
    printf("       Input format:\n");
    printf(" Sample rate: %d Hz\n", hdr.sample_rate);
    printf(" Bit depth: %d-bit\n", hdr.bits_per_sample);
    printf(" Channels: %d\n", hdr.num_channels);
    printf(" Data size: %u bytes\n", hdr.data_size);

    // Validate 16-bit audio only
    if (hdr.bits_per_sample != 16) {
        fprintf(stderr, "    Only 16-bit PCM audio is supported (got %d-bit)\n", hdr.bits_per_sample);
        fprintf(stderr, " Convert your file with: sox input.wav -b 16 output.wav\n");
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }

    // Handle extended WAV format (skip extra bytes after fmt chunk)
    if (hdr.subchunk1_size > 16) {
        int extra_bytes = hdr.subchunk1_size - 16;
        fseek(fin, extra_bytes, SEEK_CUR);

        // Re-read the data chunk marker
        char data_marker[4];
        uint32_t data_chunk_size;
        fread(data_marker, 1, 4, fin);
        fread(&data_chunk_size, 4, 1, fin);

        if (memcmp(data_marker, "data", 4) == 0) {
            hdr.data_size = data_chunk_size;
            printf(" (Extended format, actual data size: %u bytes)\n", hdr.data_size);
        }
    }

    if (hdr.data_size < (hdr.num_channels * 2)) {
        fprintf(stderr, "    Input file appears to be empty or corrupted (data_size=%u is too small)\n", hdr.data_size);
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }
    printf("\n");
    // Write custom header with original audio parameters for decoder
    // Use packed structure to ensure proper layout
    #pragma pack(push, 1)
```

```
    struct {
        char magic[4];
        uint32_t sample_rate;
        uint32_t channels;
        uint32_t bits_per_sample;
    } header = {
        {'L', 'D', 'A', 'C'},
        hdr.sample_rate,
        hdr.num_channels,
        hdr.bits_per_sample
    };
    #pragma pack(pop)
    if (fwrite(&header, sizeof(header), 1, fout) != 1) {
        fprintf(stderr, "    Failed to write header\n");
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }
    // Verify header was written correctly
    printf("       Header written: %u Hz, %u channels, %u-bit\n",
           header.sample_rate, header.channels, header.bits_per_sample);
    // Check if conversion is needed
    int need_resample = (hdr.sample_rate != TARGET_SAMPLE_RATE);
    int need_channel_convert = (hdr.num_channels != TARGET_CHANNELS);
    if (need_resample || need_channel_convert) {
        printf("       Converting to LDAC format (48kHz/16bit/stereo)...\n");
        if (need_resample) printf("     Resampling: %d Hz     48000 Hz (Lanczos windowed-sinc)\n", hdr.sample_rate);
        if (need_channel_convert) printf("     Channels: %d     2 (stereo)\n", hdr.num_channels);
        printf("\n");
    }
    // Initialize LDAC encoder
    HANDLE_LDAC_BT h = ldacBT_get_handle();
    if (!h) {
        fprintf(stderr, "    ldacBT_get_handle failure\n");
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }
    int mtu = 679; // Minimum recommended MTU (optimized for 2-DH5)
    int eqmid = LDACBT_EQMID_HQ; // 0 = High quality (909/990 kbps)
    int cm = LDACBT_CHANNEL_MODE_STEREO; // Stereo (0x01)
    int fmt = LDACBT_SMPL_FMT_S16; // S16 format (0x2)
```

```
    if (ldacBT_init_handle_encode(h, mtu, eqmid, cm, fmt, TARGET_SAMPLE_RATE) != 0) {
        int err = ldacBT_get_error_code(h);
        fprintf(stderr, "    LDAC encoder initialization failed (error code: %d)\n", err);
        fprintf(stderr, " MTU=%d, EQMID=%d, CM=%d, FMT=%d, SR=%d\n",
                mtu, eqmid, cm, fmt, TARGET_SAMPLE_RATE);
        ldacBT_free_handle(h);
        fclose(fin);
        fclose(fout);
        return EXIT_FAILURE;
    }
    printf("    LDAC encoder initialized\n");
    printf(" Quality: HIGH (EQMID=%d, ~990 kbps @ 48kHz)\n", eqmid);
    printf(" MTU: %d bytes\n", mtu);
    printf(" Output format: 48kHz/16bit/stereo\n\n");
    // Allocate buffers
    int input_frame_size = BLOCK * hdr.num_channels;
    int output_frame_size = BLOCK * TARGET_CHANNELS;

    int16_t *input_buffer = malloc(input_frame_size * sizeof(int16_t));
    int16_t *converted_buffer = malloc(output_frame_size * sizeof(int16_t));
    int16_t *resampled_buffer = NULL;
    uint8_t *ldac_output = malloc(1024);
    if (!input_buffer || !converted_buffer || !ldac_output) {
        fprintf(stderr, "    Memory allocation failed\n");
        goto cleanup;
    }
    if (need_resample) {
        int max_resampled_size = (int)ceil((double)BLOCK * TARGET_SAMPLE_RATE / hdr.sample_rate) + 10;
        resampled_buffer = malloc(max_resampled_size * TARGET_CHANNELS * sizeof(int16_t));
        if (!resampled_buffer) {
            fprintf(stderr, "    Resampling buffer allocation failed\n");
            goto cleanup;
        }
    }
    int total_frames = 0;
    int total_bytes = 0;
    int progress_counter = 0;
    int16_t **lookahead_buffer = malloc(LOOKAHEAD_FRAMES * sizeof(int16_t*));
    for (int i = 0; i < LOOKAHEAD_FRAMES; i++) {
```

```
        lookahead_buffer[i] = malloc(output_frame_size * sizeof(int16_t));
    }
    int buffer_count = 0;
    int buffer_index = 0;
    printf("       Encoding (High-Quality Mode with %d-frame lookahead)...\n", LOOKAHEAD_FRAMES);
    fflush(stdout);
    while (1) {
        // Read 16-bit PCM samples
        size_t samples_read = fread(input_buffer, sizeof(int16_t) * hdr.num_channels, BLOCK, fin);
        if (samples_read == 0) {
            if (feof(fin)) break;
            fprintf(stderr, "\n      Read error\n");
            break;
        }
        // Convert mono to stereo if needed
        int16_t *channel_converted;
        if (hdr.num_channels == 1) {
            mono_to_stereo(input_buffer, samples_read, converted_buffer);
            channel_converted = converted_buffer;
        } else {
            channel_converted = input_buffer;
        }
        // Resample if needed (using improved algorithm)
        int16_t *final_buffer;
        int final_samples;

        if (need_resample) {
            // FIX: Use ceil instead of regular casting to prevent sample loss
            final_samples = (int)ceil((double)samples_read * TARGET_SAMPLE_RATE / hdr.sample_rate);
            resample_audio_improved(channel_converted, samples_read, hdr.sample_rate,
                                    resampled_buffer, final_samples, TARGET_SAMPLE_RATE, TARGET_CHANNELS);
            final_buffer = resampled_buffer;
        } else {
            final_buffer = channel_converted;
            final_samples = samples_read;
        }
        if (final_samples < 2) {
            break;
```

```
        }
        memcpy(lookahead_buffer[buffer_index], final_buffer, final_samples * TARGET_CHANNELS * sizeof(int16_t));
        buffer_index = (buffer_index + 1) % LOOKAHEAD_FRAMES;
        if (buffer_count < LOOKAHEAD_FRAMES) {
            buffer_count++;
            if (samples_read < (size_t)BLOCK) {
                break;
            }
            continue;
        }
        int encode_index = (buffer_index + LOOKAHEAD_FRAMES) % LOOKAHEAD_FRAMES;
        int pcm_used = final_samples;
        int out_bytes = 0;
        int frames = 0;
        int ret = ldacBT_encode(h, lookahead_buffer[encode_index], &pcm_used, ldac_output, &out_bytes, &frames);
        if (ret < 0) {
            fprintf(stderr, "\  n     Encode error: %d\n", ret);
            continue;
        }
        if (out_bytes > 0) {
            size_t written = fwrite(ldac_output, 1, out_bytes, fout);
            if (written != (size_t)out_bytes) {
                fprintf(stderr, "\ n   Write error\n");
                break;
            }
            total_bytes += out_bytes;
            total_frames += frames;
        }
        progress_counter++;
        if (progress_counter % 100 == 0) {
            printf(".");
            fflush(stdout);
        }
        if (samples_read < (size_t)BLOCK) break;
    }
    printf("\  n     Flushing lookahead buffer and encoder...\n");
    for (int i = 0; i < buffer_count; i++) {
        int encode_index = (buffer_index + i) % LOOKAHEAD_FRAMES;
        int pcm_used = BLOCK;
        int out_bytes = 0;
        int frames = 0;
        int ret = ldacBT_encode(h, lookahead_buffer[encode_index], &pcm_used, ldac_output, &out_bytes, &frames);
        if (ret >= 0 && out_bytes > 0) {
```

```
            fwrite(ldac_output, 1, out_bytes, fout);
            total_bytes += out_bytes;
            total_frames += frames;
        }
    }
    for (int i = 0; i < 10; i++) {
        int pcm_used = 0;
        int out_bytes = 0;
        int frames = 0;
        int ret = ldacBT_encode(h, NULL, &pcm_used, ldac_output, &out_bytes, &frames);
        if (ret < 0 || out_bytes <= 0) break;
        size_t written = fwrite(ldac_output, 1, out_bytes, fout);
        if (written != (size_t)out_bytes) {
            fprintf(stderr, "     Flush write error\n");
            break;
        }
        total_bytes += out_bytes;
        total_frames += frames;
    }
    printf("   Encoding completed successfully!\n");
    printf("====================================\n");
    printf("LDAC frames: %d\n", total_frames);
    printf("Output size: %d bytes (+ 16 byte header)\n", total_bytes);

    if (hdr.sample_rate > 0) {
        double input_duration = (double)(hdr.data_size / (hdr.num_channels * hdr.bits_per_sample / 8)) / hdr.sample_rate;
        printf("Duration: %.2f seconds\n", input_duration);
        printf("Average bitrate: ~%.0f kbps\n", (total_bytes * 8.0) / input_duration / 1000.0);
    }

    printf("Output saved: %s\n", output_file);
cleanup:
    ldacBT_close_handle(h);
    ldacBT_free_handle(h);
    if (input_buffer) free(input_buffer);
    if (converted_buffer) free(converted_buffer);
    if (resampled_buffer) free(resampled_buffer);
    if (ldac_output) free(ldac_output);
    if (lookahead_buffer) {
        for (int i = 0; i < LOOKAHEAD_FRAMES; i++) {
            if (lookahead_buffer[i]) free(lookahead_buffer[i]);
        }
        free(lookahead_buffer);
    }
```

```
    fclose(fin);
    fclose(fout);
    return EXIT_SUCCESS;
}
```

Listing A.1: Optimized LDAC Encoder (C)

A.2 Decoder C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <sndfile.h>
#include "libldacdec/ldacdec.h"
#define BUFFER_SIZE (65536)
#define PCM_BUFFER_SIZE (MAX_FRAME_SAMPLES * 2)
#define DEBUG_LEVEL 3
#define MIN_FRAME_SIZE 1000
#define MAX_FRAME_SIZE 3000
#define MAX_CONSECUTIVE_ERRORS 3
#define OUTPUT_DELAY_FRAMES 150
// For audio smoothing
#define CROSSFADE_SIZE 128
#define NOISE_THRESHOLD 0.01f
#define NOISE_GATE_RATIO 0.5f
#define DEBUG_PRINT(level, ...) \
    do { if (DEBUG_LEVEL >= level) fprintf(stderr, __VA_ARGS__); }
        while (0)
// FIXED: Header format - MUST match encoder exactly
#pragma pack(push, 1)
struct LDACHeader {
    char magic[4]; // "LDAC" - 4 bytes
    uint32_t sampleRate; // Original sample rate - 4 bytes
    uint32_t channels; // Original channels - 4 bytes
    uint32_t bits_per_sample; // Original bit depth - 4 bytes
}; // Total: 16 bytes exactly
#pragma pack(pop)
// Structure to store previous frame data
typedef struct {
    float *samples;
    int sample_count;
    int channels;
} PreviousFrame;
```

```
// Smooth transition between frames
static void apply_crossfade(float *current, float *previous, int
    samples, int channels) {
    if (!previous || samples <= 0) return;
    for (int i = 0; i < CROSSFADE_SIZE && i < samples; i++) {
        float fade_in = (float)i / CROSSFADE_SIZE;
        float fade_out = 1.0f - fade_in;
        for (int ch = 0; ch < channels; ch++) {
            int idx = i * channels + ch;
            current[idx] = current[idx] * fade_in + previous[idx] *
                fade_out;
        }
    }
}
// Enhanced PCM to float conversion with noise reduction
static void pcm_to_float(const int16_t *pcm, float *output, int samples
    , int channels) {
    static float prev_samples[2] = {0.0f, 0.0f};
    static float env[2] = {0.0f, 0.0f};
    const float attack = 0.001f;
    const float release = 0.1f;
    for (int i = 0; i < samples; i++) {
        for (int ch = 0; ch < channels; ch++) {
            float sample = pcm[i * channels + ch] / 32768.0f;
            // Update envelope
            float abs_sample = fabs(sample);
            if (abs_sample > env[ch]) {
                env[ch] = env[ch] * (1.0f - attack) + abs_sample *
                    attack;
            } else {
                env[ch] = env[ch] * (1.0f - release) + abs_sample *
                    release;
            }
            // Adaptive noise gate
            float threshold = NOISE_THRESHOLD * (1.0f + env[ch]);
            if (abs_sample < threshold) {
                float ratio = powf(abs_sample / threshold,
                    NOISE_GATE_RATIO);
                sample *= ratio;
            }
            // Simple DC removal and smoothing
            float filtered = sample - prev_samples[ch] + 0.995f *
                prev_samples[ch];
            prev_samples[ch] = filtered;
            output[i * channels + ch] = filtered;
        }
    }
```

```
}
// Simple resampling with linear interpolation
static void resample_audio(float *input, int input_samples, int input_rate,
                           float *output, int output_samples, int output_rate, int channels) {
    double ratio = (double)input_rate / output_rate;

    for (int i = 0; i < output_samples; i++) {
        double src_pos = i * ratio;
        int src_idx = (int)src_pos;
        double frac = src_pos - src_idx;

        for (int ch = 0; ch < channels; ch++) {
            if (src_idx + 1 < input_samples) {
                float s0 = input[src_idx * channels + ch];
                float s1 = input[(src_idx + 1) * channels + ch];
                output[i * channels + ch] = s0 + (s1 - s0) * frac;
            } else if (src_idx < input_samples) {
                output[i * channels + ch] = input[src_idx * channels + ch];
            } else {
                output[i * channels + ch] = 0.0f;
            }
        }
    }
}
static int validate_frame_header(const uint8_t* buffer, size_t buffer_len, int* frame_size) {
    if (buffer_len < 3) return 0;
    if (buffer[0] != 0xAA) return 0;
    uint8_t frame_header = buffer[1];
    *frame_size = ((frame_header & 0x0F) << 8) | buffer[2];
    if (*frame_size < MIN_FRAME_SIZE || *frame_size > MAX_FRAME_SIZE) {
        return 0;
    }
    return 1;
}
static size_t find_next_frame(const uint8_t* buffer, size_t buffer_len, size_t start_offset, int* frame_size) {
    for (size_t i = start_offset; i < buffer_len - 3; i++) {
        if (buffer[i] == 0xAA) {
            if (validate_frame_header(buffer + i, buffer_len - i, frame_size)) {
                return i;
            }
        }
```

```
    }
    return buffer_len;
}
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input.ldac> <output.wav>\n", argv
            [0]);
        return EXIT_FAILURE;
    }
    const char* input_file = argv[1];
    const char* output_file = argv[2];
    FILE* fin = fopen(input_file, "rb");
    if (!fin) {
        DEBUG_PRINT(1, "Cannot open input file: %s\n", input_file);
        return EXIT_FAILURE;
    }
    fseek(fin, 0, SEEK_END);
    long file_size = ftell(fin);
    fseek(fin, 0, SEEK_SET);
    // Read header
    struct LDACHeader header;
    size_t header_read = fread(&header, 1, sizeof(header), fin);
    if (header_read != sizeof(header) || memcmp(header.magic, "LDAC",
        4) != 0) {
        DEBUG_PRINT(1, "Invalid LDAC header (read %zu bytes, expected %
            zu)\n",
                    header_read, sizeof(header));
        fclose(fin);
        return EXIT_FAILURE;
    }
    // Validate and get original audio parameters
    uint32_t original_sample_rate = header.sampleRate;
    uint32_t original_channels = header.channels;
    uint32_t original_bits = header.bits_per_sample;
    if (original_sample_rate < 8000 || original_sample_rate > 192000) {
        DEBUG_PRINT(1, "Invalid sample rate: %u (using 44100)\n",
            original_sample_rate);
        original_sample_rate = 44100;
    }
    if (original_channels < 1 || original_channels > 2) {
        DEBUG_PRINT(1, "Invalid channels: %u (using 2)\n",
            original_channels);
        original_channels = 2;
    }
    if (original_bits != 16 && original_bits != 24 && original_bits !=
        32) {
```

```c
        DEBUG_PRINT(1, "Invalid bit depth: %u (using 16)\n", original_bits);
        original_bits = 16;
    }
    printf("LDAC file info:\n");
    printf(" Original sample rate: %u Hz\n", original_sample_rate);
    printf(" Original channels: %u\n", original_channels);
    printf(" Original bit depth: %u\n", original_bits);
    // Initialize decoder
    ldacdec_t decoder;
    memset(&decoder, 0, sizeof(decoder));
    if (ldacdecInit(&decoder) != 0) {
        DEBUG_PRINT(1, "Failed to initialize decoder\n");
        fclose(fin);
        return EXIT_FAILURE;
    }
    // Create output WAV with ORIGINAL parameters (not 48kHz!)
    SF_INFO sfinfo = {0};
    sfinfo.samplerate = original_sample_rate;
    sfinfo.channels = original_channels;
    sfinfo.format = SF_FORMAT_WAV | SF_FORMAT_PCM_16;
    SNDFILE *fout = sf_open(output_file, SFM_WRITE, &sfinfo);
    if (!fout) {
        DEBUG_PRINT(1, "Cannot create output file: %s\n", sf_strerror(NULL));
        fclose(fin);
        return EXIT_FAILURE;
    }
    // Allocate buffers
    uint8_t *buffer = (uint8_t*)malloc(BUFFER_SIZE);
    int16_t *pcm = (int16_t*)malloc(PCM_BUFFER_SIZE * sizeof(int16_t));
    float *float_buffer = (float*)malloc(PCM_BUFFER_SIZE * sizeof(float));
    float *resample_buffer = NULL;

    // Allocate resampling buffer if needed
    int need_resample = (original_sample_rate != 48000);
    if (need_resample) {
        resample_buffer = (float*)malloc(PCM_BUFFER_SIZE * 2 * sizeof(float));
    }
    PreviousFrame prev_frame = {0};
    prev_frame.samples = (float*)calloc(PCM_BUFFER_SIZE, sizeof(float));
    prev_frame.channels = original_channels;
    if (!buffer || !pcm || !float_buffer || !prev_frame.samples ||
        (need_resample && !resample_buffer)) {
```

```c
        DEBUG_PRINT(1, "Failed to allocate buffers\n");
        goto cleanup;
    }
    size_t data_size = file_size - sizeof(header);
    size_t total_read = 0;
    size_t buffer_len = 0;
    size_t offset = 0;
    buffer_len = fread(buffer, 1, BUFFER_SIZE, fin);
    total_read = buffer_len;
    int frame_count = 0;
    int error_count = 0;
    int consecutive_errors = 0;
    float **output_delay_buffer = malloc(OUTPUT_DELAY_FRAMES * sizeof(float*));
    for (int i = 0; i < OUTPUT_DELAY_FRAMES; i++) {
        output_delay_buffer[i] = calloc(PCM_BUFFER_SIZE, sizeof(float));
    }
    int *delay_sample_counts = calloc(OUTPUT_DELAY_FRAMES, sizeof(int));
    int delay_write_index = 0;
    int delay_count = 0;
    printf("Starting decode (High-Quality Mode with %d-frame output delay)...\n", OUTPUT_DELAY_FRAMES);
    while (offset < buffer_len) {
        if (offset + MAX_FRAME_SIZE > buffer_len) {
            if (total_read >= data_size) break;
            size_t remaining = buffer_len - offset;
            memmove(buffer, buffer + offset, remaining);
            size_t new_bytes = fread(buffer + remaining, 1, BUFFER_SIZE - remaining, fin);
            buffer_len = remaining + new_bytes;
            total_read += new_bytes;
            offset = 0;
            if (new_bytes == 0) break;
            continue;
        }
        int frame_size = 0;
        size_t next_frame = find_next_frame(buffer, buffer_len, offset, &frame_size);
        if (next_frame != offset) {
            if (next_frame >= buffer_len - 3) {
                offset = buffer_len - 3;
                continue;
            }
            offset = next_frame;
        }
```

```
        if (offset + frame_size > buffer_len) continue;
        memset(pcm, 0, PCM_BUFFER_SIZE * sizeof(int16_t));
        int bytes_used = 0;
        int result = ldacDecode(&decoder, buffer + offset, pcm, &
           bytes_used);
        if (result < 0 || bytes_used <= 0) {
            consecutive_errors++;
            error_count++;
            if (consecutive_errors > MAX_CONSECUTIVE_ERRORS) {
                offset++;
                continue;
            }
            if (prev_frame.sample_count > 0) {
                memcpy(float_buffer, prev_frame.samples,
                       prev_frame.sample_count * original_channels *
                          sizeof(float));
                float fade = 1.0f - (float)consecutive_errors /
                   MAX_CONSECUTIVE_ERRORS;
                for (int i = 0; i < prev_frame.sample_count *
                   original_channels; i++) {
                    float_buffer[i] *= fade;
                }
                sf_writef_float(fout, float_buffer, prev_frame.
                   sample_count);
            }
            offset += bytes_used > 0 ? bytes_used : 1;
            continue;
        }
        consecutive_errors = 0;
        int samples = decoder.frame.frameSamples;
        if (samples > 0 && samples <= MAX_FRAME_SAMPLES) {
            // Convert to float with noise reduction
            pcm_to_float(pcm, float_buffer, samples, original_channels)
               ;
            // Apply crossfade
            if (prev_frame.sample_count > 0) {
                apply_crossfade(float_buffer, prev_frame.samples,
                   samples, original_channels);
            }
            // RESAMPLE back to original rate if needed
            float *output_buffer = float_buffer;
            int output_samples = samples;

            if (need_resample) {
                // Decoder outputs at 48kHz, resample to original rate
                output_samples = (int)((double)samples *
                   original_sample_rate / 48000.0);
```

```
                resample_audio(float_buffer, samples, 48000,
                             resample_buffer, output_samples, original_sample_rate,
                             original_channels);
                output_buffer = resample_buffer;
            }
            memcpy(prev_frame.samples, float_buffer, samples * original_channels * sizeof(float));
            prev_frame.sample_count = samples;
            memcpy(output_delay_buffer[delay_write_index], output_buffer,
                   output_samples * original_channels * sizeof(float));
            delay_sample_counts[delay_write_index] = output_samples;
            delay_write_index = (delay_write_index + 1) % OUTPUT_DELAY_FRAMES;
            if (delay_count < OUTPUT_DELAY_FRAMES) {
                delay_count++;
                frame_count++;
                offset += bytes_used;
                continue;
            }
            int read_index = (delay_write_index + OUTPUT_DELAY_FRAMES) % OUTPUT_DELAY_FRAMES;
            sf_count_t written = sf_writef_float(fout, output_delay_buffer[read_index],
                                                 delay_sample_counts[read_index]);
            if (written == delay_sample_counts[read_index]) {
                frame_count++;
                offset += bytes_used;
            } else {
                DEBUG_PRINT(1, "Write error\n");
                break;
            }
        } else {
            offset++;
        }
    }
    printf("Flushing output delay buffer...\n");
    for (int i = 0; i < delay_count; i++) {
        int read_index = (delay_write_index + i) % OUTPUT_DELAY_FRAMES;
        sf_writef_float(fout, output_delay_buffer[read_index], delay_sample_counts[read_index]);
    }
    printf("SUCCESS: Decoded %d frames (errors: %d)\n", frame_count, error_count);
```

```c
    double latency_ms = (OUTPUT_DELAY_FRAMES * 128.0 / 48000.0) * 1000.0;
    printf("       Decoder Latency: %.2f ms (%d frames buffered)\n", latency_ms, OUTPUT_DELAY_FRAMES);
cleanup:
    sf_close(fout);
    free(buffer);
    free(pcm);
    free(float_buffer);
    free(resample_buffer);
    free(prev_frame.samples);
    free(delay_sample_counts);
    if (output_delay_buffer) {
        for (int i = 0; i < OUTPUT_DELAY_FRAMES; i++) {
            if (output_delay_buffer[i]) free(output_delay_buffer[i]);
        }
        free(output_delay_buffer);
    }
    fclose(fin);
    return frame_count > 0 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing A.2: Optimized LDAC Decoder (C)

A.3 README / Usage

Usage examples:

- Encoder: `./encoder input.wav output.ldac`
- Decoder: `./decoder input.ldac output.wav`
- GNU Radio basic: open `test_ldac.py` in GNU Radio Companion
- GNU Radio optimized: open `test_ldac_ultimate.py` in GNU Radio Companion