Department of Electrical and Electronic Engineering
Imperial College London

EE3-19

Real Time Digital Signal Processing

Course homepage: http://learn.imperial.ac.uk

Project: Speech Enhancement

Paul D. Mitcheson
paul.mitcheson@imperial.ac.uk
Room 1112, EEE

Imperial College London

Introductory section: Spectral Analysis

You do not need to hand in any report for this introductory section of the project. It carries no marks. However, you must make sure you understand all off the concepts in this section in order to do the project successfully.

Objectives

- Understand the triple-buffering technique for processing frames of data.
- Use an existing complex FFT routine to perform real-time spectral analysis.
- Implement a DFT routine and then improve its performance.

Introduction

In this laboratory you will implement the triple buffering scheme outlined in the lecture, and after checking that it works you will then use an existing complex FFT routine to perform real-time spectral-analysis. Finally you will implement your own bespoke DFT function and compare its performance to the library FFT algorithms.

Setting up the Project

1. Make a copy of the ***lab5*** project folder and copy it into the same root. Rename the folder ***project_pt1***
2. Recycle the power on your DSK, ensure the USB lead is connected then open Code composer studio.
3. For your workspace browse to ***h:\RTDSPLab\project_pt1***
4. Navigate (using windows explorer) to the folder ***H:\ RTDSPLab \project_pt1 \RTDSP*** and delete the files ***intio.c***, ***compiler-linker.txt*** and any txt files you created to hold coefficients for previous labs
5. Now extract all the files in ***lab6.zip*** (available online on WebCT) into ***H:\ RTDSPLab \ project_pt1 \RTDSP.*** This will place the template c file ***frame.c*** into your project folder which you will use for the tasks required in this laboratory.

Project Properties

1. In the code composer window open ***frame.c*** and place your mouse cursor somewhere in the window. Select the Menu **Project>Properties**. Select **C/C++ Build** in the tree. In the **C6000 Linker** section choose **File Search Path.**
2. Add the following statements to the **include library file...** section (see figure 1). "${CCS_INSTALL_ROOT}\C6000\dsk6713\FFT\complex.obj"

   "${CCS_INSTALL_ROOT}\C6000\dsk6713\FFT\fft_functions.obj"

   A text file (FFT compiler-linker.txt) was included in the zip file if you wish to copy and paste the text for these includes into CCS. When you have done hit **Apply** then **Ok**

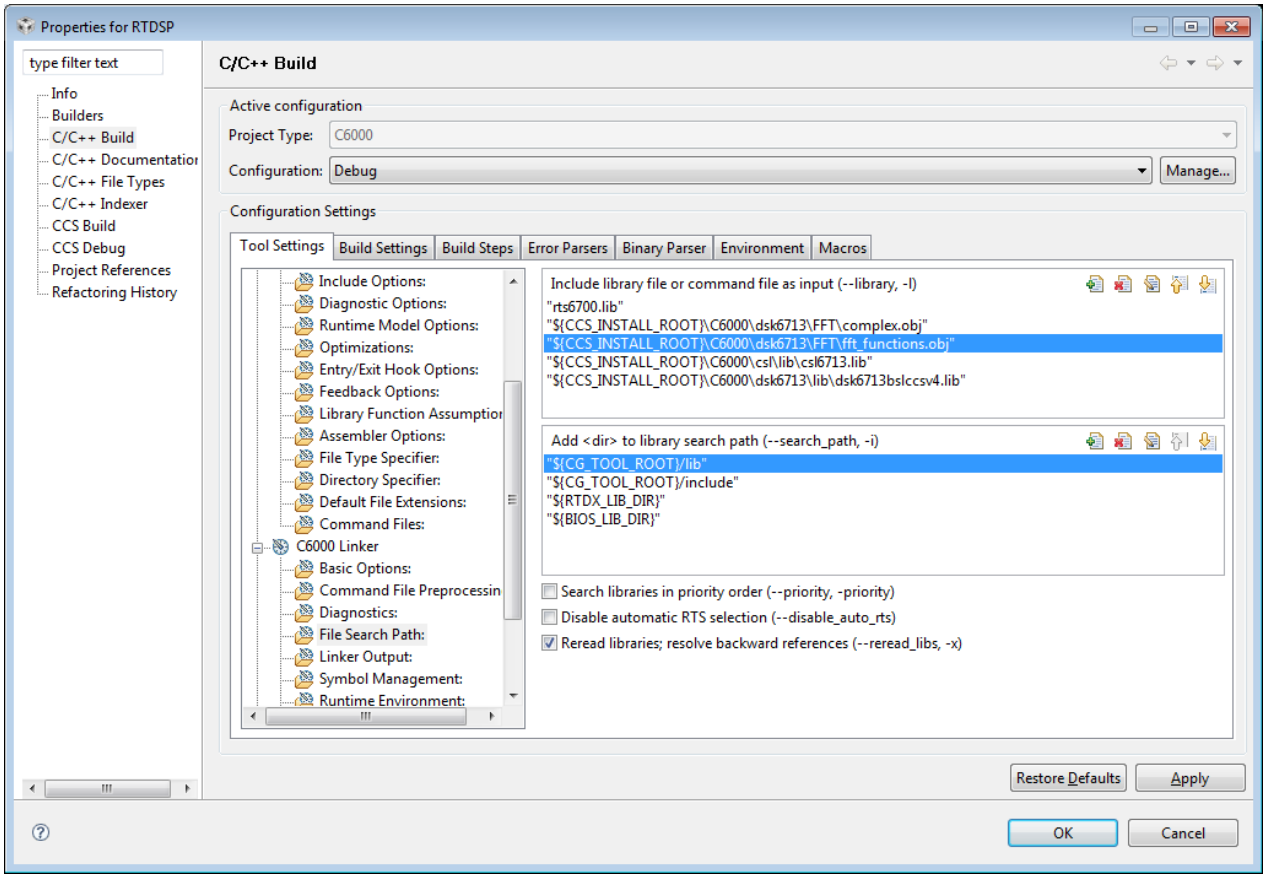


Figure 1: Adding the FFT and Complex code into the project.

The C program shell

You are provided with a program frame.c whose listing is given in Appendix A. This program contains the following functions:

- init_hardware() initialize the audio codec using BSL routines.
- init_HWI() sets up the interrupts.[1]
- init_arrays() allocates memory for the buffers.
- ISR_AIC() the interrupt service routine, reads a sample to the input buffer, writes a sample to the output buffer and updates the buffer index.
- wait_buffer() waits until the input buffer is full after which it then rotates the arrays and does the processing. This processing code will be written by you.
- The main() program initializes the codec, and allocates memory for the three buffers. It then goes into a loop where the function wait_buffer() is executed. Within this function, it waits until the input buffer has been filled by the ISR (by monitoring the global variable input) and then rotates the data buffers and (potentially) does processing on the buffers.

Exercise 1 - Checking Frame processing

1. Build the project and download it to the DSP.
2. Attach a signal generator to the input and a scope to the output. Apply a suitable sine wave. Ensure that you do not exceed 2V $_{RMS}$ on the input!
3. Run the program. Currently no processing is done on the frames of the data, so you should get an output that is the same as the input (although it will be delayed).
4. Halt the program. Use the CSS graph facility (**Tools→Graph→Single Time**) to display the 128 samples that are stored in the intermediate buffer.
5. Ensure that you understand what this program does and how the triple buffering is implemented. This will form the basis of your project over the next two weeks!

[1] Check that the physical interrupt that is mapped in the implementation of this function matches the interrupt you used in the configuration file.

Exercise 2- Using pre-compiled library functions

In this exercise you will use an existing routine to perform a complex FFT on frames of data, and use the CCS graph facility to plot the magnitude of the FFT.

The Fast Fourier routines

The object file fft_functions.obj contains optimized code to perform FFT operations. The corresponding header file containing the function definitions is listed in appendix B:

- Fft(N,*X) Calculates the FFT of its input buffer X which contains complex-valued time domain samples. The parameter N contains the number of data values in X. The complex-valued output spectrum is returned by overwriting the input buffer X.
- Ifft(N, *X) Calculates them inverse FFT of its input buffer X which contains complex-valued frequency domain data. The parameter N contains the number of data values in X. The complex-valued output time series is returned by overwriting the input buffer X.

Representing complex data

Complex data representation is achieved by defining an appropriate data structure and functions within complex.obj whose corresponding header file complx.h is listed in appendix C.

Using these definitions and array of 10 complex values, for example, would be defined as:

```
Complex C[10];
```

To assign the first element of C as 1 + 2j, for example, the following code would be used:

```
C[0].r = 1;
C[0].i = 2;
```

Or the assignment could be achieved using the cmplx function:

```
C[0] = cmplx(1,2);
```

Implementing a spectrum analyser

1. Within wait_buffer() add appropriate code (where is says DO PROCESSING OF FRAME HERE) to perform a FFT on the data stored in the intermediate buffer. You will have to copy the data in intermediate to a new complex-valued array before the call.
2. After the FFT call, calculate the magnitude of the resultant spectrum. (You can use the function cabs() to calculate the absolute value of a single complex number) and store the result in a global variable called mag which is an array of floats.
3. Build and run your code. Note that you should not be modifying the intermediate buffer, so the output will still be the same as the input.
4. Halt the DSP and graph the data stored in mag. Check that this is producing the output you would expect.
5. Restart the DSP and change the frequency of the input sine wave. Halt the DSP again and check the graph has changed appropriately. Notice that although your program is performing real time spectral analysis you can only graph the result if you halt the DSP[2].
6. After the magnitude is calculated add a function called ifft and then store the real part of the result in intermediate. Within the wait_buffer() code you should now be performing an FFT on the data in intermediate, calculating the magnitude of the spectrum, performing an inverse FFT, and then storing the real part of the result back in intermediate. The net effect is that the output is just the same as the input. Run your code and check that this is indeed the case!

[2] There is a feature of CCS that allows you to view data in real time. This feature, called RTDX (Real Time Data Exchange) will not be explored in these labs.

Exercise 3 - DFT implementation

You should now code an ordinary DFT algorithm and IDFT algorithm and use this instead of the binary FFT algorithms you were given. They will, of course, be slower than the FFT and IFFT algorithms. You should find out at what frame size your DFT algorithms are no longer fast enough to process all the data in a frame on this processor (at the sampling rate of 8 KHZ).

How does this compare to the FFT implementation?

Try to improve your algorithm performance. Try to think of ways of reducing the number of execution cycles required to process each frame.[3]

Deliverables

There are no deliverables for this introductory section of the project. Just try to make sure you can answer the questions above as you will need to understand these things in order to be able to do the speech enhancement,

[3] Hint: One saving that could be made is by writing your own cabs() function.

Appendix A: Listing of Frame.c

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

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

                            LAB 6: Frame Processing

                         ********* F R A M E. C **********

             Demonstrates Frame Processing (Interrupt driven) on the DSK.

 *************************************************************************************
             Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
             Updated for ccsV4 Sept 2010
 *************************************************************************************/
/*
 *     You should modify the code so that an FFT is applied to an input frame
 *  which is then IFFT'd and sent to the audio port.
 */
/***************************** Pre-processor statements ******************************/

//  Included so program can make use of DSP/BIOS configuration tool.
#include <stdlib.h>
#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>


/*********************************** 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               */\
```

```
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\

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

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

// PI defined here for use in your code
#define PI 3.141592653589793

#define BUFLEN 128  /* Frame buffer length must be even for real fft */


/* Pointers to data buffers */
float *input;
float *intermediate;
float *output;
volatile int index = 0;


 /****************************** Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
void init_arrays(void);
void wait_buffer(void);

/********************************** Main routine ***********************************/
void main()
{

  /* setup arrays */
  init_arrays();

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

  /* initialize hardware interrupts */
  init_HWI();


  /* loop indefinitely, waiting for interrupts */
  while(1)
  {
             wait_buffer();
  };

}



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

}
/************************** Allocate memory for arrays *****************************/
void init_arrays(void)
{
       input        = (float *) calloc(BUFLEN, sizeof(float)); /* Input array */
       output       = (float *) calloc(BUFLEN, sizeof(float)); /* Output array */
       intermediate = (float *) calloc(BUFLEN, sizeof(float)); /* Array for processing*/
}

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

// Map this to the appropriate interrupt in the DSP BIOS

void ISR_AIC(void)
{
       short sample;
       float scale = 11585;

       sample = mono_read_16Bit();

       /* add new data to input buffer
          and scale so that 1v ~= 1.0 */
       input[index] = ((float)sample)/scale;

       /* write new output data */
       mono_write_16Bit((short)(output[index]*scale));

       /* update index and check for full buffer */
       if (++index == BUFLEN)
              index=0;
}

/******************* Wait for buffer of data to be input/output *********************/
void wait_buffer(void)
{
       float *p;

       /* wait for array index to be set to zero by ISR */
       while(index);

       /* rotate data arrays */
       p = input;
       input = output;
       output = intermediate;
       intermediate = p;

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

```
                        /*please add your code */

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

       /* wait here in case next sample has not yet been read in */
       while(!index);
}
```

Appendix B: fft_functions.h

```
void fft(int N, complex *X);
void ifft(int N, complex *X);
```

Appendix C: complx.h

```
/* cmplx.h - complex arithmetic declarations */

#include <math.h>

typedef struct {
 float r;
 float i;
} complex;

float cabs(complex);

complex cmplx(float, float);          /* define complex number */
complex conjg(complex);               /* complex conjugate */

complex cadd(complex, complex);       /* complex addition */
complex csub(complex, complex);       /* complex subtraction */
complex cmul(complex, complex);       /* complex multiplication */
complex cdiv(complex, complex);       /* complex division */

complex rmul(float, complex);         /* multiplication by real */
complex rdiv(complex, float);         /* division by real */

float real(complex);                  /* real part */
float aimag(complex);                 /* imaginary part */

complex cexp(complex);                /* complex exponential */
```

Main section: Speech Enhancement Project

You must hand in a formal report for this section. To solve the enhancement problems read the cited documents and try to interpret them to implement and understand the proposed algorithms. Note that we have found that the code checking tools we use will detect even the slightest plagiarised section of code.

Introduction

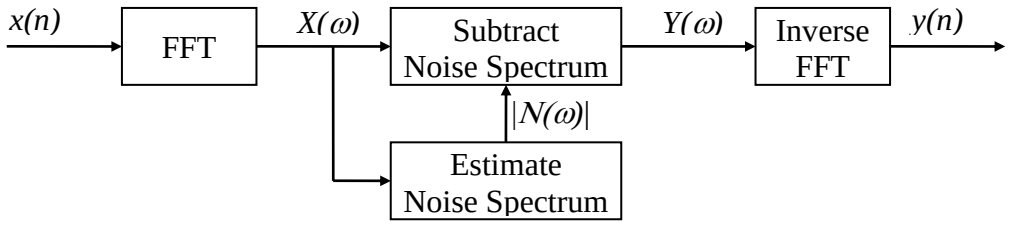
Telephones are increasingly being used in noisy environments such as cars, airports and undergraduate laboratories! The aim of this project is to implement a real-time system that will reduce the background noise in a speech signal while leaving the signal itself intact: this process is called *speech enhancement*.

Algorithm

Many different algorithms have been proposed for speech enhancement: the one that we will use is known as *spectral subtraction*. This technique operates in the frequency domain and makes the assumption that the spectrum of the input signal can be expressed as the sum of the speech spectrum and the noise spectrum.

The procedure is illustrated in the diagram below and contains two tricky parts:

- estimating the spectrum of the background noise
- subtracting the noise spectrum from the speech

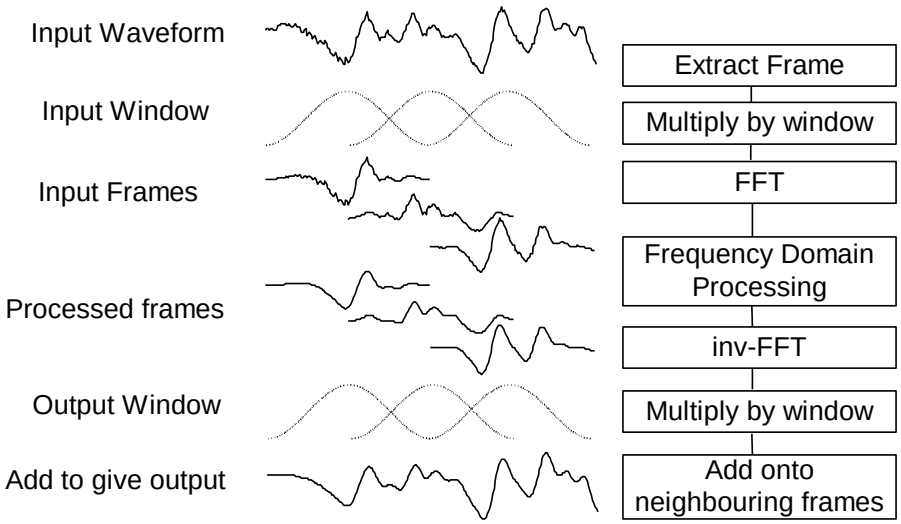


The sample rate of the system is 8 kHz[4] and a 256-point Fourier transform is performed on the input signal every 64 samples (8 ms).

Overlap-Add Processing

To perform frequency-domain processing, it is necessary to split the continuous time-domain signal up into overlapping chunks called frames. After processing, the frames are then reassembled to create a continuous output signal. To avoid spectral artefacts, we multiply the frame by a window function before performing the FFT and again after performing the inverse-FFT.

[4] The sample rate of the C6713 should be kept at 8KHZ. This lab will not be exploring any of the other sampling frequencies available on the hardware.



The output signal is thus formed by adding together a continuous stream of 256-sample frames each of which has been multiplied by both an input and an output window. If we choose the windows to be the square root of a Hamming window:

$$\sqrt{1-0.85185\cos\left((2k+1)\pi/N\right)} \quad \text{for} \quad k=0,\cdots,N-1$$

then the overlapped windows will sum to a constant and the output signal will be undistorted by the framing process.

In the diagram above, each frame starts half a frame later than the previous one giving an *oversampling ratio* of 2. This normally gives acceptable results but can introduce distortion if the processing alters the gain of a particular frequency bin abruptly between successive frames. It is therefore more common to use an oversampling ratio of 4 in which each frame starts only a quarter of a frame after the previous one. In this case, each output sample is the sum of contributions from four successive frames.

Subtracting the Noise Spectrum

The basic idea is just to subtract the noise off the input signal:

$$Y(\omega) = X(\omega) - N(\omega)$$

Unfortunately we don't know the correct phase of the noise signal so we subtract the magnitudes and leave the phase of X alone:

$$Y(\omega) = X(\omega) \times \frac{|X(\omega)| - |N(\omega)|}{|X(\omega)|} = X(\omega) \times \left(1 - \frac{|N(\omega)|}{|X(\omega)|}\right) = X(\omega) \times g(\omega)$$

We can regard $g(\omega)$ as a frequency-dependent gain factor, so this is really just a form of zero-phase filtering.

A further problem is that it is quite possible for the multiplicative factor in the above expression to go negative from time to time. To avoid this, we actually use the following formula:

$$g(\omega) = \max\left(\lambda, 1 - \frac{|N(\omega)|}{|X(\omega)|}\right)$$

where the constant λ is typically 0.01 to 0.1.

Estimating the noise spectrum

When someone is speaking, they inevitably have to pause for breath from time to time. We can use these gaps in the speech to estimate the background noise. One way of doing so is to design a Voice Activity Detector (VAD) which identifies whether or not speech is present in a signal. Unfortunately a reliable VAD is exceptionally difficult to make and so we choose an easier approach [4].

For each frequency bin in the Fourier transform, we determine the minimum magnitude that has been present in any frame over the past ten seconds or so. Under the assumption than no one ever talks for more than ten seconds without a break, this spectral minimum will correspond to the minimum noise amplitude that occurred during non-speech intervals. Since this will underestimate the *average* noise magnitude, we must multiply by a compensating factor before using it in the formulae above.

Determining the precise minimum over the past ten seconds requires storing all spectra from this interval which is unrealistic. We can determine an approximate minimum with far less storage as follows. We store four estimates of the minimum spectrum $M_i(\omega)$ where i=1,2,3,4. For each frame, we update M_1 by:

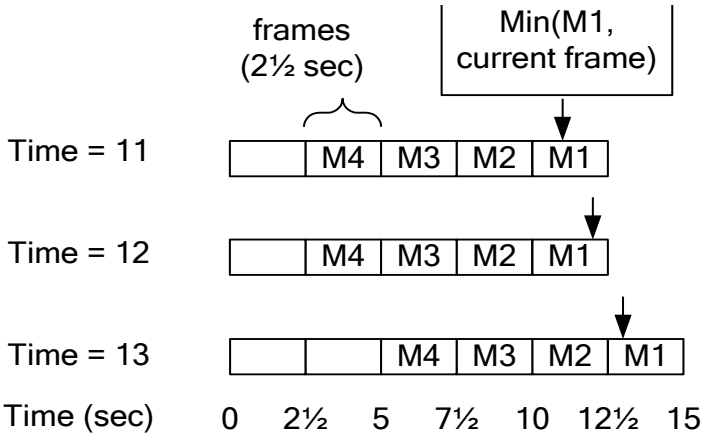
$$M_1(\omega) = \min\left(|X(\omega)|, M_1(\omega)\right)$$

Every 2.5 seconds, we transfer $M_i(\omega)$ to $M_{i+1}(\omega)$ and set $M_1(\omega)=|X(\omega)|$.

We can therefore estimate the noise spectrum as:

$$|N(\omega)| = \alpha \min_{i=1..4}\left(M_i(\omega)\right)$$

where the factor α corrects for the underestimation of the noise discussed above. The factor α may need to be as high as 20 but the use of enhancement (1) described at the end of this sheet will allow it to be reduced to around 2 and will give more reliable estimates.



The diagram above indicates the situation at 11 sec, 12 sec and 13 sec after the start of the program. In the first case, at time 11 seconds, M4, M3 and M2 contain the minimum spectrum that occurred in the three intervals 2½ to 5 seconds, 5 to 7½ seconds and 7½ to 10 seconds (the minimum of 312 frames in each case). M1 contains the minimum spectrum that occurred between 10 and 11 seconds. Taking the minimum of all four buffers therefore gives the minimum spectrum over the 8½ second interval from 2½ to 11 seconds.

In the second case, M4, M3 and M2 are unchanged, but M1 now contains the minimum from 10 to 12 and so taking the minimum of all four buffers gives the minimum spectrum over the 9½ second interval from 2½ to 12 seconds.

At time 12½, we transfer M3 to M4, M2 to M3 and M1 to M2, so in the third case the minimum of all four buffers gives the minimum spectrum over the 8 second interval from 5 to 13 seconds.

Skeleton Program

A skeleton program is available as ***enhance.c***. This program performs the input/output buffering and interrupt handling that are needed. It does not perform the FFTs, noise estimation or spectral subtraction: that is for you to implement. All the constant parameters associated with the algorithm are specified in a definitions block near the start of the program: it is bad practice to bury peculiar constants in the middle of your code so you should follow this procedure for any other constants that you need to add. Some of the constants are used to define buffer lengths while others are used in arithmetic expressions. Some of the latter (e.g. INGAIN and OUTGAIN) are copied into global variables in the initialisation section of the main program: the reason for this is so that they can be altered in real time while the program is running. Note that when defining a constant to equal an expression, you should enclose the expression in parentheses to avoid unexpected results if it is used in an arithmetic expression.

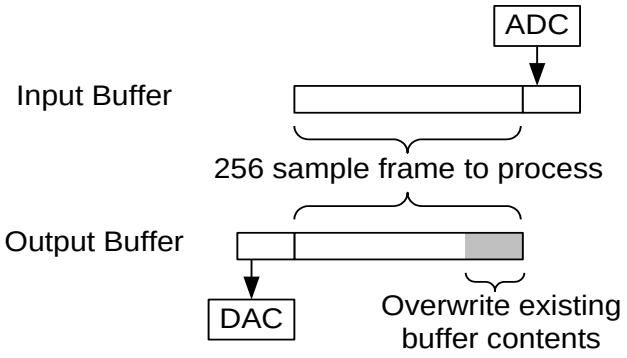
Setting up the project file

1. Make a copy of the ***project_pt1*** folder and copy it into the same root. Rename the folder ***project_pt2***
2. Recycle the power on your DSK, ensure the USB lead is connected then open Code composer studio.
3. For your workspace browse to ***h:\RTDSPLab\project_pt2***
4. Navigate (using windows explorer) to the folder ***H:\ RTDSPLab \project_pt2 \RTDSP*** and delete the file ***frame.c*** that you used in the previous lab
5. Now extract all the files in ***project.zip*** (available online on WebCT) into ***H:\ RTDSPLab \ project_pt1 \RTDSP.*** This will place the template c file ***enhance.c*** into your project folder which you will use for the tasks required in this laboratory

Now compile the project which should be successful if you have done everything correctly. Ensure that the program has loaded onto the hardware. Run the code. Apply an appropriate (i.e. correct frequency and voltage) test signal to the line inputs. You should see an identical signal on the line out jack.

Input/Output Buffers

With an oversampling rate of 4, we need to process six quarter-frames at any given moment:



While the A/D converter is transferring samples into the current ¼-frame, we process the previous four ¼-frames as indicated in the diagram. We add the processed frame onto the existing contents of the output buffer (from previous frames). For the last quarter of the frame, there is no data from previous frames and so we overwrite whatever information happens to be in the buffer instead of adding onto it. While all of this is happening, the previous ¼-frame, which is now completely specified, is sent to the D/A converter. Our algorithm therefore has a 1¼-frame delay that is independent of the processor speed (this is in addition to the codec delay which is quite large).

Buffer Synchronization

Notice that both the input and output buffer need only be five ¼-frames long. We implement them as circular buffers and reset the variable **io_ptr** to zero in the interrupt service routine whenever it increments beyond the end of the buffer. When we have finished processing a frame, we need to wait until the ADC and DAC have finished the current ¼-frame. We do this by checking the value of **io_ptr/64** to see which section of the buffer it has reached. We also use **io_ptr** to calculate the value of **cpufrac** which tells us what fraction of the available CPU time we are using: you can monitor this from a watch window.

Objectives and Milestones

The aim of this project is to implement the spectral subtraction technique described above and then to improve it to obtain the best possible performance on the test files provided.

You may find it easier to implement the two parts of the algorithm independently at first and check that they work correctly. As an initial test, you could implement a simple frequency-domain filter by setting some of the FFT spectrum values to zero.

Test Data

A number of test signals are available along with descriptions on WebCT in ***wav.zip***

You can use Windows to replay the test signals via the soundcard.

Enhancements

You may wish to evaluate some of the following enhancements that researchers have suggested. Some of them are a good idea but others make things worse. It is helpful to write your program with software switches so that you can turn particular enhancements on and off in real time. It is possible to have two sets of parameters stored so that you can make instant comparisons between them.

1. Use a low-pass filtered version of $|X(\omega)|$ when calculating the noise estimate. Note that this low-pass filter is operating on successive frames not on the speech samples themselves. If $P_t(\omega)$ is the low-pass filtered input estimate for frame t, then $P_t(\omega) = (1-k)\times|X(\omega)| + k \times P_{t-1}(\omega)$ where $k = \exp(-T/\tau)$ is the z-plane pole for time constant τ and frame rate T. You can calculate T from the FFT length, sample rate and oversampling ratio. With this enhancement you will be able to reduce the oversubtraction factor α described above. The value of T is defined in the C program as **TFRAME**: you can calculate the value of k in the initialisation section of the main program. A plausible time constant to use is in the range 20 to 80 ms.
2. The above low-pass filtering can be performed in the power domain rather than the magnitude domain. That is, you can low-pass filter $|X(\omega)|^2$ and then take the square root to find $P(\omega)$.
3. Low pass filter the noise estimate $|N(\omega)|$ to avoid abrupt discontinuities when the minimization buffers rotate. This will only have a noticeable effect if the noise level is very variable.
4. Instead of setting $g(\omega) = \max\left(\lambda, 1-\frac{|N(\omega)|}{|X(\omega)|}\right)$, set it to $\max\left(\lambda\frac{|N(\omega)|}{|X(\omega)|}, 1-\frac{|N(\omega)|}{|X(\omega)|}\right)$, $\max\left(\lambda\frac{|P(\omega)|}{|X(\omega)|}, 1-\frac{|N(\omega)|}{|X(\omega)|}\right)$, $\max\left(\lambda\frac{|N(\omega)|}{|P(\omega)|}, 1-\frac{|N(\omega)|}{|P(\omega)|}\right)$, or $\max\left(\lambda, 1-\frac{|N(\omega)|}{|P(\omega)|}\right)$.
   In all cases you may wish to calculate $P(\omega)$ using a different (probably longer) time constant than used for enhancement (1).

5. Calculate g in the power domain rather than the magnitude domain, i.e. set $g(\omega) = \max\left(\lambda, \sqrt{1 - \frac{|N(\omega)|^2}{|X(\omega)|^2}}\right)$ or a similar modification of an expression from (4).
6. Deliberately overestimate the noise level (by increasing α) at those low frequency bins that have a poor signal-to-noise ratio. This *oversubtraction* technique can reduce the "musical noise" artifacts that are introduced by spectral subtraction.
7. Evaluate different frame lengths: according to [1], short frames sound rough with increased musical noise, while long frames sound slurred.
8. Reduce musical noise by applying the "residual noise reduction" described in [2]: if $\frac{|N(\omega)|}{|X(\omega)|}$ exceeds some threshold, then replace $Y(\omega)$ by its minimum calculated value in three adjacent frames (this entails adding an additional 1-frame delay since you need $Y(\omega)$ from the next frame in order to calculate the output in the current frame).
9. You can estimate the noise by taking the minimum spectrum over a shorter period. This makes the system respond more quickly to a rise in noise level but may introduce distortion into the speech.
10. Several other possible enhancements are described in the references listed below.

Assessment

At the end of the project period, your program by evaluated by listening to its effect on a number of test files. You are not allowed to have different programs or parameter values for different test files.

You need to write a report (1 report per pair) explaining how your program works, the evaluations you performed and the reasons for you choice of parameters. You should submit a copy of your source code as an appendix to your report (which does not count in the page limit). Your report should give formulae that specify precisely what quantities are calculated in your program. It is unlikely that a single set of parameters will be optimum for all types of speech and noise; your report should make clear what compromises you make in choosing your final algorithm. You must also try to give explanations of your understanding of why the enhancements work (or do not work) better than the simple algorithm. See the mark sheet for marking details.

Please keep the main body of your report to no more than 12 pages. This is ample space in which to describe what you have done, if you write clearly and concisely. (Remember that research papers, which usually contain around 1 year's worth of work are usually only 6 pages long)

Check Blackboard for the mark allocation sheets.

References

These references are available on WebCT in file ***refs.zip***

[1] Berouti,M. Schwartz,R. & Makhoul,J.,"*Enhancement of Speech Corrupted by Acoustic Noise*", Proc ICASSP, pp208-211, 1979.

[2] Boll,S.F.,"*Suppression of Acoustic Noise in Speech using Spectral Subtraction*", IEEE Trans ASSP 27(2):113-120, April 1979.

[3] Lockwoord, P. & Boudy,J.,"*Experiments with a Nonlinear Spectral Subtractor (NSS), Hidden Markov Models and the projection, for robust speech recognition in cars*", Speech Communication, 11, pp215-228, Elsevier 1992

[4] Martin, R., "*Spectral Subtraction Based on Minimum Statistics*", Signal Processing VII: Theories and Applications, pp1182-1185, Holt, M., Cowan, C., Grant, P. and Sandham, W. (Eds.), 1994

Mike Brookes, May 2001

Darren Ward, March 2002

Paul D. Mitcheson, March 2006

Daniel Harvey August 2006/2009

Updated for Windows 7 Daniel Harvey Feb 2010