Real Time Digital Signal Processing

Lamber Yang (00933105) *Yumeng Sun (00921666)*

Project: Speech Enhancement

Aims:

- To build a real-time system to dynamically reduce background noise from speech signals.

Premise and algorithm

System parameters:

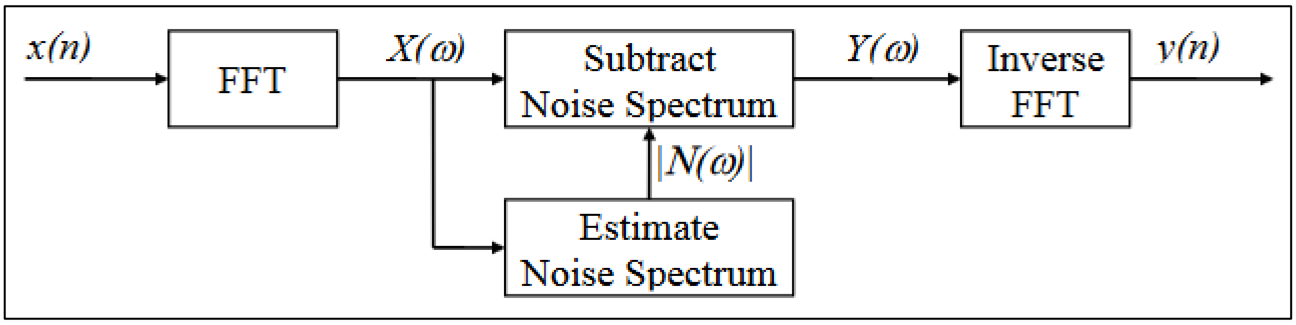


Figure 1. Spectral Subtraction System

Sample Rate: $8kHz$.

256-point Fourier Transform per 64 samples ($8ms$).

The technique used in the project is **spectral subtraction**, which subtracts two spectrums from each other to provide an output. It works on the assumption that the input signal can be split into the speech and noise spectrums in the frequency domain. Following this assumption, the noise spectrum is estimated over several frames and subtracted from the input spectrum to retrieve the speech spectrum.

Overlap-Add Processing

In order to achieve this frequency domain processing, the input must be split into frames. These frames are overlapping segments of the continuous signal which can be reassembled after processing.

Before performing the FFT and after performing the inverse-FFT, the frames must be multiplied by windows to avoid spectral artefacts. The windows chosen were that of the square root of the Hamming window:

$$\sqrt{1 - 0.85185cos\frac{(2k+1)\pi}{N}}\ for\ k = 0, \dots, N-1$$

This window was chosen because the window function is applied twice, once before the FFT and once after the inverse-FFT, so the resultant effect is that of a Hamming window.

By having an **oversampling ratio** of 4, since there are 256 samples per frame, each frame starts 64 samples after the preceding one. This reduces the distortion effect of gain difference between successive frames. The output sample is constructed from all four frames.

The frame and window processes are included in the example template, **enhance.c**.

Basic Program

```
// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#define WINCONST 0.85185            /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0                /* sample frequency, ensure this matches Config for AIC */
#define FFTLEN 256                  /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)          /* number of frequency bins from a real FFT */
#define OVERSAMP 4                  /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP)  /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC)   /* length of I/O buffers */

#define OUTGAIN 16000.0             /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)       /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP       /* time between calculation of each frame */

#define NOISE_FACTOR 3              /* oversubtraction factor alpha */
#define LAMBDA 0.01                 /* gainFactor minimum */
```

Figure 2. Definitions of constants

These are constants defined to implement the skeleton program.

```
/*  Initialize and zero fill arrays */

    inbuffer        = (float *) calloc(CIRCBUF, sizeof(float)); /* Input array */
    outbuffer       = (float *) calloc(CIRCBUF, sizeof(float)); /* Output array */
    inframe         = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    outframe        = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    cintermediate   = (complex *) calloc(FFTLEN, sizeof(complex)); /*Complex array */
    inwin           = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
    outwin          = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */
    m1              = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 1 */
    m2              = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 2 */
    m3              = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 3 */
    m4              = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 4 */
    noise           = (float *) calloc(FFTLEN, sizeof(float));  /* noise estimate */
```

Figure 3. Initialisation of arrays

Compared to previous experiments, the project requires the use of 4 arrays to store minimum buffers, with m1 being updated with the most recent sample. A noise estimate array is also initialised, as well as arrays required for framing and windowing.

```
/* initialize algorithm constants */

    for (k=0;k<FFTLEN;k++)
    {
    inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
    outwin[k] = inwin[k];
    }
    ingain=INGAIN;
    outgain=OUTGAIN;
```

Figure 4. Input and output parameters

This code initialises the input and output windows, as well as input and output gain for the ADC and DAC respectively, as previously defined.

```
int i, k, m;
int io_ptr0;
float mag, minM, gainFactor, alpha;

/* work out fraction of available CPU time used by algorithm */
cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;

/* wait until io_ptr is at the start of the current frame */
while((io_ptr/FRAMEINC) != frame_ptr);

/* then increment the framecount (wrapping if required) */
if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;

/* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where the
data should be read (inbuffer) and saved (outbuffer) for the purpose of processing */
io_ptr0=frame_ptr * FRAMEINC;

/* copy input data from inbuffer into inframe (starting from the pointer position) */

m=io_ptr0;
for (k=0;k<FFTLEN;k++)
{
    inframe[k] = inbuffer[m] * inwin[k];
    if (++m >= CIRCBUF) m=0; /* wrap if required */
}
```

Figure 5. Setup of input frame

Receiving the input samples from the audio port in the interrupt function, this code is provided by the template to calculate and wrap the frames with the window.

```
for (i=0; i<FFTLEN; i++){
    cintermediate[i]= cmplx(inframe[i], 0);
}
fft(FFTLEN, cintermediate);
```

Figure 6. Fast Fourier Transform of input frame

First, the FFT must be performed on the windowed frames.

```
//for every element in the spectrum
for (i=0; i<FFTLEN; i++){

    //evaluate magnitude of element
    mag = cabs(cintermediate[i]);

    //evaluate appropriate element of corresponding minimum buffer
    if (++count >= 313) { //313 for 2.5 seconds, 251 for 2.0 seconds
        count=0;
        m4[i] = m3[i];
        m3[i] = m2[i];
        m2[i] = m1[i];
        m1[i] = mag;
    }
    else {
        if(mag<m1[i]) m1[i] = mag;
    }
```

Figure 7. Filling minimum buffers

The complex absolute value of each sample of the frame is calculated. Each of these magnitudes, **mag**, is inserted into **m1[i]** if **mag** is smaller than the existing value. After 313 counts of frames have passed, which is calculated from $\frac{Frame\ Delay}{TFRAME} = \frac{2.5}{\frac{256}{4}/8000} = \frac{2.5}{0.008} = 312.5$, rounded up to 313, the previous values are shifted along the buffers, causing a temporal delay of 2.5 seconds between each minimum buffer.

```
//evaluate element of the noise magnitude spectrum
minM = m1[i];
if (m2[i]<minM) minM = m2[i];
if (m3[i]<minM) minM = m3[i];
if (m4[i]<minM) minM = m4[i];

noise[i] = NOISE_FACTOR*minM;

//Perform noise subtraction for each element
gainFactor = 1 - ((noise[i])/mag);
if (gainFactor < LAMBDA) gainFactor = LAMBDA; //this is to prevent gainFactor from going negative

cintermediate[i]= rmul(gainFactor, cintermediate[i]);
```

Figure 8. Calculation of gain factor.

Then out of these 4 minimum buffers, the lowest value for each sample is selected, and compiled into a single **minM** variable. The noise spectrum is then multiplied by the oversubtraction factor, alpha, here called **NOISE_FACTOR**, and the gain factor is calculated according to $1 - \frac{|N(\omega)|}{|X(\omega)|}$.

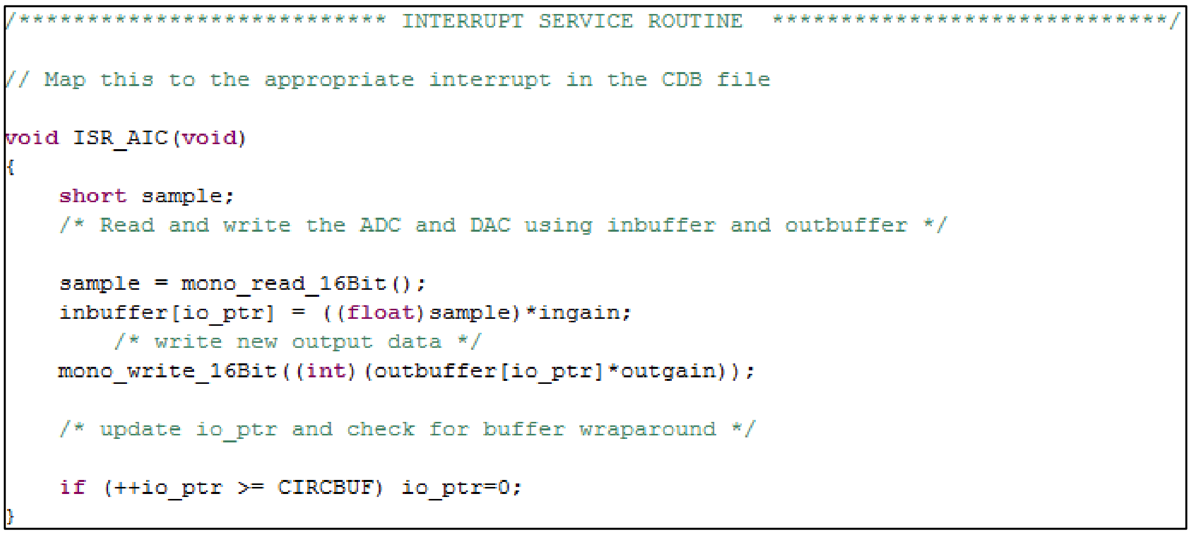
The factor $\max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$ is introduced, with λ being a value between 0.01 and 0.1. λ is set to 0.01 as this is closer to 0. This is used in order to keep the value of the gain factor, $g(\omega)$, from going negative. Once $1 - \frac{|N(\omega)|}{|X(\omega)|}$ approaches 0, the gain will default to λ until it increases beyond it.

The input frame is multiplied by the gain factor to produce the output frame.

```
ifft(FFTLEN, cintermediate);

for (i=0; i<FFTLEN; i++){
    outframe[i]= real(cintermediate[i]);
}

/*******************************************************************************/

/* multiply outframe by output window and overlap-add into output buffer */

m=io_ptr0;

for (k=0;k<(FFTLEN-FRAMEINC);k++)
{                                          /* this loop adds into outbuffer */
    outbuffer[m] = outbuffer[m]+outframe[k]*outwin[k];
    if (++m >= CIRCBUF) m=0; /* wrap if required */
}
for (;k<FFTLEN;k++)
{
    outbuffer[m] = outframe[k]*outwin[k];    /* this loop over-writes outbuffer */
    m++;
}
```

Figure 9. Output frame and buffer

Finally the inverse-FFT is performed, the frame is windowed again, and the sample is ready to be outputted to the audio port once it returns to the interrupt function.



```
/**************************** INTERRUPT SERVICE ROUTINE  *****************************/

// Map this to the appropriate interrupt in the CDB file

void ISR_AIC(void)
{
    short sample;
    /* Read and write the ADC and DAC using inbuffer and outbuffer */

    sample = mono_read_16Bit();
    inbuffer[io_ptr] = ((float)sample)*ingain;
        /* write new output data */
    mono_write_16Bit((int)(outbuffer[io_ptr]*outgain));

    /* update io_ptr and check for buffer wraparound */

    if (++io_ptr >= CIRCBUF) io_ptr=0;
}
```

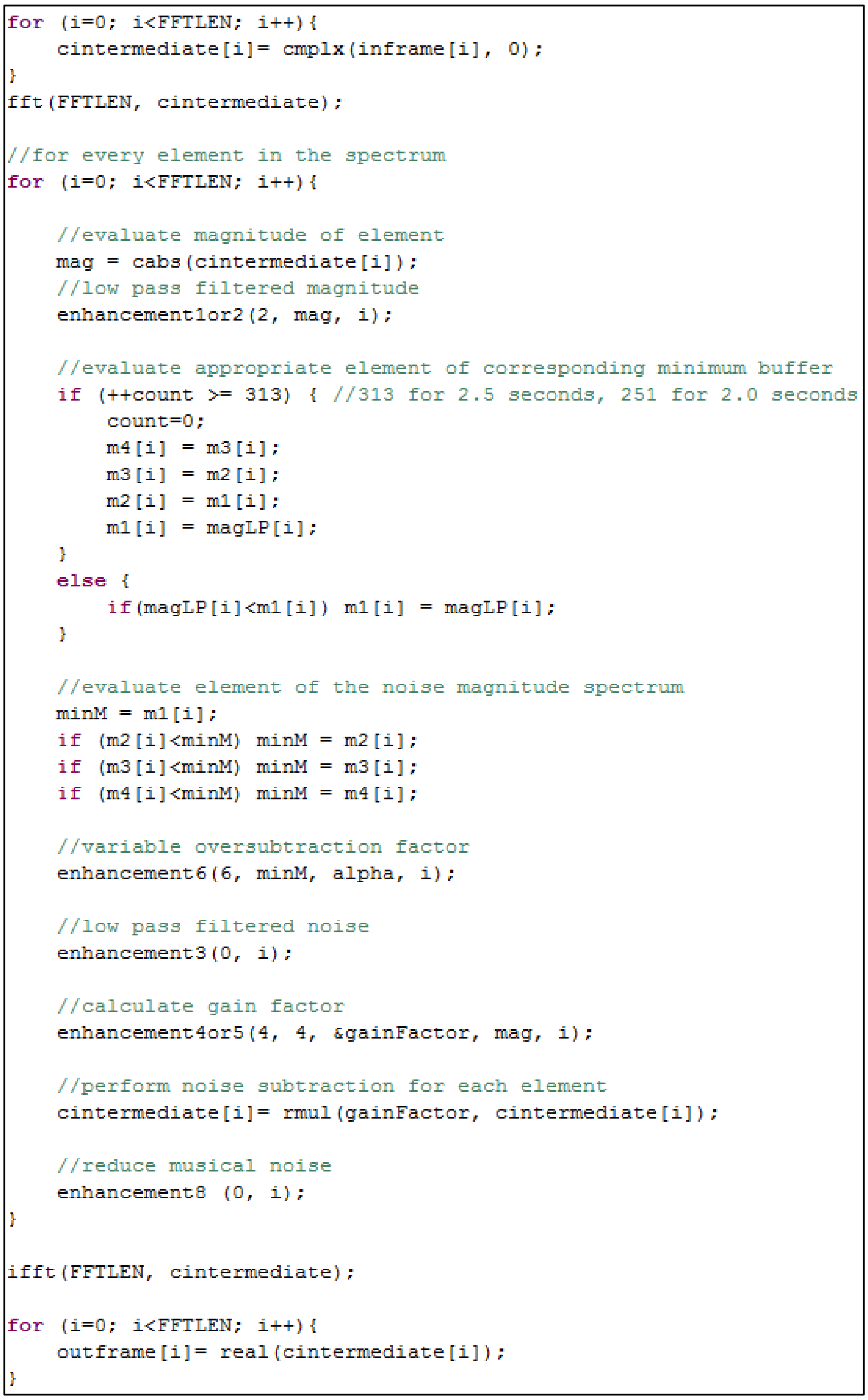
Figure 10. Interrupt function

The interrupt function reads and writes the input and output respectively. One difference between this interrupt and previous iterations is the use of a circular buffer for assignment of values.

Conclusion

The basic code produces a tangible effect on the input sample, amplifying and distorting the voice, but little effect on filtering out the background noise at first. It takes approximately 10 seconds for the filtering effect to begin, as expected from the 4 frames of comparison and 2.5 seconds required to sample each, and even then is not optimal. Noise is still present and the voice is slightly distorted. Further enhancements to the code are required.

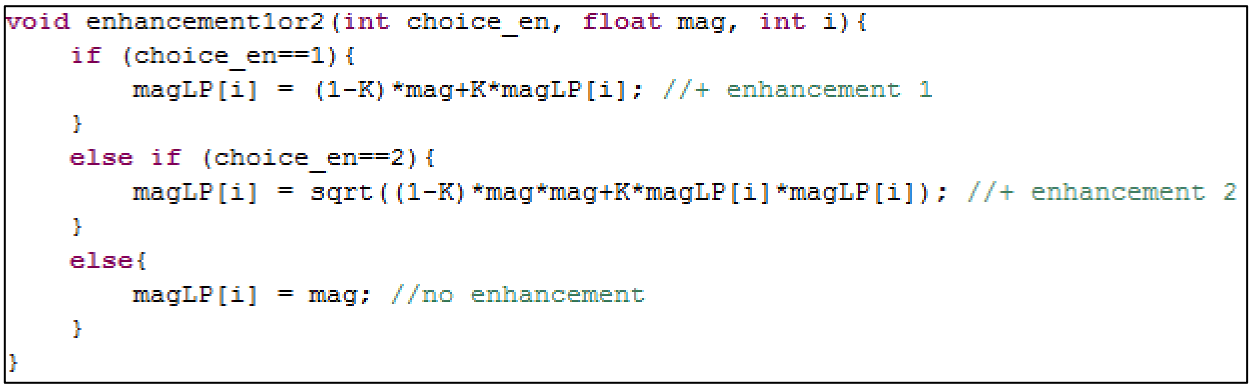
Enhancements



```
for (i=0; i<FFTLEN; i++){
    cintermediate[i]= cmplx(inframe[i], 0);
}
fft(FFTLEN, cintermediate);

//for every element in the spectrum
for (i=0; i<FFTLEN; i++){

    //evaluate magnitude of element
    mag = cabs(cintermediate[i]);
    //low pass filtered magnitude
    enhancement1or2(2, mag, i);

    //evaluate appropriate element of corresponding minimum buffer
    if (++count >= 313) { //313 for 2.5 seconds, 251 for 2.0 seconds
        count=0;
        m4[i] = m3[i];
        m3[i] = m2[i];
        m2[i] = m1[i];
        m1[i] = magLP[i];
    }
    else {
        if(magLP[i]<m1[i]) m1[i] = magLP[i];
    }

    //evaluate element of the noise magnitude spectrum
    minM = m1[i];
    if (m2[i]<minM) minM = m2[i];
    if (m3[i]<minM) minM = m3[i];
    if (m4[i]<minM) minM = m4[i];

    //variable oversubtraction factor
    enhancement6(6, minM, alpha, i);

    //low pass filtered noise
    enhancement3(0, i);

    //calculate gain factor
    enhancement4or5(4, 4, &gainFactor, mag, i);

    //perform noise subtraction for each element
    cintermediate[i]= rmul(gainFactor, cintermediate[i]);

    //reduce musical noise
    enhancement8 (0, i);
}

ifft(FFTLEN, cintermediate);

for (i=0; i<FFTLEN; i++){
    outframe[i]= real(cintermediate[i]);
}
```

Figure 11. Processing of cintermediate

Compared to the previous execution of the code, the enhancements are now included as functions to be called within the **process_frame** function. Enhancements 1 and 2 do low pass filtering of the magnitude, which is used to calculate **noise[i]**. If enhancement 6 is used, a variable oversubtraction factor is used to calculate **noise[i]**. Enhancement 3 low pass filters the noise. Enhancements 4 and 5 calculate the gain based on different equations. Enhancement 8 aims to reduce musical noise. Enhancements 7 and 9 are not included in the code, but explained below.



```
void enhancement1or2(int choice_en, float mag, int i){
    if (choice_en==1){
        magLP[i] = (1-K)*mag+K*magLP[i]; //+ enhancement 1
    }
    else if (choice_en==2){
        magLP[i] = sqrt((1-K)*mag*mag+K*magLP[i]*magLP[i]); //+ enhancement 2
    }
    else{
        magLP[i] = mag; //no enhancement
    }
}
```

Figure 12. Function enhancement1or2()

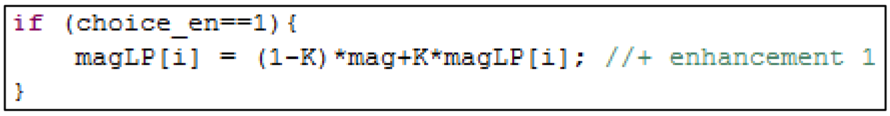
<u>Enhancement 1</u>

Using a low pass filtered version of the magnitude, mag, to estimate noise.

$P_t(\omega) = (1-k) \times mag + k \times P_{t-1}(\omega)$, where $k = e^{\frac{-T}{\tau}}$.

T = Frame rate, given by TFRAME in the template.

τ = Time constant, between 20ms and 80ms. It is initialised to 50ms for parity.



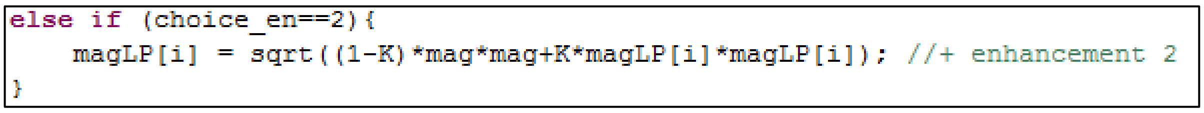
```
if (choice_en==1){
    magLP[i] = (1-K)*mag+K*magLP[i]; //+ enhancement 1
}
```

Figure 13. Enhancement 1

This enhancement reduced a lot of background noise. As a result of this enhancement, the oversubtraction factor could be reduced to a provisional value of 3 whilst performing similarly.

<u>Enhancement 2</u>

Similar to the first enhancement except low pass filtering **mag** and **magLP[i]** squared, and square rooting the sum, to obtain $P(\omega)$. $P_t(\omega) = \sqrt{(1-k) \times mag^2 + k \times P_{t-1}(\omega)^2}$,



```
else if (choice_en==2){
    magLP[i] = sqrt((1-K)*mag*mag+K*magLP[i]*magLP[i]); //+ enhancement 2
}
```

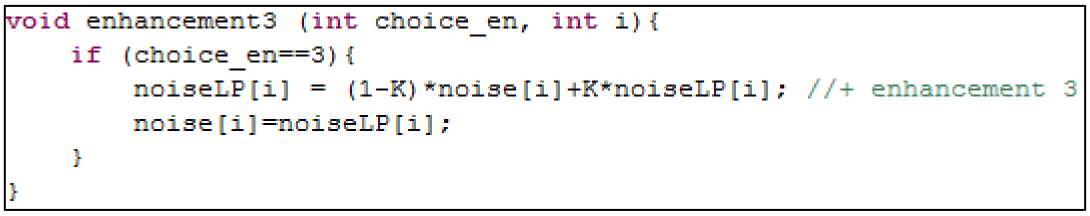
Figure 14. Enhancement 2

This enhancement produced a slight improvement on enhancement 1 and so rendered enhancement 1 obsolete, as they performed the same function.

<u>Enhancement 3</u>

Low pass filtering the noise estimate to avoid discontinuities when rotating minimum buffers.

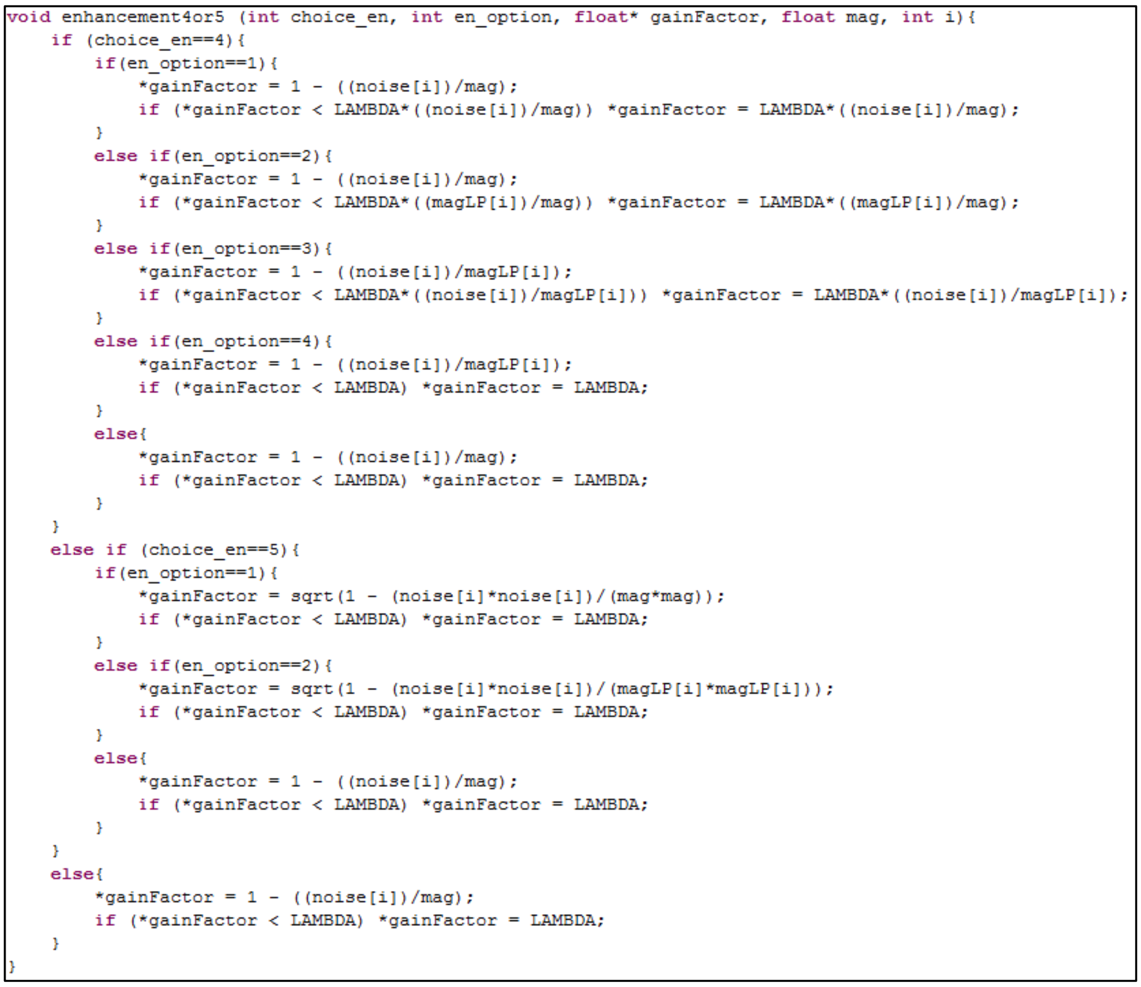
$$N'_t(\omega) = (1-k) \times N(\omega) + k \times N'_{t-1}(\omega)$$



```
void enhancement3 (int choice_en, int i){
    if (choice_en==3){
        noiseLP[i] = (1-K)*noise[i]+K*noiseLP[i]; //+ enhancement 3
        noise[i]=noiseLP[i];
    }
}
```

Figure 15. Function enhancement3()

This enhancement did not produce a noticeable effect for any of the provided sound files, so it was not chosen. It is ineffective for low-variance noise profiles, but may help in high-variance situations.

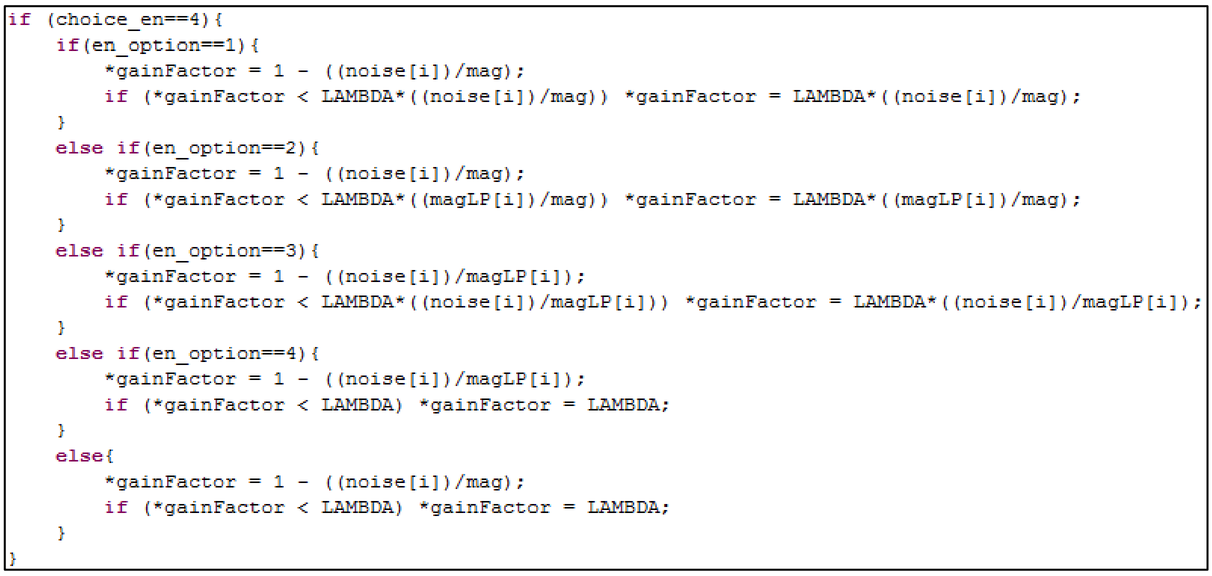


```
void enhancement4or5 (int choice_en, int en_option, float* gainFactor, float mag, int i){
    if (choice_en==4){
        if(en_option==1){
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA*((noise[i])/mag)) *gainFactor = LAMBDA*((noise[i])/mag);
        }
        else if(en_option==2){
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA*((magLP[i])/mag)) *gainFactor = LAMBDA*((magLP[i])/mag);
        }
        else if(en_option==3){
            *gainFactor = 1 - ((noise[i])/magLP[i]);
            if (*gainFactor < LAMBDA*((noise[i])/magLP[i])) *gainFactor = LAMBDA*((noise[i])/magLP[i]);
        }
        else if(en_option==4){
            *gainFactor = 1 - ((noise[i])/magLP[i]);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else{
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
    }
    else if (choice_en==5){
        if(en_option==1){
            *gainFactor = sqrt(1 - (noise[i]*noise[i])/(mag*mag));
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else if(en_option==2){
            *gainFactor = sqrt(1 - (noise[i]*noise[i])/(magLP[i]*magLP[i]));
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else{
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
    }
    else{
        *gainFactor = 1 - ((noise[i])/mag);
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
}
```

Figure 16. Function enhancement4or5()

Enhancement 4

Setting $g(\omega)$ to a value other than $\max(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|})$ to improve corrective gain. Of all the values tested, $\max(\lambda, 1 - \frac{|N(\omega)|}{|P(\omega)|})$ performed the best. The rest of the options are stated in the project handout.



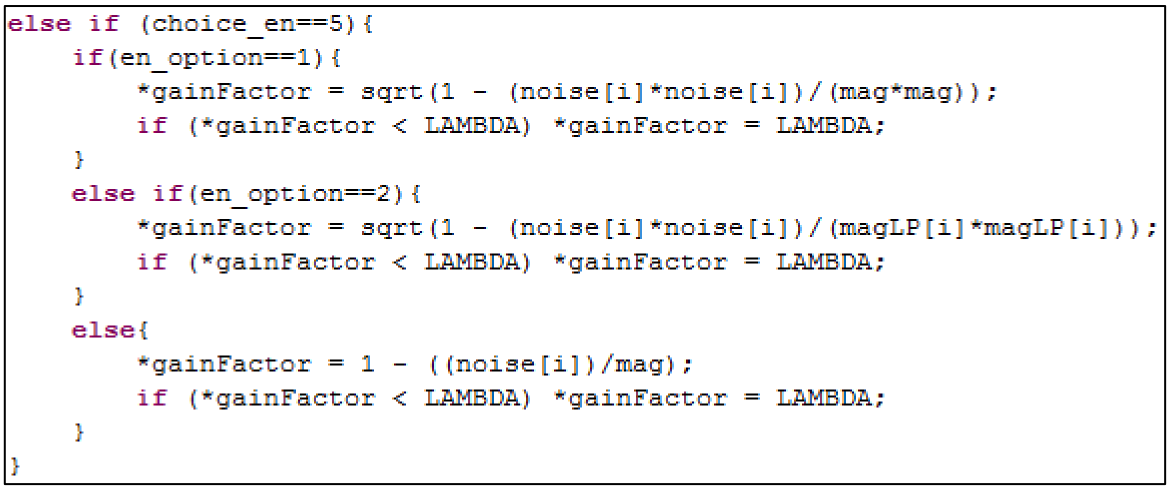
```
if (choice_en==4){
    if(en_option==1){
        *gainFactor = 1 - ((noise[i])/mag);
        if (*gainFactor < LAMBDA*((noise[i])/mag)) *gainFactor = LAMBDA*((noise[i])/mag);
    }
    else if(en_option==2){
        *gainFactor = 1 - ((noise[i])/mag);
        if (*gainFactor < LAMBDA*((magLP[i])/mag)) *gainFactor = LAMBDA*((magLP[i])/mag);
    }
    else if(en_option==3){
        *gainFactor = 1 - ((noise[i])/magLP[i]);
        if (*gainFactor < LAMBDA*((noise[i])/magLP[i])) *gainFactor = LAMBDA*((noise[i])/magLP[i]);
    }
    else if(en_option==4){
        *gainFactor = 1 - ((noise[i])/magLP[i]);
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
    else{
        *gainFactor = 1 - ((noise[i])/mag);
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
}
```

Figure 17. Enhancement 4

This enhancement helps split the voice from the noise and make the musical noise less noticeable, and as such was included.

Enhancement 5

Calculating the gain in the power domain, i.e. $g(\omega) = \max(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}})$.



```
else if (choice_en==5){
    if(en_option==1){
        *gainFactor = sqrt(1 - (noise[i]*noise[i])/(mag*mag));
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
    else if(en_option==2){
        *gainFactor = sqrt(1 - (noise[i]*noise[i])/(magLP[i]*magLP[i]));
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
    else{
        *gainFactor = 1 - ((noise[i])/mag);
        if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
    }
}
```

Figure 18. Enhancement 5

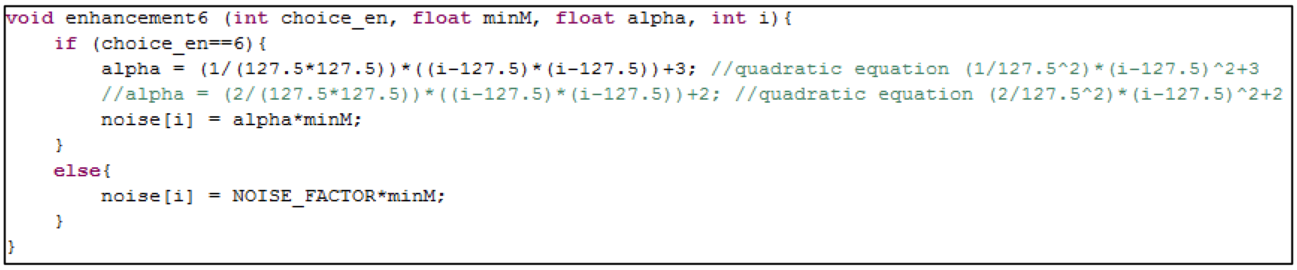
This enhancement seemed to be less effective at reducing the effects of musical noise than enhancement 4, so the latter was used instead of this.

Enhancement 6

Overestimating the oversubtraction factor for low frequency bins.

This can be achieved by applying a quadratic form to the oversubtraction factor variable **alpha**, which now substitutes **NOISE_FACTOR** to produce the noise estimation.

Using the quadratic equation $alpha = \frac{(\mathrm{i}-127.5)^2}{127.5^2} + 3$, a curve can be achieved which varies from 3 to 4.



```
void enhancement6 (int choice_en, float minM, float alpha, int i){
    if (choice_en==6){
        alpha = (1/(127.5*127.5))*((i-127.5)*(i-127.5))+3; //quadratic equation (1/127.5^2)*(i-127.5)^2+3
        //alpha = (2/(127.5*127.5))*((i-127.5)*(i-127.5))+2; //quadratic equation (2/127.5^2)*(i-127.5)^2+2
        noise[i] = alpha*minM;
    }
    else{
        noise[i] = NOISE_FACTOR*minM;
    }
}
```

Figure 19. Function enhancement6()

This enhancement produced a slightly clearer output than without the enhancement, so it was included.

Enhancement 7

Varying the duration of the frames.

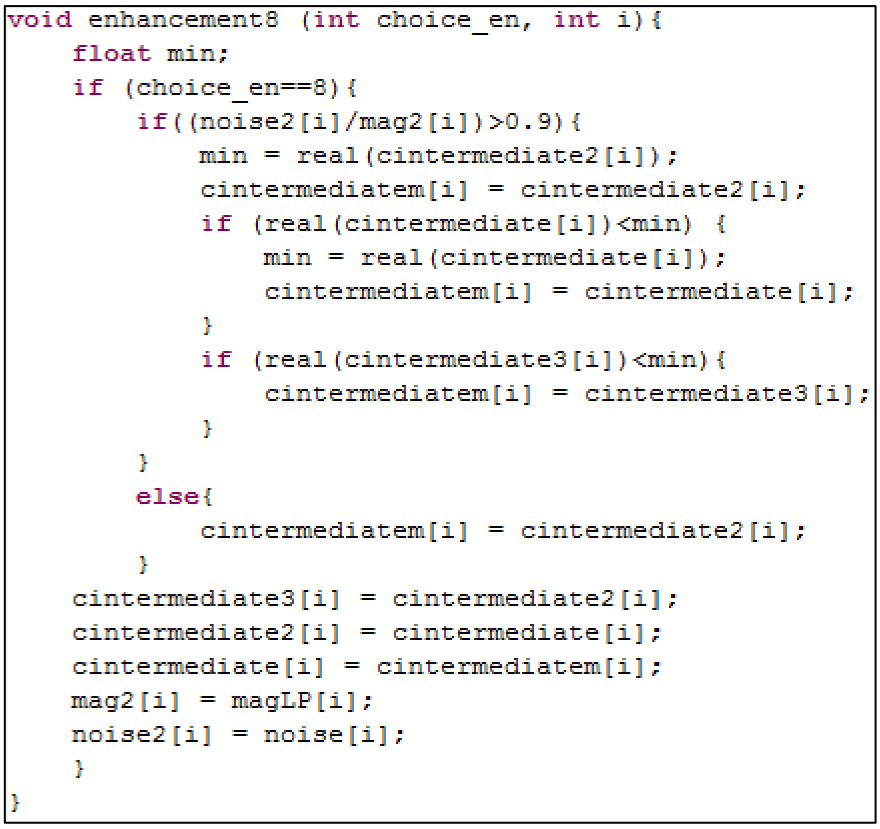
This can be achieved by changing the number of **count** iterations. Currently it is set to 313, rounded up from 312.5 calculated from $\frac{Frame\ Delay}{TFRAME}$. By changing the **count** iterations to a lower value such as 251, it is possible to reduce the frame duration, in this case from 2.5 seconds to 2.0 seconds.

This reduces the time taken for filtering to begin, but also reduces the accuracy of the noise estimation, so it was decided to retain the original **count** range from 0 to 312.

Enhancement 8

Minimising $Y(\omega)$ from 3 adjacent frames if $\frac{|N(\omega)|}{|X(\omega)|}$ exceeds a certain value.

A single frame delay is implemented as the subsequent $Y(\omega)$ is required to calculate the current one.



```
void enhancement8 (int choice_en, int i){
    float min;
    if (choice_en==8){
        if((noise2[i]/mag2[i])>0.9){
            min = real(cintermediate2[i]);
            cintermediatem[i] = cintermediate2[i];
            if (real(cintermediate[i])<min) {
                min = real(cintermediate[i]);
                cintermediatem[i] = cintermediate[i];
            }
            if (real(cintermediate3[i])<min){
                cintermediatem[i] = cintermediate3[i];
            }
        }
        else{
            cintermediatem[i] = cintermediate2[i];
        }
    cintermediate3[i] = cintermediate2[i];
    cintermediate2[i] = cintermediate[i];
    cintermediate[i] = cintermediatem[i];
    mag2[i] = magLP[i];
    noise2[i] = noise[i];
    }
}
```

Figure 20. Function enhancement8()

The enhancement averages out the output, slightly muffling the sound in order to cancel musical noise, which is assumed to be sharp and relatively high pitched compared to the voice.

Unfortunately when tested, this did muffle the sound overall, but made the voice no easier to hear over the musical noise.

Enhancement 9

Reducing the number of frames that are processed and used to compile the output sample.

This method simply involves removing one of the frames from consideration by excluding it from the minimum comparison for the noise spectrum.

This enhancement, when tested for 3 frames, reduces the time taken for noise filtering to become effective, beginning at approximately 7.5 seconds compared to the previous 10 seconds, but removes less background noise than its 4-frame counterpart.

As such it was decided to retain the current frame number of 4, to maximise noise reduction capabilities.

Conclusion

There are two major compromises to consider. One between overall noise and speed of filtering, and another between background noise and musical noise. Furthermore, there are considerations to be made regarding noise reduction and voice distortion.

Factors	**Pros**	**Cons**
Increase Oversubtraction Factor	More background noise filtered out	More of the speech also filtered out
Musical vs. Background Noise	Musical noise reduced	Less effective filtering of background noise
Reduce time of each minimum buffer estimate (Enhancement 7)	Reduced time of response	Less effective filtering of background noise
Threshold on $\frac{\lvert N(\omega)\rvert}{\lvert X(\omega)\rvert}$ (Enhancement 8)	Musical noise reduced slightly	Speech gets muffled
Reduce number of minimum buffers used to estimate noise (Enhancement 9)	Reduced time of response	Less effective filtering of background noise

Table 1. Compromises

In this project, effectiveness of background noise filtering was prioritised in comparison to speed and musical noise. It was determined that this would produce the best long-term real-time speech enhancement, as generally a system will have a long signal, and have sufficient delay before speech starts to profile the noise.

The final code includes enhancements 2, 4 and 6. It performs the filtering of background noise well, generally reducing consistent noise to nothing. However, the speech gets distorted when noise is comparable to the speech in volume and does not perform very well with the filtering of musical noise.

Anti-Plagiarism Declaration

Declaration: I confirm that this submission is my own work. In it, I give references and citations whenever I refer to or use the published, or unpublished, work of others. I am aware that this course is bound by penalties as set out in the college examination offenses policy.

Signed: Lamber Yang, Yumeng Sun

Appendix A. Enhance.c

```
/*************************************************************************************
                         DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                              IMPERIAL COLLEGE LONDON

                              EE 3.19: Real Time Digital Signal Processing
                                     Dr Paul Mitcheson and Daniel Harvey

                                           PROJECT: Frame Processing

                                     ********* ENHANCE. C **********
                                           Shell for speech enhancement

            Demonstrates overlap-add frame processing (interrupt driven) on the DSK.

 *************************************************************************************
                                     By Danny Harvey: 21 July 2006
                                           Updated for use on CCS v4 Sept 2010
 ************************************************************************************/
/*
 *    You should modify the code so that a speech enhancement project is built
 *  on top of this template.
 */
/**************************** Pre-processor statements ******************************/
//  library required when using calloc
#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

/* Some functions to help with Complex algebra and FFT. */
#include "cmplx.h"
#include "fft_functions.h"

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#define WINCONST 0.85185 /* 0.46/0.54 for Hamming window */
#define FSAMP 8000.0     /* sample frequency, ensure this matches Config for AIC */
#define FFTLEN 256                 /* fft length = frame length 256/8000 = 32 ms*/
#define NFREQ (1+FFTLEN/2)         /* number of frequency bins from a real FFT */
#define OVERSAMP 4                 /* oversampling ratio (2 or 4) */
#define FRAMEINC (FFTLEN/OVERSAMP) /* Frame increment */
#define CIRCBUF (FFTLEN+FRAMEINC)  /* length of I/O buffers */

#define OUTGAIN 16000.0            /* Output gain for DAC */
#define INGAIN  (1.0/16000.0)      /* Input gain for ADC  */
// PI defined here for use in your code
#define PI 3.141592653589793
#define TFRAME FRAMEINC/FSAMP      /* time between calculation of each frame */

#define NOISE_FACTOR 3             /* oversubtraction factor alpha */
#define LAMBDA 0.01                /* gainFactor minimum */
#define TAU 0.05                   /* LPF time constant */

/******************************* Global declarations ********************************/
```

```
/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \

/**********************************************************************/
                   /*   REGISTER                FUNCTION                     SETTINGS
*/

/**********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control        8 KHZ-ensure matches FSAMP */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\

/**********************************************************************/
};

// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

float *inbuffer, *outbuffer;        /* Input/output circular buffers */
float *inframe, *outframe;          /* Input and output frames */
float *inwin, *outwin;              /* Input and output windows */
float ingain, outgain;              /* ADC and DAC gains */
complex *cintermediate;             /* Complex processing buffer */
complex *cintermediate2;            /* Complex processing buffer (for en8) */
complex *cintermediate3;            /* Complex processing buffer (for en8) */
complex *cintermediatem;            /* Complex processing buffer (for en8) */
float *magLP, * mag2;               /* Input magnitude low passed */
float *m1, *m2, *m3, *m4;           /* Minimum buffers */
float *noise, *noise2, *noiseLP;    /* Noise estimation buffers */
float cpufrac;                      /* Fraction of CPU time used */
volatile int io_ptr=0;              /* Input/ouput pointer for circular buffers */
volatile int frame_ptr=0;           /* Frame pointer */
volatile int count=0;               /* Count frames */
float K;                            /* LPF constant */

/******************************* Function prototypes *******************************/
void init_hardware(void);     /* Initialize codec */
void init_HWI(void);          /* Initialize hardware interrupts */
void ISR_AIC(void);           /* Interrupt service routine for codec */
void process_frame(void);     /* Frame processing routine */

//enhancements
void enhancement1or2(int choice_en, float mag, int i);
void enhancement3 (int choice_en, int i);
void enhancement4or5 (int choice_en, int en_option, float* gainFactor, float mag, int
i);
void enhancement6 (int choice_en, float minM, float alpha, int i);
void enhancement8 (int choice_en, int i);

/********************************** Main routine ************************************/
void main()
{
      int k; // used in various for loops

/*  Initialize and zero fill arrays */

      inbuffer    = (float *) calloc(CIRCBUF, sizeof(float));  /* Input array */
    outbuffer     = (float *) calloc(CIRCBUF, sizeof(float));  /* Output array */
      inframe     = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    outframe      = (float *) calloc(FFTLEN, sizeof(float));  /* Array for processing*/
    cintermediate = (complex *) calloc(FFTLEN, sizeof(complex)); /*Complex array */
    cintermediate2= (complex *) calloc(FFTLEN, sizeof(complex)); /*Complex array en8*/
    cintermediate3= (complex *) calloc(FFTLEN, sizeof(complex)); /*Complex array en8*/
    cintermediatem= (complex *) calloc(FFTLEN, sizeof(complex)); /*Complex array en8*/
    inwin   = (float *) calloc(FFTLEN, sizeof(float));  /* Input window */
    outwin  = (float *) calloc(FFTLEN, sizeof(float));  /* Output window */
    magLP   = (float *) calloc(FFTLEN, sizeof(float));  /* Input magnitude low pass */
    mag2    = (float *) calloc(FFTLEN, sizeof(float));  /* Input magnitude (for en8)  */
    m1      = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 1 */
    m2      = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 2 */
    m3      = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 3 */
    m4      = (float *) calloc(FFTLEN, sizeof(float));  /* minimum buffer 4 */
    noise   = (float *) calloc(FFTLEN, sizeof(float));  /* noise estimate */
    noise2  = (float *) calloc(FFTLEN, sizeof(float));  /* noise estimate (for en8)*/
    noiseLP = (float *) calloc(FFTLEN, sizeof(float));  /* noise estimate low pass */

      /* initialize LPF constant */
      K = powf(2.718281828,-TFRAME/TAU);

      /* initialize board and the audio port */
      init_hardware();

      /* initialize hardware interrupts */
      init_HWI();

      /* initialize algorithm constants */

      for (k=0;k<FFTLEN;k++)
      {
      inwin[k] = sqrt((1.0-WINCONST*cos(PI*(2*k+1)/FFTLEN))/OVERSAMP);
      outwin[k] = inwin[k];
      }
      ingain=INGAIN;
      outgain=OUTGAIN;


      /* main loop, wait for interrupt */
      while(1)    process_frame();
}

/********************************** init_hardware() **********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

      /* Function below sets the number of bits in word used by MSBSP (serial port) for
      receives from AIC23 (audio port). We are using a 32 bit packet containing two
      16 bit numbers hence 32BIT is set for  receive */
      MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

      /* Configures interrupt to activate on each consecutive available 32 bits
      from Audio port hence an interrupt is generated for each L & R sample pair */
      MCBSP_FSETS(SPCR1, RINTM, FRM);

      /* These commands do the same thing as above but applied to data transfers to the
      audio port */
      MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
      MCBSP_FSETS(SPCR1, XINTM, FRM);


}
```

```
/********************************** init_HWI() **************************************/
void init_HWI(void)
{
	IRQ_globalDisable();			// Globally disables interrupts
	IRQ_nmiEnable();			// Enables the NMI interrupt (used by the
debugger)
	IRQ_map(IRQ_EVT_RINT1,4);		// Maps an event to a physical interrupt
	IRQ_enable(IRQ_EVT_RINT1);		// Enables the event
	IRQ_globalEnable();				// Globally enables interrupts

}

/******************************** process_frame() ************************************/
void process_frame(void)
{
	int i, k, m;
	int io_ptr0;
	float mag, minM, gainFactor, alpha;

	/* work out fraction of available CPU time used by algorithm */
	cpufrac = ((float) (io_ptr & (FRAMEINC - 1)))/FRAMEINC;

	/* wait until io_ptr is at the start of the current frame */
	while((io_ptr/FRAMEINC) != frame_ptr);

	/* then increment the framecount (wrapping if required) */
	if (++frame_ptr >= (CIRCBUF/FRAMEINC)) frame_ptr=0;

 /* save a pointer to the position in the I/O buffers (inbuffer/outbuffer) where the
data should be read (inbuffer) and saved (outbuffer) for the purpose of processing */
	io_ptr0=frame_ptr * FRAMEINC;

/* copy input data from inbuffer into inframe (starting from the pointer position) */

	m=io_ptr0;
    for (k=0;k<FFTLEN;k++)
	{
		inframe[k] = inbuffer[m] * inwin[k];
		if (++m >= CIRCBUF) m=0; /* wrap if required */
	}

/************************* DO PROCESSING OF FRAME  HERE **************************/

	/* please add your code, at the moment the code simply copies the input to the
	ouptut with no processing */

	for (i=0; i<FFTLEN; i++){
		cintermediate[i]= cmplx(inframe[i], 0);
	}
	fft(FFTLEN, cintermediate);

	//for every element in the spectrum
	for (i=0; i<FFTLEN; i++){

		//evaluate magnitude of element
		mag = cabs(cintermediate[i]);
		//low pass filtered magnitude
	    enhancement1or2(2, mag, i);

		//evaluate appropriate element of corresponding minimum buffer
		if (++count >= 313) { //313 for 2.5 seconds, 251 for 2.0 seconds
			count=0;
			m4[i] = m3[i];
			m3[i] = m2[i];
			m2[i] = m1[i];
			m1[i] = magLP[i];
		}
```

```c
            else {
                  if(magLP[i]<m1[i]) m1[i] = magLP[i];
            }

            //evaluate element of the noise magnitude spectrum
            minM = m1[i];
            if (m2[i]<minM) minM = m2[i];
            if (m3[i]<minM) minM = m3[i];
            if (m4[i]<minM) minM = m4[i];

            //variable oversubtraction factor
            enhancement6(6, minM, alpha, i);

            //low pass filtered noise
            enhancement3(0, i);

            //calculate gain factor
            enhancement4or5(4, 4, &gainFactor, mag, i);

            //perform noise subtraction for each element
            cintermediate[i]= rmul(gainFactor, cintermediate[i]);

            //reduce musical noise
            enhancement8 (0, i);
      }

      ifft(FFTLEN, cintermediate);

      for (i=0; i<FFTLEN; i++){
            outframe[i]= real(cintermediate[i]);
      }

/*************************************************************************************/

    /* multiply outframe by output window and overlap-add into output buffer */

      m=io_ptr0;

    for (k=0;k<(FFTLEN-FRAMEINC);k++)
      {     /* this loop adds into outbuffer */
            outbuffer[m] = outbuffer[m]+outframe[k]*outwin[k];
            if (++m >= CIRCBUF) m=0; /* wrap if required */
      }
    for (;k<FFTLEN;k++)
      {
            outbuffer[m] = outframe[k]*outwin[k]; /* this loop over-writes outbuffer */
          m++;
      }
}
/*************************** INTERRUPT SERVICE ROUTINE  *****************************/

// Map this to the appropriate interrupt in the CDB file

void ISR_AIC(void)
{
      short sample;
      /* Read and write the ADC and DAC using inbuffer and outbuffer */

      sample = mono_read_16Bit();
      inbuffer[io_ptr] = ((float)sample)*ingain;
            /* write new output data */
      mono_write_16Bit((int)(outbuffer[io_ptr]*outgain));

      /* update io_ptr and check for buffer wraparound */

      if (++io_ptr >= CIRCBUF) io_ptr=0;
}
```

```
/**********************************Enhancements**********************************/

void enhancement1or2(int choice_en, float mag, int i){
      if (choice_en==1){
            magLP[i] = (1-K)*mag+K*magLP[i]; //+ enhancement 1
      }
      else if (choice_en==2){
            magLP[i] = sqrt((1-K)*mag*mag+K*magLP[i]*magLP[i]); //+ enhancement 2
      }
      else{
            magLP[i] = mag; //no enhancement
      }
}

void enhancement3(int choice_en, int i){
      if (choice_en==3){
            noiseLP[i] = (1-K)*noise[i]+K*noiseLP[i]; //+ enhancement 3
            noise[i]=noiseLP[i];
      }
}

void enhancement4or5(int choice_en, int en_option, float* gainFactor, float mag, int
i){
      if (choice_en==4){
        if(en_option==1){
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA*((noise[i])/mag)){
                   *gainFactor = LAMBDA*((noise[i])/mag);
            }
        }
        else if(en_option==2){
                *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA*((magLP[i])/mag)){
                  *gainFactor = LAMBDA*((magLP[i])/mag);
            }
        }
        else if(en_option==3){
            *gainFactor = 1 - ((noise[i])/magLP[i]);
            if (*gainFactor < LAMBDA*((noise[i])/magLP[i])){
                   *gainFactor = LAMBDA*((noise[i])/magLP[i]);
            }
        }
        else if(en_option==4){
            *gainFactor = 1 - ((noise[i])/magLP[i]);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else{
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
      }
      else if (choice_en==5){
        if(en_option==1){
            *gainFactor = sqrt(1 - (noise[i]*noise[i])/(mag*mag));
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else if(en_option==2){
            *gainFactor = sqrt(1 - (noise[i]*noise[i])/(magLP[i]*magLP[i]));
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
        else{
            *gainFactor = 1 - ((noise[i])/mag);
            if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
      }
```

```
        else{
              *gainFactor = 1 - ((noise[i])/mag);
              if (*gainFactor < LAMBDA) *gainFactor = LAMBDA;
        }
}

void enhancement6 (int choice_en, float minM, float alpha, int i){
        if (choice_en==6){
              //quadratic equation (1/127.5^2)*(i-127.5)^2+3
              alpha = (1/(127.5*127.5))*((i-127.5)*(i-127.5))+3;
              //quadratic equation (2/127.5^2)*(i-127.5)^2+2
              //alpha = (2/(127.5*127.5))*((i-127.5)*(i-127.5))+2;
              noise[i] = alpha*minM;
        }
        else{
              noise[i] = NOISE_FACTOR*minM;
        }
}

void enhancement8 (int choice_en, int i){
        float min;
        if (choice_en==8){
              if((noise2[i]/mag2[i])>0.9){
                    min = real(cintermediate2[i]);
                    cintermediatem[i] = cintermediate2[i];
                    if (real(cintermediate[i])<min) {
                          min = real(cintermediate[i]);
                          cintermediatem[i] = cintermediate[i];
                    }
                    if (real(cintermediate3[i])<min){
                          cintermediatem[i] = cintermediate3[i];
                    }
              }
              else{
                    cintermediatem[i] = cintermediate2[i];
              }
        cintermediate3[i] = cintermediate2[i];
        cintermediate2[i] = cintermediate[i];
        cintermediate[i] = cintermediatem[i];
        mag2[i] = magLP[i];
        noise2[i] = noise[i];
        }
}
```