18-551, Spring 2003
Group 10, Final Report:

Adaptive Noise Cancellation System using Subband LMS

Prasanna Malaiyandi (pkm@andrew.cmu.edu)
David Mitchell (dwm3@andrew.cmu.edu)
Samir Sahu (ssahu@andrew.cmu.edu)

1.1 Table of Contents

2.1 Introduction

In our increasingly mobile society, individuals are prone to doing just about everything on the move. Listening to music is certainly not an exception. However, when one listens to music away from the home, one necessarily has less control over noise exposure. Airplane, bus and car engines are the most common noise distractions as one travels. Lawnmower engines, others' speech and music are also frequently encountered. Surely, there is considerable benefit in obtaining headphones that could perform active noise cancellation – be able to filter out noise as one encounters it.

Large electronics manufacturers have not ignored this need. Indeed, there are several products on the market, the most notable of which – Sony® MDR-NC20 Noise Canceling Closed Headphones and Bose® QuietComfort™ Acoustic Noise Canceling® Headphones – are discussed in section 2.3 in greater detail. Both products' noise attenuation capability is advertised as up to 10 dB for frequencies below 300 Hz. However, frequency analysis of numerous real-life noise signals – even car and airplane engines – reveals significant noise (up to 30 dB in a Boeing 747[1] and up to 60 dB in the cockpit of a Cessna 210[2]) at up to 3 kHz.

Thus, the products' peak attenuation is limited to frequencies below 300 Hz notwithstanding the considerable desirability of a more comprehensive solution.

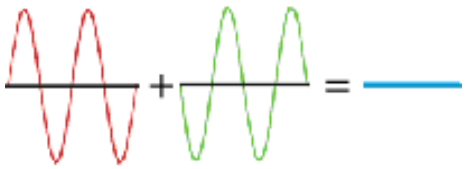
[1] Boeing planes are sheltered with noise-absorbing coating which reduces the noise present.
[2] Cessna 210 is a small, popularly owned private plane. If medium-frequency noise attenuating precautions are not taken, frequent pilots have up to 41% chance of developing permanent hearing damage according to US EPA Report 550/9-73-008.

Although it was not feasible to determine the reason for the limitation as neither Sony nor Bose has revealed the algorithm used in the companies' respective products, it seems realistic that existing solutions are not effective outside the low-frequency range due to some processing constraint. The limitations on a single-band Least-Mean-Square (LMS) algorithm as established by Siravara, et al. in 2002 (hereafter [1]) coincide with product constraints as advertised. The proposed improvement – the new subband LMS algorithm examined in section 2.6 – facilitates significant improvement in medium-frequency noise attenuation while reducing the computing resources needed to update the adaptive filter.

Although subband LMS is potentially a promising alternative, as of this writing, there are no publicly accessible reports regarding a headphone implementation of the algorithm. Under these circumstances, writing a DSP implementation of the algorithm and creating an active-feedback headphone system may be tangible contributions to the active feedback Adaptive Noise Cancellation (ANC) field.

2.2 Active Feedback ANC

Adaptive Noise Cancellation (ANC) is a widely applicable set of noise attenuating techniques. Unlike simple filtering, ANC techniques attenuate noise through the addition of an "anti-noise" signal with 180-degree phase difference, thereby dampening the energy of the noise waves. Active feedback via an embedded microphone facilitates targeted noise cancellation without any requisite a priori knowledge about the signal transmitted or the noise present. There are several algorithms used to calculate the "anti-noise" signal. Wideband (single band) and subband (2 or more bands) Least Mean Square (LMS) algorithms are analyzed in sections 2.4 and 2.6 respectively.



Fig, 1: Destructive Interference

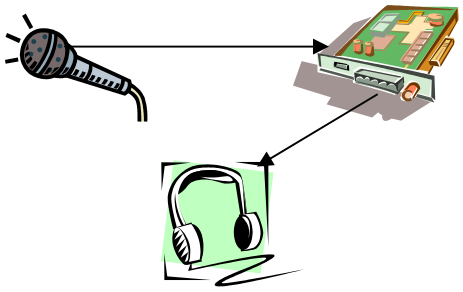


Fig. 2: Single channel active feedback with microphone, headphones, EVM

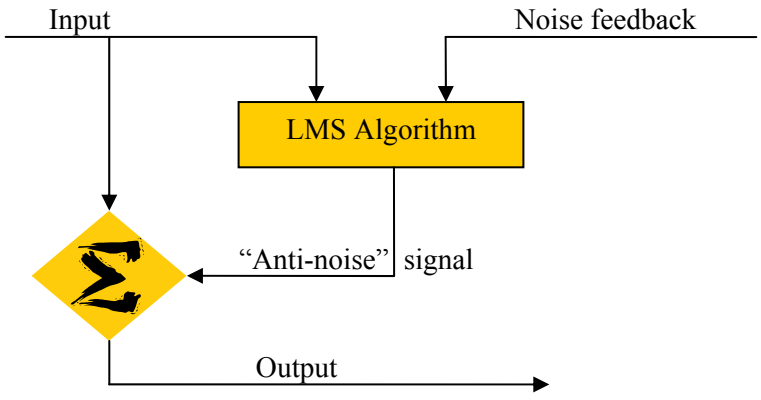


Fig. 3: Sample ANC flowchart with LMS

2.3 Commercial Products

Sony® MDR-NC20 Noise Canceling Closed

MSRP: $199

Attenuation up to 10 dB for frequencies up to 300 Hz

Algorithm: wideband LMS [1]

Bose® QuietComfort™ Acoustic Noise Canceling®

MSRP: $299

Attenuation up to 10 dB for frequencies up to 300 Hz

Algorithm: unknown; wideband LMS likely

2.4 LMS

The LMS algorithm is comprised of two processes – a filtering process producing the output signal and the estimation error, and an adaptive process responsible for the automatic adjustment of filter tap weights. The following definitions and notations will be used throughout:

Input signal: $\mathbf{u}(n)$
Desired signal: $\mathbf{d}(n)$
Filter tap weights: $\mathbf{w}(n)$
Filter output:

$$\mathbf{y}(n) = \sum_{k=0}^{M-1} \mathbf{w}_k^* \mathbf{u}(n-k) = \mathbf{w}^H(n)\mathbf{u}(n)$$

Estimation error:

$$\mathbf{e}(n) = \mathbf{d}(n) - \mathbf{y}(n)$$

Instantaneous error approximations:

$$\hat{\mathbf{R}}(n) = \mathbf{u}(n)\mathbf{u}^H(n)$$

$$\hat{\mathbf{p}}(n) = \mathbf{u}(n)\mathbf{d}^*(n)$$

The LMS algorithm is obtained by substituting the instantaneous error approximations into the basic steepest-descent Weiner filter algorithm.

$$\begin{aligned}
\mathbf{w}(n+1) &= \mathbf{w}(n) + \frac{1}{2}\mu[\hat{\mathbf{p}} - \hat{\mathbf{R}}\,\mathbf{w}(n)] && \text{Steepest descent with error} \\
&\approx \mathbf{w}(n) + \mu[\mathbf{u}(n)\mathbf{u}^H(n)\mathbf{w}(n)] \\
&= \mathbf{w}(n) + \mu\,\mathbf{u}(n)[\mathbf{d}^*(n) - \mathbf{y}^*(n)] \\
&= \mathbf{w}(n) + \mu\,\mathbf{u}(n)\mathbf{e}^*(n) && \text{LMS filter coefficient adjustment}
\end{aligned}$$

In the preceding definition, μ – the step size of the algorithm – is essentially the driving factor in the coefficient adjustment. The LMS algorithm can be mathematically shown to converge when $0 < \mu < \frac{2}{\lambda_{max}}$ where $\lambda_{max} < \sum_{i=1}^{M} \lambda_i = E[\mathbf{u}^H(n)\mathbf{u}(n)]$. Siravara et al. show that LMS μ values for practical adaptive noise canceling applications range between 0.0002 and 0.04 [1]. Moreover, Principe et al. propose a $\mu = \frac{\lambda_{max}}{10}$ rule of thumb resulting in speedy and accurate convergence suitable for most applications [2].

2.5 Prior CMU Projects

Spring 1999: Group 6

Ormsby et al. followed a wideband LMS approach in a project titled *Noise Canceling Headphones: An Adaptive Solution*. The group demonstrated significant noise attenuation for some music signals in Matlab with a 64-tap LMS filter. The results were comparable to expected headset performance.

Moreover, Ormsby, et al. determined that real-time attenuation between 7 and 10 dB required an LMS filter size of between 128 and 512 taps. Initially, the group attempted to implement a 256-tap solution with 8 kHz signal on the C30 EVM but could not achieve real-time noise attenuation. A 16-tap wideband LMS filter processing an 8 kHz single-channel (mono) signal was proven to require fewer EVM instruction cycles than were permissible between sample inputs, yet noise attenuation on the EVM was not demonstrated.

Although, some noise canceling is possible with a 16-tap wideband LMS, a 2-band 16-tap algorithm allows up to double attenuation and more rapid convergence. Thus, barely audible attenuation of ~4dB is improved to cancel 60-70% of the noise frequency power. Using the subband approach on the C67 EVM it becomes possible to construct a stereo ANC system capable of processing 44.1 kHz, CD-quality signals.

2.6 Subband LMS

The newly proposed subband algorithm calls for parsing the desired and feedback noise signals into at least two bands, running the LMS algorithm on each band and finally combining the individual band outputs into a single noise-canceling signal.

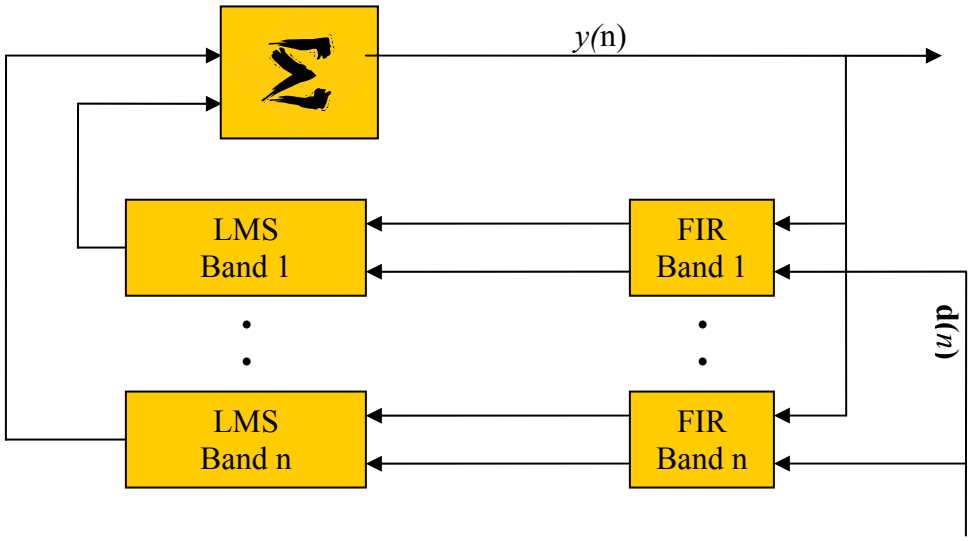


Fig. 4: Subband LMS

This approach facilitates faster convergence with smaller filter size – $\mathbf{w}(n)$ – while increasing the maximum noise attenuation possible for constant μ. These advantages enable significant real-time noise attenuation for a 44.1 kHz – CD quality – signal on the C67 EVM board.

Frequency of Primary Noise Signal	Wide-band noise attenuation	Subband noise attenuation
0-250 Hz	14.4dB	**16dB**
500-750Hz and 1250-1500Hz	5.66dB	**10.03dB**
750-1000Hz and 2000-2250Hz	6.31dB	**11.9dB**
3000-3250Hz	**9.7dB**	**15.79dB**

Table 1: Subband advantages according to Siravara et al. [1]

The most tangible advantages are for medium-frequency noise with $500\text{Hz} < f < 3250\text{Hz}$ and thus are precisely in the underperforming frequency range targeted for needed improvement.

3.1 Project Design

We propose to demonstrate the effectiveness of the subband LMS algorithm for a real-time, active feedback ANC system. The stereo active feedback will be performed by two microphones (one for each ear).

Koss UR/15 Personal listening headphones

MSRP: $29.99

Selected for closed ear design, wide frequency response ($25\text{Hz} < f < 15000\text{Hz}$), low distortion and reasonable price

RadioShack Tie-Clip Omni directional Electret

MSRP: $24.99

Selected for small size, wide frequency response ($50\text{Hz} < f < 16000\text{Hz}$), low impedance and reasonable price.

In order to follow through with the proposal, the authors must (1) establish the subband proof of concept, (2) determine real-time, sample-by-sample update capability, and (3) demonstrate significant noise attenuation across the frequency spectrum for CD quality music in real-time on the C67 EVM board. Task (1) is most readily accomplished in Matlab, task (2) in C and task (3) involves the EVM board.

3.2 Matlab

Determination of μ:

Using the rule of thumb suggested by Principe et al., μ was calculated from a 10-sample average of the max eigenvalue [2]. Frequency eigenvalues were calculated for various song inputs ranging from Mozart to Eminem, with the average $\lambda_{max} = 0.25$, $\mu = \frac{\lambda_{max}}{10} = 0.025$. Thereafter, the calculated μ was tested alongside the 0.0002 to 0.04 limits [1] in Matlab for LMS convergence and attenuation quality.

μ	Iterations until max attenuation (for Gaussian noise)	Limitation on final quality $> 10^{-5}$?
0.0002	~60,000	No
0.0100	~8,000	No
0.0250	***~3,000***	***No***
0.0400	~2,400	No
0.1000	~1650	Yes

Table 2: μ-value comparison based on Matlab experimentation

Setting μ = 0.025 is confirmed to be reasonable with significantly more rapid attenuation as compared to much lower values and essentially trivial loss of final accuracy.

Matlab code implementation:

Using ANSI C code for wideband LMS from Texas Instruments [3], we implemented wideband and 2-band Matlab LMS solutions. The initial transformation was iteratively intensive and required almost 10 minutes to process 15 seconds of 8 kHz signals. The Matlab code was thereafter optimized to perform more matrix calculations instead of loop iteration. Most importantly, the TI implementation of LMS was modified to calculate **e**(n) prior to updating the **w(*n*)** vector. This algorithmic change facilitates the transition

to sample-by-sample processing as compared to the TI code, which uses buffers. Adding code for 32-tap FFT and IFFT, we successfully created and tested a 2-band implementation. The Matlab code is located in Appendix A at the end of this report.

Proof of Subband efficacy

The wideband and 2-band implementations were tested under identical conditions. All signals were sampled at 44.1 kHz.

Desired signal: first 2 minutes of Mozart's 4th Concerto

Noise signal one: $\eta_1 = .2\sin(2\pi(200)) + .3\sin(2\pi(500)) + .4\sin(2\pi(900))$

Noise signal two: $\eta_2 = .5\sin(2\pi(1300)) + .1\sin(2\pi(2200)) + .2\sin(2\pi(3100))$

Noise signal three: $\eta_3 = \eta_1 + \eta_2$

First band: $f \le 1kHz$ Second band: $1kHz < f \le 4kHz$

When η_1 or η_2 were tested, the algorithms performed essentially in tandem because the two noise signals were chosen to have noise components corresponding to each of the two sub bands respectively. However, for η_3, which contained noise in both frequency ranges, the 2-band algorithm attenuated noticeably faster and achieved significantly higher final noise reduction. The estimation error – $\mathbf{e}(n) = \mathbf{d}(n) - \mathbf{y}(n)$ – for the last 220500 samples (5 seconds of the clip) was analyzed in GoldWave to determine the amplitudes associated with each noise frequency.

For each frequency, the achieved attenuation was calculated as follows:

$NdB = -20\log(\frac{A_r}{A_i})$ $A_r \rightarrow$ average residual amplitude $A_i \rightarrow$ known initial amplitude

Frequency Noise Signal	Final wideband noise attenuation	Final 2-band noise attenuation
200 Hz	12.1 dB	**13.3 dB**
500 Hz	6.9 dB	**10.8 dB**
900 Hz	5.3 dB	**7.5 dB**
1300 Hz	7.1 dB	**10.4 dB**
2200 Hz	5.4 dB	**7.7 dB**
3100 Hz	10.7 dB	**12.9 dB**

Table 3: Matlab wideband vs. 2-band attenuation

3.3 C

C algorithm:

In [3], TI provides assembly code for a wideband LMS. Also included is the ANSI C version of the assembly functionality. However, the C code does not correspond to the ASM functionality due to some flaw that was not discovered during debugging. Instead, a modified version of the LMS algorithm was written in C. Key modifications include calculating **e**(n) prior to updating the **w(*n*)** vector and processing the LMS algorithm after each sample (as opposed to each 16-sample chunk). FFT and IFFT with 16 coefficients from 18-551 Homework 2A were used to set up the two frequency bands. The C code is located in Appendix B.

Realistic signal testing:

Minor code optimization reduced the running time of the C code to function real time for real audio and noise signals. Realistic noise signals provided by http://www.exhaustsoundclips.com [4] and http://physics.nku.edu/asg/noisesamples.html [5] were introduced to test one of the algorithm's primary applications – engine noise reduction. The exhaust of a 1967 Ford Mustang and aircraft noise at the Cincinnati/Northern Kentucky International Airport were analyzed using GoldWave, having substantial noise components for $0 < f \leq 4kHz$. This range covers most commonly experienced, locally periodic noise signals and confirms the original choice for band definition. Eminem's *Without Me* raw file (converted from mp3 using GoldWave) was the desired signal. The following figure contains the waveforms played during the final oral update.

Attenuation results:

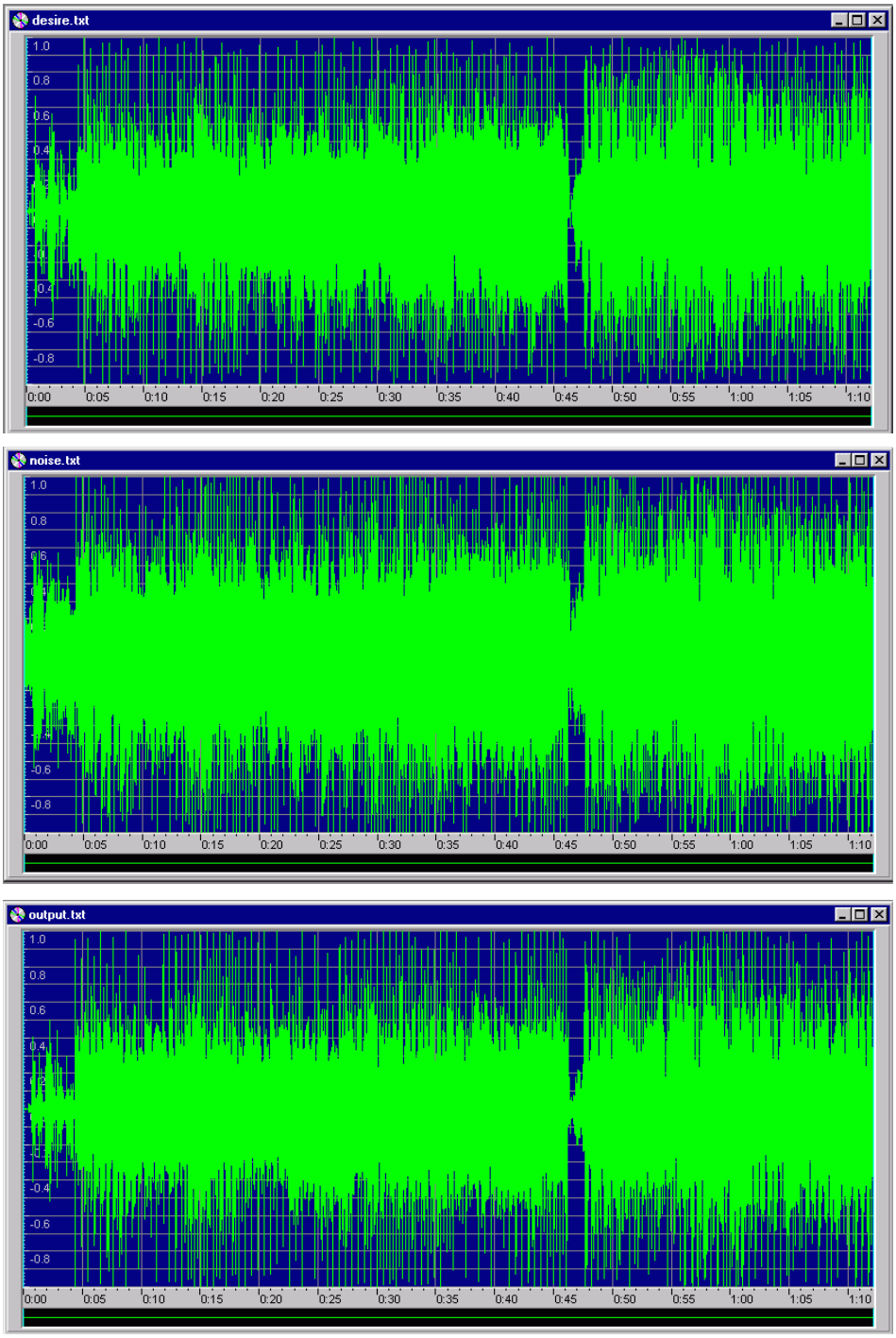


Fig. 5: Real-time 2-band LMS noise attenuation. Top is **d(*n*) –** desired signal; middle is **u(*n*) –** input signal; bottom is **y(*n*) –** filter output

Algorithm efficacy is readily observed graphically in Figure 5 between 0:00 and 0:05 and 0:45 and 0:48 where the input signal noise clearly overrides the desired signal. Filter output demonstrates strong attenuation and is almost indiscernible from desired output. Strong attenuation is thus possible even in the first few milliseconds, demonstrating the advantage of the 2-band system over a wideband solution.

As detailed in the previous section, decibel attenuation is determined through amplitude vs. frequency analysis of the signals' last 5 seconds. Because the initial amplitude was not user defined, a running average was computed for the last 5 seconds of the noise signal. Thereafter, the following formula was used to compute the attenuation.

$$NdB = -20\log(\frac{A_r}{A_i})$$

$A_r \rightarrow$ average residual amplitude $A_i \rightarrow$ average initial amplitude

Frequency of Noise Signal	2-band Noise Attenuation
0-1 kHz	**~13dB**
1-4 kHz	**~7dB**

Table 4: 2-Band attenuation by frequency range

3.4 EVM

To port the C code from the previous section to the C67 EVM, 18-551 Lab 1 interrupt system was adapted to the current needs. All processing code (FFT/IFFT, LMS) was placed into receiveISR. The data is then readily transmitted with a single call in transmitISR. Using the sample-by-sample processing methodology and a CD quality signal, the EVM code can comprise at most $166MHz / 44.1kHz \approx 3800$ cycles. Although rough, preliminary calculations indicated only $4*640+1000=3560$ [1] cycles necessary for the FFT/IFFT approach, the actual implementation required more and therefore did not work real time.

The solution was to replace the FFT/IFFT with 32-tap FIR band-pass filters. Matlab functions firls and remez generated the filter coefficients used. The FIR filters performed sufficiently, with negligible performance error as compared to the FFT/IFFT method. Three filters were implemented – low-pass for $f \le 1kHz$, band-pass for $1kHz < f \le 4kHz$, and high-pass for $f > 4kHz$. The low-pass and band-pass filters created the bands to be processed via the LMS. The high frequency component is passed to the headphones. Much of the noise above this threshold is either negligible in amplitude or inaudible. Indeed, there is a very limited advantage to dampening it yet a tangible drawback with extra requisite cycles [6]. This current implementation is described in detail in Figures 6 and 7 on the following page. The EVM code itself is located in Appendix C.

[1] 640 cycles for each FFT and IFFT. For 2 bands, that is 2 FFTs and 2 IFFTs. 1000 cycles for 2 16-tap LMS updates, error processing, signal addition and overhead

Active feedback processing:

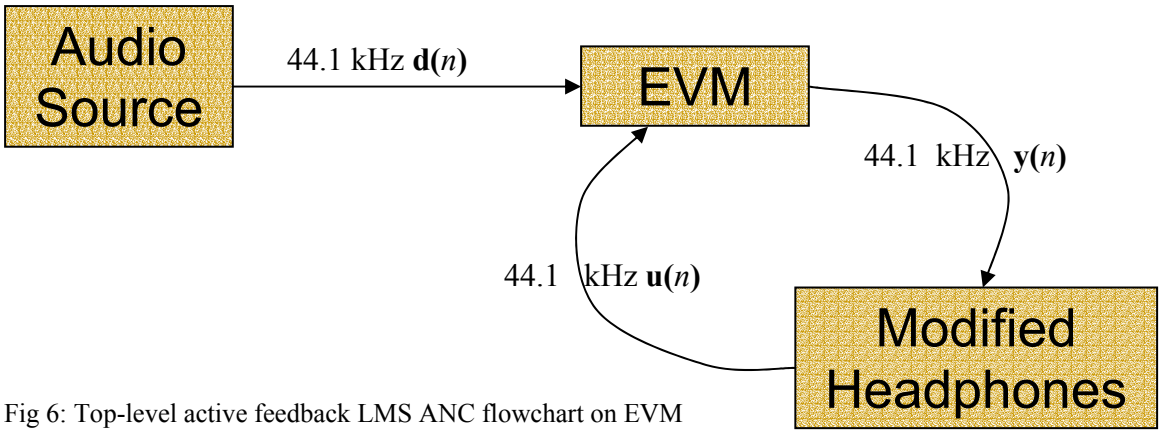


Fig 6: Top-level active feedback LMS ANC flowchart on EVM

As the input **u**(*n*) and **d**(*n*) arrive for each n, the EVM performs the following:

1. Desired **d**(*n*) and noise feedback **u**(*n*) signals are summed.
2. The combined signal is divided into three bands
3. The high frequencies ($f > 4kHz$) are passed through to the headphones
4. The lower two bands ($f \le 1kHz$, $1kHz < f \le 4kHz$) are processed via the LMS algorithm (which also receives **d**(*n*))
5. The LMS outputs are summed and **y**(*n*) is passed to the headphones

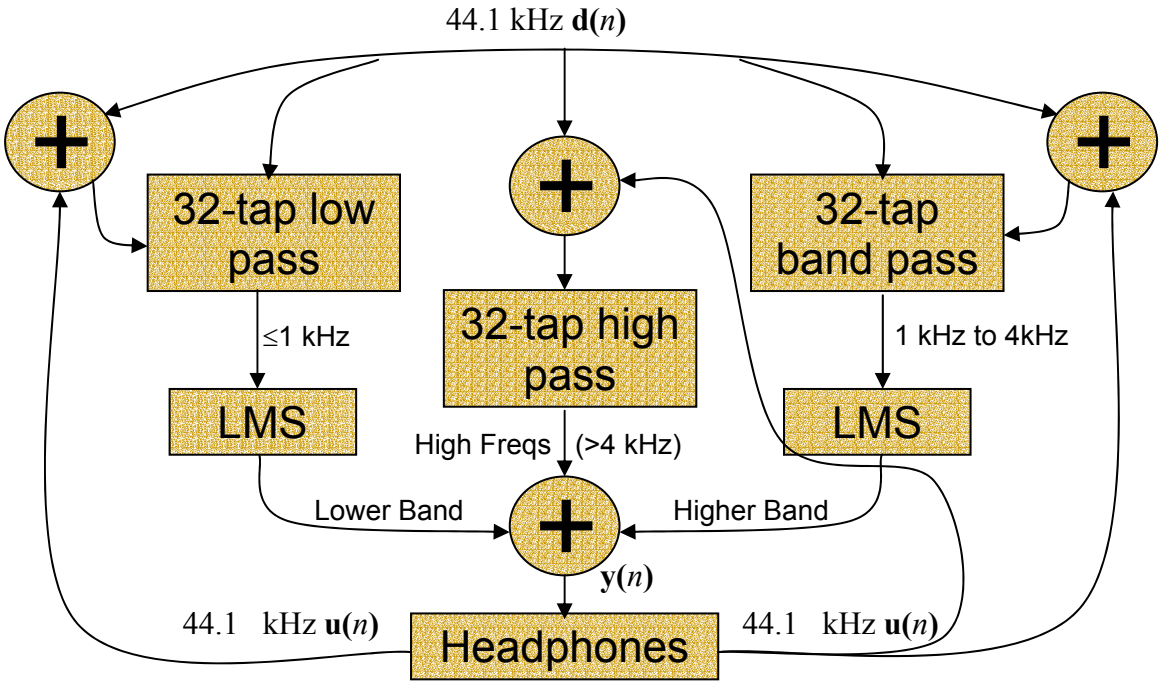


Fig 7: Low-level active feedback LMS ANC flowchart on EVM

Testing:

This "real-time", active feedback LMS ANC was tested on two adjacent C67 EVM boards. This set up was necessary in order to process the two channels separately and demonstrate stereo capability. Feedback microphones were fixed on the outside of the closed ear headphone cups to eliminate feedback interference and insure sufficient input amplitudes. Test signals included the aforementioned 1967 Ford Mustang exhaust, aircraft noise at the Cincinnati/Northern Kentucky International Airport, speech and outside music. Although attenuation was achieved for all inputs, in the case of outside music the process was gradual and took almost a minute. This is because outside music is a significantly more complex signal and is not always locally periodic.

Demo:

The 1967 Ford Mustang exhaust was successfully attenuated by the ANC during the demo. Participants listened to music via the Koss UR/15 Personal listening headphones while speakers positioned near the feedback microphones reproduced the Mustang ignition and exhaust noise. In just a few seconds, highly noticeable attenuation was achieved. The LMS did not run until final convergence but improved performance over time was demonstrated. Although the decibel attenuation results were not calculated explicitly during the demo, experience with prior testing suggests attenuation between 6 and 12 dB varying by frequency.

Optimizations and Profiling:

After the FFT/IFFT approach replaced in favor of the FIR filter design, we also set the optimization level to O3 to optimize for speed. These two improvements resulted in a drastic cycle reduction as is evidenced in Table 5 on the next page.

Element	Cycles before Optimizations	Cycles after Optimizations
rcvISR	3868	**958**
xmitISR:	20	**14**
Initialization	43 million	**35.5 million**

Table 5: Cycles by element before/after optimization

Element	Size
Globals	**568**
Local temp	**40 bytes stack**
ONCHIP_PROG	**46.5 Kbytes**
ONCHIP_DATA	**3196 bytes**

Table 6: Final code data size by element

Because our subband approach utilized sample-by-sample processing, with **w(*n*)** updates after each sample, memory paging was not necessary. This approach proved very efficient and allowed significant buffer size reductions. Likewise, heuristic observations made from the output of the assembly file suggest that the loops were unrolled by a factor of two.

Although only 30-40% of the code was parallelized given the utilized build options, most implicit advantages in cycle performance vis-à-vis size of code drawbacks are likely to be acceptable.

4.1 Conclusion and Future Work

Whereas most readily encountered noise signals contain medium range and higher frequency components ($500Hz < f \leq 4000Hz$), commonly available noise canceling headphone products provide up to 10 dB attenuation – 70-80% dampening – only for frequencies below 300 Hz. The cause of the predicament likely lies in the algorithm chosen for at least some of these products – the wideband LMS. Instead of wideband, we show that a new subband approach initially proposed by Siravara et al. in 2002 allows for greatly improved attenuation over a large range of frequencies $0 < f \leq 4kHz$. The following has been successfully demonstrated in meeting the specification of our proposal:

1. A "real-time" active feedback subband LMS ANC system can process 44.1 kHz stereo signals on the TI C67 EVM board.
    a. C code for the algorithm is provided
    b. EVM setup code is likewise made available
2. Even a 2-band system is at least 20-50% more effective at adaptive noise canceling for various frequency bands.

Additionally, subband Matlab code has been created, with repeat testing placing further emphasis on the advantages of the subband system.

Naturally, follow-up work is necessary in order to more clearly identify the additional benefits of 4-band and 6-band systems. Whereas a 4-band system can most likely be implemented on the C67 board, a 6-band may require too much processing power. Scaling the input signal to 22050 Hz mono should accommodate this complication. Most

importantly, actual dB attenuation and time to attenuation for 4- and 6-band systems should be ascertained in order to knowledgeably determine solutions to particular noise canceling applications. Likewise, given further occasion, we would determine the source of LMS irregularities and high interference noticed when feedback microphones were located inside the closed-ear headphones.

Hopefully, the contribution made by making these findings publicly available may stimulate further research possibly culminating in a commercial product, which would be more adept at broad frequency range noise attenuation. Moreover, ameliorated processing requirements – due to a reduction in the LMS filter size that is made possible by the subband algorithm – may in turn also facilitate a reduction in product cost.

4.2 References

[1] Siravara, et al. *A Novel Approach for Single Microphone Active Noise Cancellation*. **2002.** http://www.utdallas.edu/~loizou/speech/mwscas_2002.pdf

Comments: Proposal of subband approach and mathematical definition of algorithm. No code.

[2] Principe et al. *Neural and Adaptive Learning Systems.* **2000.** http://nd.com/NSBook/NEURAL%20AND%20ADAPTIVE%20SYSTEMS15_Estimation_of_the_Gradient_Th.html

Comments: Rule of thumb for step size μ in LMS algorithm. No code.

[3] Texas Instruments. *TMS320C67x DSP Library Programmer Reference Guide.* **2003.** http://focus.ti.com/lit/ug/spru657/spru657.pdf pp. 4-2 and 4-3

Comments: ANSI C and ASM code for wideband LMS. Testing revealed that C code does not correspond to ASM functionality and may have errors. Some parts of the code were used, some were modified as was necessary.

[4] http://www.exhaustsoundclips.com

Comments: provides sound clips of various automobile exhausts. Used 1967 Ford Mustang exhaust as a noise signal for testing.

[5] http://physics.nku.edu/asg/noisesamples.html

Comments: provides noise samples, particularly jet exhaust at Cincinnati/Northern Kentucky International Airport. Used sound as noise signal for testing.

[6] http://www.adaptyv.com/en/

Comments: bulletin board used for debugging.

5.1 Appendix A—Matlab Code

```
% Script file for subband LMS
% Assumes desired.wav and noise.wav exist
% Returns output, the real portion of which is playing using
soundsc(real(output));

temp_d = wavread('desired.wav');
d = temp_d(:, 1);

temp_x = wavread('noise.wav');
x = temp_x(:,1);


% Sets the length to the smaller number of samples
if (length(d) < length(x))
    len = length(d);
else
    len = length(x);
end

i = 1:len;
% initialize coefficients to 0, and buffers to 0
W1 = zeros(16,1);
W2 = zeros(16,1);
buffer_input = [zeros(16,1)];
buffer_desire = [zeros(16,1)];

% Run through for each sample available
for t = 1:len
    % Add the new values to the end of the buffers
    buffer_input = [buffer_input(2:16);(x(t) - d(t))];
    buffer_desire = [buffer_desire(2:16);(-(x(t) - d(t)))];

    % Calculate the fft of the input buffer to see where the noise lies
    fft_buffer_input = fft(buffer_input,16);

    % Split it into two different bands
    input1 = [fft_buffer_input(1:8);zeros(8,1)];
    input2 = [zeros(8,1);fft_buffer_input(9:16)];

    % Take the IFFT to change it back to time domain
    ifft_input1 = ifft(input1, 16);
    ifft_input2 = ifft(input2, 16);

    % Calculate the fft of the desire buffer
    fft_buffer_desire = fft(buffer_desire, 16);

    % Split into two bands
    desire1 = [fft_buffer_desire(1:8);zeros(8,1)];
    desire2 = [zeros(8,1);fft_buffer_desire(9:16)];
```

```
    % Take IFFT to get back actual samples
    ifft_d1 = ifft(desire1, 16);
    ifft_d2 = ifft(desire2, 16);

    % Calculate output and error for first subband
    output1_temp = W1'*ifft_input1;
    e1 = ifft_d1(16) - output1_temp;

    % Update coefficients
    W1 = W1 + .025*ifft_input1*conj(e1);

    % Calculate output and error for 2nd subband
    output2_temp = W2'*ifft_input2;
    e2 = ifft_d2(16) - output2_temp;

    % Update coefficients
    W2 = W2 + .025*ifft_input2*conj(e2);

    % Final output is just sum of subband outputs
    final_output(t) = output1_temp + output2_temp;
end
```

5.1 Appendix B—C Code

```
/*
 * C Code to implement ANC
 * Group 10, Spring 2003
 * Prasanna Malaiyandi, David Mitchell, Samir Sahu
 * Files expected in directory: Noise files, desired files, fftn.h,
fftn.c
 * Usage: saturday  noise#  noise_amplitude  desired#  sampling
 * Sampling does not work, should always be set to 1.
 *
*/

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "fftn.h"

#include <string.h>



#define mu .01

#define pi 3.1415926



int dims[1];     // Used to store the FFT-Size, used by fftn()

double *x;       // Used to store the noise signal

double *x_temp;

double *d;       // Used to store the desired signal

double *d_temp;

double *output_real;  // Used to store the real values of the output

double *output_imag;  // Used to store the imag values of the output



double output1_temp_real;  // Used for real values of the output of
subband 1

double output1_temp_imag;  // Used for imag values of the output of
subband 1
```

```
double output2_temp_real;  // Used for real values of the output of
subband 2

double output2_temp_imag;  // Used for imag values of the output of
subband 2



double e1_real;  // Used to store the real value of the error for
subband 1

double e1_imag;  // Used to store the imag value of the error for
subband 1

double e2_real;  // Used to store the real value of the error for
subband 2

double e2_imag;  // Used to store the imag value of the error for
subband 2



double buffer_input[16];  // Used to store the last 16 values of the
input signal

double buffer_desire[16]; // Used to store the last 16 values of the
desired signal



double W1_real[16];  // Used to store the real values of the filter
coefficients for subband 1

double W1_imag[16];  // Used to store the imag values of the filter
coefficients for subband 1

double W2_real[16];  // Used to store the real values of the filter
coefficients for subband 2

double W2_imag[16];  // Used to store the imag values of the filter
coefficients for subband 2



double fft_buffer_input_real[16];   // Used to store the real values
from the fft of the input buffer

double fft_buffer_input_imag[16];   // Used to store the imag values
from the fft of the input buffer



double fft_buffer_desire_real[16];   // Used to store the real values
from the fft of the desire buffer
```

```
double fft_buffer_desire_imag[16];   // Used to store the imag values
from the fft of the desire buffer


double input1_real[16];  // Used to store the real values of subband 1
for the input

double input1_imag[16];  // Used to store the imag values of subband 1
for the input

double input2_real[16];  // Used to store the real values of subband 2
for the input

double input2_imag[16];  // Used to store the imag values of subband 2
for the input


double desire1_real[16]; // Used to store the real values of subband 1
for the desire

double desire1_imag[16]; // Used to store the imag values of subband 1
for the desire

double desire2_real[16]; // Used to store the real values of subband 2
for the desire


double desire2_imag[16]; // Used to store the imag values of subband 2
for the desire


double ifft_input1_real[16]; // Used to store real values of the ifft
of subband 1 for the input

double ifft_input1_imag[16]; // Used to store imag values of the ifft
of subband 1 for the input

double ifft_input2_real[16]; // Used to store real values of the ifft
of subband 2 for the input

double ifft_input2_imag[16]; // Used to store imag values of the ifft
of subband 2 for the input


double ifft_d1_real[16]; // Used to store real values of the ifft of
subband 1 for the desire

double ifft_d1_imag[16]; // Used to store real values of the ifft of
subband 1 for the desire

double ifft_d2_real[16]; // Used to store real values of the ifft of
subband 1 for the desire
```

```
double ifft_d2_imag[16]; // Used to store real values of the ifft of
subband 1 for the desire




int main(int argc, char ** argv)

{

      int i, j, ret;

      long size1, size2;

      int num, temp;

      double amplitude;

      int sampling;

      FILE *out;

      FILE *noise;

      FILE *desire;

      FILE *dfile;

      FILE *nfile;


      // Open up files in which the outputs will be written to

      out = fopen("output.txt", "wb");

      noise = fopen("noise.txt", "w");

      desire = fopen("desire.txt", "w");


      if (argc != 5)

      {

            printf("Usage: saturday  noise#  noise_amplitude  desired#
sampling\n");

            exit(1);

      }
```

```
	amplitude = atof(argv[2]);
	sampling = atoi(argv[4]);

	if (strcmp(argv[1],"noise1") == 0)
		nfile = fopen("noise1.snd", "rb");
	else if (strcmp(argv[1], "noise2") == 0)
		nfile = fopen("noise2.snd", "rb");
	else if (strcmp(argv[1], "noise3") == 0)
		nfile = fopen("noise3.snd", "rb");
	else
		nfile = fopen("noise1.snd", "rb");

	fseek(nfile, 0, SEEK_END);
	size1 = ftell(nfile);   // Figure out the number of elements in
the noise file
	rewind(nfile);

	  // Read in the noise file into x_temp
	x_temp = (double *) malloc (size1);
	fread(x_temp, 8, size1/8, nfile);
	fclose(nfile);

	if (strcmp(argv[3],"desire1") == 0)
		dfile = fopen("desire1.snd", "rb");
	else if (strcmp(argv[3], "desire2") == 0)
		dfile = fopen("desire2.snd", "rb");
	else if (strcmp(argv[3], "desire3") == 0)
```

```
      dfile = fopen("desire3.snd", "rb");
else
      dfile = fopen("desire1.snd", "rb");

fseek(dfile, 0, SEEK_END);
size2 = ftell(dfile);
rewind(dfile);

d_temp = (double *) malloc (size2);
fread(d_temp, 8, size2/8, dfile);
fclose(dfile);

if (size1 < size2)
      num = (int)size1/(8*sampling);
else
      num = (int)size2/(8*sampling);

  // Initialize the output arrays
output_real = (double *) malloc (num*sizeof(double));
output_imag = (double *) malloc (num*sizeof(double));

  // The FFT Size
dims[0] = 16;
printf("d=%d  x=%d\n", size1/8, size2/8);

  // Create the actual noise input
x = (double *) malloc (num*sizeof(double));
j = 0;
for (i = 0; i < num; i=i+sampling)
{
```

```
      x[j] = amplitude*x_temp[i] + d_temp[i];
      j++;
}
free(x_temp);
printf("Done Creating Noise\n");

  // Create the actual desired input
d = (double *) malloc (num*sizeof(double));
j = 0;
for (i = 0; i < num; i=i+sampling)
{
      d[j] = d_temp[i];
      j++;
}
free(d_temp);
printf("Done Creating Desire\n");


  // Initialize all the coefficients and buffers to 0
for( i = 0 ; i < 16 ; i++)
{
      buffer_input[i] = 0.0;
      buffer_desire[i] = 0.0;
      W1_real[i] = 0.0;
      W1_imag[i] = 0.0;
      W2_real[i] = 0.0;
      W2_imag[i] = 0.0;
}
```

```
        // Run loop till there are no more samples left
      for( i = 0 ; i < num ; i++ )
      {

            // Store the next input and desire value into the end of
the buffers
            for( j = 0 ; j < 15 ; j++)
            {
                  buffer_input[j] = buffer_input[j+1];
                  buffer_desire[j] = buffer_desire[j+1];
            }
            buffer_input[15] = x[i];
            buffer_desire[15] = d[i];



            // Find the fft of the input buffer
            for (j = 0; j < 16; j++)
            {
                  fft_buffer_input_real[j] = buffer_input[j];
                  fft_buffer_input_imag[j] = 0.0;
            }
            ret = fftn(1, dims, fft_buffer_input_real,
fft_buffer_input_imag, 1, 0.0);




            // Split the fft values into 2 subbands, zero padding
subband 1 at the end
```

```
for (j = 0; j < 8; j++)
{
      input1_real[j] = fft_buffer_input_real[j];
      input1_imag[j] = fft_buffer_input_imag[j];
      input1_real[j+8] = 0.0;
      input1_imag[j+8] = 0.0;
}

// Subband 2, zero padded at the beginning
for (j = 0; j < 8; j++)
{
      input2_real[j] = 0.0;
      input2_imag[j] = 0.0;
      input2_real[j+8] = fft_buffer_input_real[j+8];
      input2_imag[j+8] = fft_buffer_input_imag[j+8];
}

// Take the ifft of subband 1 to get real values in time
domain
ret = fftn(1, dims, input1_real, input1_imag, -1, 16.0);

// Take the ifft of subband 2 to get real values in time
domain
ret = fftn(1, dims, input2_real, input2_imag, -1, 16.0);



// Take the fft of the buffer for desire
for (j = 0; j < 16; j++)
{
```

```
                fft_buffer_desire_real[j] = buffer_desire[j];

                fft_buffer_desire_imag[j] = 0.0;

            }

            ret = fftn(1, dims, fft_buffer_desire_real,
fft_buffer_desire_imag, 1, 0.0);




              // Split desire into 2 subbands, just like the input
buffer

            for (j = 0; j < 8; j++)

            {

                desire1_real[j] = fft_buffer_desire_real[j];

                desire1_imag[j] = fft_buffer_desire_imag[j];

                desire1_real[j+8] = 0.0;

                desire1_imag[j+8] = 0.0;

            }


            for (j = 0; j < 8; j++)

            {

                desire2_real[j] = 0.0;

                desire2_imag[j] = 0.0;

                desire2_real[j+8] = fft_buffer_desire_real[j+8];

                desire2_imag[j+8] = fft_buffer_desire_imag[j+8];

            }




            // Take the ifft of each band to get values in the time
domain

            ret = fftn(1, dims, desire1_real, desire1_imag, -1, 16.0);
```

```
            ret = fftn(1, dims, desire2_real, desire2_imag, -1, 16.0);



            // ifft returns values that are conjugates, so you need to
take the inverse of the imaginary values
            for (j = 0; j < 16; j++)
            {
                  desire1_imag[j] = -desire1_imag[j];
                  input1_imag[j] = -input1_imag[j];
                  desire2_imag[j] = -desire2_imag[j];
                  input2_imag[j] = -input2_imag[j];
            }




            // Calculate the output, which is sum of w(n)*i(n)
            output1_temp_real = 0.0;
            output1_temp_imag = 0.0;
            for (j = 0; j < 16; j++)
            {
                  output1_temp_real += (W1_real[j]*input1_real[j] +
W1_imag[j]*input1_imag[j]);
                  output1_temp_imag += (W1_imag[j]*input1_real[j] -
W1_real[j]*input1_imag[j]);
}



              // Error is equal to the last desired minus the output
calculated above
            e1_real = desire1_real[15] - output1_temp_real;
            e1_imag = -desire1_imag[15] - output1_temp_imag;
```

```
            // Update the coefficients. w(n) = w(n) +
mu*ifft_input1*conj(e1)
            for (j = 0; j < 16; j++)

            {

                  W1_real[j] += mu * (input1_real[j]*e1_real -
input1_imag[j]*e1_imag);

                  W1_imag[j] += mu * (input1_imag[j]*e1_real +
input1_real[j]*e1_imag);

            }




              // Repeat calculations for output, error, and
coefficients for subband 2

            output2_temp_real = 0.0;

            output2_temp_imag = 0.0;

            for (j = 0; j < 16; j++)

            {

                  output2_temp_real += (W2_real[j]*input2_real[j] +
W2_imag[j]*input2_imag[j]);

                  output2_temp_imag += (W2_imag[j]*input2_real[j] -
W2_real[j]*input2_imag[j]);

            }



            e2_real = desire2_real[15] - output2_temp_real;

            e2_imag = -desire2_imag[15] - output2_temp_imag;



            for (j = 0; j < 16; j++)

            {

                  W2_real[j] += mu * (input2_real[j]*e2_real -
input2_imag[j]*e2_imag);

                  W2_imag[j] += mu * (input2_imag[j]*e2_real +
input2_real[j]*e2_imag);

            }
```

```
            // Final output is the sum of the real output for the two
subbands
            output_real[i] = output1_temp_real + output2_temp_real;
            output_imag[i] = -output1_temp_imag + -output2_temp_imag;


      }

      // Done calculating all the outputs
      printf("Done With output");

      // Store the outputs, the desired sound, and the input sound into
files
      for (i = 0; i < num; i++)
      {
            fprintf(out, "%f\n", output_real[i]);
            fprintf(desire, "%f\n", d[i]);
            fprintf(noise, "%f\n", x[i]);


      }
      fclose(out);
      fclose(desire);
      fclose(noise);
      free(output_real);
      free(output_imag);
      free(x);
      free(d);
      return 0;
}
```

5.3 Appendix C—EVM Code

```
/*
 * EVM to implement subband ANC
 * Group 10, Spring 2003
 * Prasanna Malaiyandi, David Mitchell, Samir Sahu
 * Code is used to handle one channel of audio. Therefore, 2 EVMs
needed to actually
 * implement stereo solution. Only output changes for the difference
between
 * left channel and right channel. Assuming that audio is already split
at source
 * before entering EVM.
*/



#include <stdlib.h>



#include <mcbsp.h>              /* mcbsp devlib */

#include <common.h>

#include <mcbspdrv.h>           /* mcbsp driver */

#include <board.h>              /* EVM library */

#include <codec.h>              /* codec library */

#include <mathf.h>              /* math library */

#include <intr.h>               /* interrupt library */

#include <linkage.h>



#define fs 44100  // Sampling rate

#define mu .025         // Value for the LMS algorithm

#define pi 3.1415926



int output;         // Output value

float buffer_input1[16];  // Buffer for input subband 1

float buffer_input2[16];  // Buffer for input subband 2

float desire[33];  // Buffer for desire signal
```

```
float input[33];   // Buffer for input signal

float W1[16];  // COefficients for subband 1

float W2[16];  // coefficients for subband 2


// Filter coefficients for 0-1kHz
float filter1[33] = {0.0135, 0.0165, 0.0196, 0.0227, 0.0258, 0.0288,
0.0317, 0.0344,

                                0.0370, 0.0393, 0.0414, 0.0433, 0.0448,
0.0460, 0.0469, 0.0474,

                           0.0476, 0.0474, 0.0469, 0.0460, 0.0448,
0.0433, 0.0414, 0.0393,

                           0.0370, 0.0344, 0.0317, 0.0288, 0.0258,
0.0227, 0.0196, 0.0165,

                           0.0135};

// Filter coefficients for 1-4 kHz
float filter2[33] = {-0.0123, -0.0040,  0.0008, -0.0010, -0.0107, -
0.0271, -0.0468, -0.0646,

                                -0.0748, -0.0725, -0.0553, -0.0239,
0.0177,  0.0625,  0.1026,  0.1302,

                                 0.1401,  0.1302,  0.1026,  0.0625,
0.0177, -0.0239, -0.0553, -0.0725,

                                -0.0748, -0.0646, -0.0468, -0.0271, -
0.0107, -0.0010,  0.0008, -0.0040,

                                -0.0123};

// Filter coeffcients for > 4 kHz

float filter3[33] = {-0.0081, -0.0176, -0.0226, -0.0209, -0.0119,
0.0028,  0.0194,  0.0332,

                                0.0388,  0.0325,  0.0123, -0.0206, -
0.0619, -0.1054, -0.1436, -0.1698,

                                0.8209, -0.1698, -0.1436, -0.1054, -
0.0619, -0.0206,  0.0123,  0.0325,


                                0.0388,  0.0332,  0.0194,  0.0028, -
0.0119, -0.0209, -0.0226, -0.0176,

                               -0.0081};
```

```
/************************ FUNCTIONS ****************************/


/*************************************************************
 *    Name: rcvISR
 *  Inputs: none
 *  Output: none
 * Purpose: Interrupt vector to be called whenever a single
 *  sample of data is ready to be read.  For each sample, we
 *  simply store it in a buffer and increment the index into the
 *  buffer.
 *************************************************************/
interrupt void rcvISR(void) {
      int j;
      float output_temp1, output_temp2, desire_b1, desire_b2, input_b1,
input_b2, input_b3, error1, error2;


      output_temp1 = MCBSP0_DRR;


        // Shift the buffer to the left to make room for the new input
and desire value
      for (j = 0; j < 32; j++)
      {
            desire[j] = desire[j+1];
            input[j] = input[j+1];
      }


      // Mask out the input and desired from the serial port register.
Input is the top 16 bits, desired is the bottom 16 bits. Normalize to -
1 -> 1
      input[32] = (float)((signed short int)(((MCBSP0_DRR) &
0xffff0000) >> 16))/32768;
```

```
      desire[32] = (float)((signed short int)((MCBSP0_DRR) &
0x0000ffff))/32768;

      input[32] += desire[32];  // Add the noise to the input




      // Initialize the input arrays to 0

      input_b1 = 0.0;

      input_b2 = 0.0;

      input_b3 = 0.0;


      // Calculate the inputs to the subbands, doing FIR filtering

      for (j = 0; j < 33; j++)

      {

            input_b1 += filter1[j]*input[j];

            input_b2 += filter2[j]*input[j];

            input_b3 += filter3[j]*input[j];

      }


      // Create room in the buffer arrays for the new inputs, then add
them to the end
      for( j = 0 ; j < 15 ; j++)

      {

            buffer_input1[j] = buffer_input1[j+1];

            buffer_input2[j] = buffer_input2[j+1];

      }

      buffer_input1[15] = input_b1;

      buffer_input2[15] = input_b2;
```

```
	// Calculate the desired for the subbands
	desire_b1 = 0.0;

	desire_b2 = 0.0;

	for (j = 0; j < 33; j++)

	{

		desire_b1 += filter1[j]*desire[j];

		desire_b2 += filter2[j]*desire[j];

	}



	// Calculate the outputs based on the coefficients times the
input buffer

	output_temp1 = 0.0;

	for (j = 0; j < 16; j++)

	{

		output_temp1 += (W1[j]*buffer_input1[j]);

	}


	// Calculate the error based on desired minus output

	error1 = desire_b1 - output_temp1;



	// Update the coefficients
	for (j = 0; j < 16; j++)

	{

		W1[j] += mu * (buffer_input1[j]*error1);

	}



	// Do the same things as above for the 2nd subband

	output_temp2 = 0.0;

	for (j = 0; j < 16; j++)

	{
```

```
            output_temp2 += (W2[j]*buffer_input2[j]);
      }

      error2 = desire_b2 - output_temp2;

      for (j = 0; j < 16; j++)
      {
            W2[j] += mu * (buffer_input2[j]*error2);
      }

      // Output is just the sum of the outputs, plus input_b3 which is
the high frequency.  Uncomment appropriate line
      // output = (((short
int)((output_temp1+output_temp2+input_b3)*32768)<<16)&0xffff0000);
      // Left Channel
      // output = (((short
int)((output_temp1+output_temp2+input_b3)*32768))&0x0000ffff);
// Right Channel
}

/*****************************************************************
 *    Name: xmitISR
 *  Inputs: none
 *  Output: none
 * Purpose: Interrupt vector to be called whenever the serial
 *  port is ready for a sample to be written.
 *****************************************************************/
interrupt void xmitISR(void) {

  // Write the output to the serial port register
  MCBSP0_DXR=output;
```

```
}

/*****************************************************************
 *    Name: main
 *  Inputs: none
 *  Output: none
 * Purpose:
 *****************************************************************/
int main(void) {
  Mcbsp_dev dev;                /* Serial port device */
  int i;


  evm_init();                   /* Standard board initialization */
  mcbsp_drv_init();             /* Call this before using McBSP
functions */


  /* Open serial port */
  if (!(dev=mcbsp_open(0))) {
    return(ERROR);
  }
  /* Configure McBSP */
  mcbsp_setup(dev);   /* See bottom of this file */


  /******************* configure CODEC *********************/
  /* EXIT_ERROR is a macro which jumps to exit_err if the function
     returns an ERROR */
  EXIT_ERROR(codec_init());
  codec_change_sample_rate(fs, TRUE);
```

```
  EXIT_ERROR(codec_adc_control(LEFT,20.0,FALSE,MIC_SEL));     // Put
noise on the left channel

  EXIT_ERROR(codec_adc_control(RIGHT,20.0,FALSE,LINE_SEL));   //
Desired on the right channel

  /* mute (L/R)LINE input to mixer */

  EXIT_ERROR(codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,TRUE));

  EXIT_ERROR(codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,TRUE));

  /* D/A 0.0 dB atten, do not mute DAC outputs */

  EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));

  EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));




  /**************** initialize coefficients and buffer *********/

      for( i = 0 ; i < 16 ; i++)

      {

            buffer_input1[i] = 0.0;

            buffer_input2[i] = 0.0;

            W1[i] = 1.0;

            W2[i] = 1.0;

      }

      output = 0.0;


      for ( i = 0; i < 33; i++)

      {

            input[i] = 0.0;

            desire[i] = 0.0;

      }
```

```
  /**************** setup interrupt routines ******************/


  intr_init();

  /* Hook up serial transmit interrupt to CPU Interrupt 14 */

  /* Repeat the same process for the receive interrupt */

  intr_map(CPU_INT15,ISN_RINT0);

  INTR_CLR_FLAG(CPU_INT15);

  intr_hook(rcvISR,CPU_INT15);

  intr_map(CPU_INT14,ISN_XINT0);

  INTR_CLR_FLAG(CPU_INT14);     /* Clear any old interrupts */

  intr_hook(xmitISR,CPU_INT14); /* Hook our own xmitISR into chain for
14 */

  /* Enable all necessary interrupts */

  INTR_ENABLE(CPU_INT_NMI);     /* Non-maskable interrupt */

  INTR_ENABLE(CPU_INT15);

  INTR_ENABLE(CPU_INT14);

  INTR_GLOBAL_ENABLE();         /* Controls whether ANY interrupts
function */


  /****************** Turn on the serial port **********************/

  MCBSP_ENABLE(dev->port,MCBSP_RX|MCBSP_TX);


  /* At this point, the program leaves main and enters an infinite

   *  idle loop.  Interrupts continue to function */

  while(1);
  exit_err:

  return(ERROR);
}


/*************************************************************
```

```
 *    Name: mcbspSetup
 *  Inputs: Mcbsp_dev
 *  Output: none
 * Purpose: McBSP stands for Multi-Channel Buffered Serial Port.
 *  It is build onto the C67 processor itself, and is how the
 *  codec communicates with the processor.  This function sets
 *  up the serial port for communication with the codec, and
 *  should never need to be modified.
 ************************************************************/
int mcbsp_setup(Mcbsp_dev dev) {
  /* Structure with all configuration parameters for serial port */
  Mcbsp_config mcbspConfig;
  memset(&mcbspConfig,0,sizeof(mcbspConfig)); /* Initialize everything
to 0 */

  mcbspConfig.loopback              = FALSE;
  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;

  mcbspConfig.rx.update             = TRUE;
  mcbspConfig.rx.clock_polarity     = CLKR_POL_FALLING;
```

```
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.rx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1      = 0;
  mcbspConfig.rx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.rx.data_delay         = DATA_DELAY0;

  /* Pass entire structure to mcbsp_config, a library function which
   *  sets registers according to the contents of the structure */
  if(mcbsp_config(dev,&mcbspConfig) != OK) {
    return(ERROR);
  }
  return(OK);
}
```