ELEN90058 Signal Processing

Workshop 2, Weeks 5-7

Introduction to filters

Aim: To investigate time and frequency domain characteristics of some simple digital filters and practical applications of the filters.

Workshop: The workshop consists of three parts. Part A consists of a number of pen and paper questions combined with Matlab and serves as preparation for the two other parts. Part B is about design and implementation of echo filters and reverberators using Matlab and Analog Devices' BF533 DSP, and part C is about BF533 implementation of notch filters for removal of sinusoidal disturbances. You can do Part C before Part B if you prefer since they do not depend on each other.

Assessment: The workshop is assessed based on two written reports (see below) and on demonstrations of the algorithms implemented on the DSP board in the workshop. The lab constitute 6% of the overall assessment of the course.

Submission of the reports: The students are to submit the report for part A to the workshop demonstrator in week 6 (week starting on the 28th of August) in the workshop. **Note however, that you are expected to have completed most of part A before the workshop starts in week 5.** The reports for part B and C should be submitted to the workshop demonstrator in week 8 (week starting on the 11th of September) in the workshop. You also have to sign a declaration that the report is your own work.

Student groups: The students should work in groups of three.

Reports: There are two reports. The report for part A should be handed to the workshop demonstrator in week 6. Keep a copy of this report for yourself and include it as an appendix to the second report for part B and C which you can refer to as required. The reports should be clearly written and explain in a logical way how the different tasks in the workshop have been carried out. Choices you made should be explained and justified. Calculations of filter coefficients, time delays etc. should be clearly shown. Results obtained using Matlab should be explained, i.e. it is not sufficient to copy the output of Matlab without further explanations of what the numbers or graphs mean. Figures should be included where it is appropriate. The Matlab and c code should be submitted via LMS and also be included in an appendix. See also the note on LMS about workshop expectations and reports.

Collaboration between and within groups: It is perfectly OK to discuss problems and possible solutions with other groups. However, each group has to carry out the workshop independently, and e.g. copying of other groups' mathematical derivations, c code or Matlab code is not acceptable. Both group members should

do an equal amount of work on both the workshop tasks themselves and the writing of the report.

A Preparations

Question 1

Let $H(z)$ be the transfer function of a filter, and let $H(e^{j\omega})$ be the corresponding frequency response.

a) Consider now $H(z^D)$.

(i) What is the effect in the time domain of replacing z with z^D?

(ii) What is the effect on the frequency response?

b) Let $H(z)$ be an ideal low pass filter with cut off frequency $\pi/4$. Sketch the magnitude response of the following systems:

(i) $H(z^3)$

(ii) $H(z)H(z^3)$

(iii) $H(-z)H(z^3)$

c) (Optional). Find the error(s) in question **b)**.

Question 2

Let

$$H_1(z) = 1 + \alpha z^{-1}$$

a) Find expressions for the the frequency response and impulse response coefficients of this filter.

b) Consider now

$$H(z) = H_1(z^D) = 1 + \alpha z^{-D}, \quad \alpha > 0$$

Determine the frequency response $H(e^{j\omega})$. What are the maximum and minimum values of the magnitude response? How many peaks and dips of the magnitude response occur in the range $0 \leq \omega < 2\pi$? What are the locations of the peaks and dips? What are the zeros of the transfer function? Any comments.

Let $D = 8$.Use Matlab to plot (i) the zeros of the transfer function (ii) the impulse response and (iii) the magnitude and phase response on the interval $[0, 2\pi)$. (Hint the functions `zplane`, `freqz` and `impz` may prove useful.)

Question 3

Let

$$H_2(z) = \frac{1}{1 + \alpha z^{-1}}$$

be the transfer function of a causal filter.

a) Calculate the impulse response coefficients and the frequency response for this filter.

b) Consider now $H_2(z^D)$. Use Matlab to plot the impulse response and the magnitude and phase response for $D = 8$.

Question 4

Given the causal and stable filter

$$H_3(z) = \frac{z^{-1} - \alpha}{1 - \alpha z^{-1}} \quad 0 < \alpha < 1$$

a) Show that $|H_3(e^{j\omega})|^2 = 1$.

b) Show that the impulse response is given by

$$h[n] = \begin{cases} 0 & n < 0 \\ -\alpha & n = 0 \\ (1 - \alpha^2)\alpha^{n-1} & n > 0 \end{cases}$$

c) Use Matlab to plot the impulse response and the frequency response of $H_3(z^D)$ with $D = 8$.

Question 5

An audio signal is sampled at 44.1kHz. The signal is corrupted by a pure sinusoidal disturbance situated at 330 Hz. Calculate the filter coefficients of a second order IIR notch filter for removal of the sinusoidal disturbance with 3 dB bandwidth of

(i) 0.1π.

(ii) 0.01π.

(iii) 0.005π.

Use Matlab to plot the pole/zero map of the filters and their magnitude and phase responses. Comment upon the results.

Write a Matlab script which calculates the filter coefficients for a second order notch filter. The input variables should be the 3dB bandwith (in Hz), the sampling frequency (in Hz) and the notch frequency (in Hz). Implement the filter and test it on the signal stored on `testsignal.mat` on LMS. The signal is sampled at 8192 Hz and corrupted by a sinusoid with frequency 550 Hz. Let the 3dB bandwith be around 30 Hz (roughly the distance between two semi-tones around 550Hz). The matlab command `sound` can be used for playing the signal before and after filtering.

B Implementation of echo filters and reverberators using Matlab and AD BF533

References:

Mitra S.K (2011). Digital signal processing applications, Section 5 in the file applications2.pdf on the CD accompanying Mitra's book.

Moorer J.A. (1977). "Signal Processing Aspects of Computer Music: A survey", *Proceedings of the IEEE*, Vol. 65, No. 8, pp. 1108-1137.

Music generated in an inert studio does not sound natural compared to the music performed inside a room. In the latter case, the sound waves propagate in all directions and reach the listener from various directions and at various times, depending on the distance travelled by the sound waves from the source to the listener. The sound wave coming directly to the listener, called the direct sound, reaches first and determines the listener's perception of the location and nature of the sound source. This is followed by a few closely spaced echoes called early reflections, generated by reflections of sound waves from all sides of the room and reaching the listener at irregular times. These echoes provide the listener's subconscious cues as to the size of the room. After these early reflections, more and more densely packed echoes reach the listener due to multiple reflections. The latter group of echoes is referred to as the reverberation. The amplitude of the echoes decay exponentially with time as a result of attenuation at each reflection.

In order to make sound recorded in an inert studio sound more natural, artificial echo and reverberation effects are often added to studio recordings. In this part you will use Matlab and AD BF533 to investigate techniques and methods for creating these artificial effects. In the tasks below you are going to use the speech signal stored on the file `speech.wav` as a test signal. This signal is sampled at 8 kHz, and you can download it from LMS. You can load this signal into Matlab using the command

```
[x,fs,nbits]=wavread('speech.wav');
```

`x` contains the signal, `fs` is the sampling frequency and, `nbits` is the number of bits per sample. You can play the signal back using the command

```
wavplay(x,fs)
```

When you test your DSP implementation, you can redirect the audio from the PC to the DSP board. If you need a longer signal for testing you can concatenate `x` with itself a number of times, e.g.

```
xx=[x; x; x; x; x; x];
```

Alternatively, you can use another audio source when you are testing the DSP implementations, but remember that the sampling rate is set to 8 kHz, and you

may experience aliasing effects if you play a music signal.

Note that the Matlab implementation is a batch implementation, that is the whole signal is available for processing, while the DSP implementation is real time, that is the signal becomes available sample by sample. You can choose between doing the task in the order below or do the Matlab parts first and then do the DSP questions. It is not recommended that you do it the other way, since most students will find the Matlab implementation easier, and having done and tested the Matlab implementation you have a fair idea of what the DSP implementation should look and sound like. Before you start on the DSP implementation read Appendix 1 which describes the hardware and software setup and Appendix 2 which contains a number of hints on coding in c.

In this part you have to carry out a number of listening tests. The outcomes of such tests are always subjective, and remember that the human ear is not necessarily behaving according to linear systems theory and that the quality of the speakers also influence the results. Try to be as precise as possible when describing the results of listening tests, but remember that since such test are subjective there is not necessarily a correct answer.

B.1 Workshop tasks

Show your workings on Question 1 to 4 in part A to the demonstrator before starting on this part.

WARNING: Never put on a headset or have the loudspeakers turned up at high volumes before you have verified that the c code is working as expected. If a filter is unstable, very loud and unpleasant audio output is generated.

a) Consider the filter $H_1(z^D)$ in question 2 in part A.

(i) Explain why this filter is sometimes called an echo filter. (Hint: Look at the impulse response.)

(ii) Let $\alpha = 0.75$ and find D such the echo comes 220 ms after the direct sound. Implement the filter in Matlab and test it on the speech signal. Try out a few values of α and D and comment on the effect.

(iii) Connect the DSP board as described in Appendix 1. Redirect the audio from the PC to the DSP board (or use an alternative audio source). Start up CCES, and open the project SPWS2-Echo which can be downloaded from LMS. In order to test the setup, build the project without modifying the skeleton file and run the project on the board. The output to the speakers should be a repeated pattern of 3 seconds of the audio source followed by half a second of silence.

(iv) Implement the filter with $\alpha = 0.75$ on the DSP board. Here and in the other DSP questions in this part it is sufficient that you process only the left or right channel and output the processed signal to both the left and right speaker.

b) Here we consider the filter $H_2(z^D)$ from part A.

(i) Explain why this filter can be used to generate multiple echoes.

(ii) Again let $\alpha = 0.75$ and find D such the echo comes 220 ms after the direct sound. Implement the filter in Matlab and test it on the speech signal. Compare with the echo generated in **a)**. Which signal sounds louder?

(iii) Keep the value $\alpha = 0.75$ for the filter $H_1(z^D)$ in **a)**, but adjust the value of α for $H_2(z^D)$ such that

$$\sum_{k=0}^{\infty} h_1^2[k] = \sum_{k=0}^{\infty} h_2^2[k]$$

where $h_1[k]$ are the impulse response coefficients of the filter $H_1(z^D)$ and $h_2[k]$ are the impulse response coefficients of the filter $H_2(z^D)$. How do the loudness of the echo signals compare now? What is the reason for this?

(iv) Let $\alpha = 1.05$ and implement $H_2(z^D)$. What happens? Explain why?

(v) Implement the filter in (iii) on the DSP board and compare with the echo generated in **a)** (iv).

c) The filters above do not generate natural sounding reverberations. There are two main reasons for this. The magnitude responses, which you have computed and plotted in part A, vary with frequency. This causes a 'coloration' of the sound which is often perceived as unpleasant. The other reason is that the echo density is much lower that in a real room.

(i) Explain why the filter $H_3(z^D)$ in question 4 in part A can be used as an echo filter, and why it gives less 'coloration' of the sound than the two filters above.

(ii) Implement and test the filter with $\alpha = 0.75$ and a value of D such the first echo comes 220 ms after the direct sound. How does this echo generator sound compared to the ones in **a)** and **b)**?

(iii) Implement the filter on the DSP board and compare with your DSP implementations from **a)** and **b)**. **When you have completed this part, show and demonstrate your DSP implementations of all the echo filters to you workshop demonstrator.**

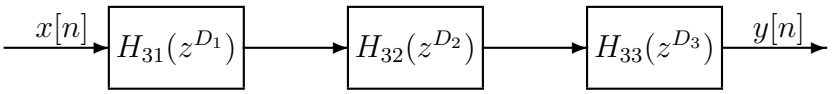

Figure 1: Three stage reverberator

Reverberator 1		
	D	α
H_{31}	50	0.7
H_{32}	40	0.665
H_{33}	32	0.63175

Reverberator 2		
	D	α
H_{31}	50	0.7
H_{32}	17	0.77
H_{33}	6	0.847

Table 1: Reverberator parameters

d) The signal generated in **c)** does not sound natural either. The objective of a reverberator is usually to get a smooth sounding reverberation, free of repetitive echoes. The difference between echo and reverberation is that with pure echos there are clear repetitions of the signal being echoed, but with reverberations there are no well-defined repetitions, but instead a diffuse sound results. In a room the reverberations are caused by reflections in surfaces generating echoes whose speed of repetition is so fast that they can not be perceived as being separate from each other.

To achieve the densely packed echoes desired in a reverberator one can cascade three to five filters of the type $H_3(z^D)$ with different values of α and D. A three stage reverberator is shown in Figure 1. The parameters of two reverberators are given in Table 1.

(i) Use Matlab to compute and plot the impulse response of the two reverberators in Table 1. Which one looks like the better reverberator? Remember that the objective is to get a smooth sounding reverberation, free of repetitive echoes.

(ii) If the delays D_1, D_2 and D_3 in a reverberator are prime numbers, the impulse response of the reverberator is more "dense". Explain why.

(iii) Implement reverberator 1 from Table 1 in Matlab, but let $D_i,\ i = 1, 2, 3$ in the table be the delays in milliseconds. Round the delays in samples of to the nearest prime number. How does it compare soundwise with previous approaches. (Hint: The Matlab command `primes` is useful.)

C Notch filter design

Bring along an audio source (e.g. smart phone) for this part of the workshop. In this part of the workshop you are going to filter out a sinusoidal disturbance from a noise corrupted audio signal in real time using the BF533 DSP.

The hardware setup is the same as for part B, and it is described in Appendix 1. The software setup is also very similar. In particular the communication between the AD1836 codec and the processor is the same. Note, however, that interrupts are sent for every second received sample, so the sampling rate is now 24kHz.

Below are descriptions of how the files in your project perform these operations:

- **main.c**: This file contains the main program, which is run after initialisation of the DSP. The file will set up the necessary hardware (i.e. the AD1386 codec) to receive and transmit data from the relevant ports. The function that programs the filter coefficient values (`FilterCoeff()`) will also be called. After initialisation, the `main()` function goes into an infinite waiting loop.
- **Initialize.c**: This is called by the `main()` function during initialisation, and is used to set all the relevant *hardware* parameters to their correct values.
- **ISR.c**: When the receive buffer is full, that is, when the next audio sample is available, an interrupt service routine is called which will in turn call the function `Process_data()` to perform the relevant signal processing.
- **Process_data.c**: Contains the `Process_data()` routine, called every time an audio sample is received. This function is responsible for processing the input audio signal and outputting data to the speakers. The functions `AddSinus()` and `NotchFilter()` (explained later), are called from here to modify and filter the input audio signals.
- **_LabTasks.c**: *This is the only file you need to modify.* The functions `FilterCoeff()`, `AddSinus()` and `NotchFilter()`, explained shortly, are all found here.
- **SPWS2-notch.h**: The header file containing prototypes and macros and global variable definitions.

Your task is to write **three** c functions in `_LabTasks.c`:

- `FilterCoeff()` is used to initialise the filter coefficients. This function is called from `main()` only once upon initialisation. You must use global variables as filter coefficients.

- `AddSinus()` will be used to mix a sinusoid signal to the input audio signal. It is called from `Process_data()` every time a new music sample is received by the DSP - that is, every sampling period. The global variables `LeftInput` and `RightInput` contain the current sample and should be used to write to the global variables `LeftInputCorrupted` and `RightInputCorrupted`.
- `NotchFilter()` is where the notch filter is applied to filter the corrupted audio signals created by `AddSinus()`. This function is also called from `Process_data()` every sampling period. The global variables `LeftInputCorrupted` and `RightInputCorrupted` should used as inputs. The filtered output at each sampling interval should be stored in the global variables `LeftOutputFiltered` and `RightOutputFiltered`.

The skeleton of the three functions are given in Appendix 4. See also Appendix 2 for coding hints.

The software is set up so that the speaker output is changed every 5 seconds between `Left/RightInput`, `Left/RightInputCorrupted` and `Left/ RightOutputFiltered`. You should not change this setup as it will help your lab demonstrator evaluate your code.

C.1 Workshop tasks

Show your workings on Question 5 in part A to the demonstrator before starting on this part.

WARNING: Never put on a headset or have the loudspeakers turned up at high volumes before you have verified that the code is working as expected. If the notch filter is unstable or the amplitude of the sinusoid is too large, very loud and unpleasant audio output is generated.

a) Start up CCES on the PC. Open the project file SPWS2-notch which can be downloaded from LMS. The file `_LabTasks.c` contains the skeleton code to be modified. Connect an audio source to the DSP board.

In order to test the lab setup, build the project without modifying the functions on the skeleton file, and run the project on the board. The output to the speakers should be the noise free music signal.

b) The array `MusicSignal` contains the first 100 samples of the left audio signal. Check the amplitude of the audio signal and modify `AddSinus()` to add an audible sinusoidal disturbance at (continuous) frequency F Hz. The workshop demonstrator will tell you which value of F you are going to use. Verify that the sinusoid is audible. You should hear the original signal for 5 seconds, followed by the corrupted signal for 10 seconds.

c) Let the 3-dB bandwidth of your digital notch filter be 0.1π. Program the computations of the filter coefficients in `FilterCoeff` and implement the filter with the computed filter coefficients in `NotchFilter`. Verify that the filter is working as it should (that is, that the sinusoid is audibly removed). The audio output should alternate between 5 seconds of the original (clean) signal, 5 seconds of the corrupted signal, and 5 seconds of the filtered signal. You should be able to *hear* a clear difference.

d) Design two more notch filters with smaller bandwidths, e.g. 0.01π and 0.0025π, and compare with the results from **c)**. **Demonstrate your program and show the code to your workshop demonstrator after you have completed this task.**

1 Hardware and software setup

In this workshop you are going to process audio signals in real time using Analog Devices Blackfin 533 Digital Signal Processor which is a 16 bit fix point processor (further details on http://www.analog.com/processors/processors/blackfin/). You will be using the CCES development environment (see the notes for the first workshop for a description of this development environment) and the ADSP-BF533 EZ-Kit Lite board (hereafter called the board or the DSP board).

1.1 Hardware Setup

An overall system architecture can be seen in Figure 2. The ADSP-BF533 evaluation board has many features, but for this workshop you will be using only a subset of the components. The **AD1836** codec is responsible for acquiring the data (sampled at 48kHz) from the **Stereo IN Phono Jack** and passing it to the ADSP-BF533 processor via the **SPORT0** buffer. After adjustments are made to the signal (it is up to you to program these adjustments) the data is passed back to the AD1836 codec and output onto the Stereo OUT Phono Jack, connected to some speakers.

Locate the processor, the codec, and the input/output jacks on your board. Before you start with the workshop tasks, make sure that the cabling is correct. You should have power on the evaluation board, a USB connection to the computer, sound going into AUDIO IN and speakers connected to AUDIO OUT. See Figure 3 for a picture of the setup. If you are unsure of any of these, see your workshop demonstrator.

WARNING:
DO NOT OVERLY HANDLE THE HARDWARE
Electrostatic discharges can cause damage

1.2 Software description

The program written for the implementation of the echo filters works as follows: communication between the processor and the AD1836 codec, which is responsible for sampling the audio, is set up. Samples received from the AD1836 are moved into the DSP's receive buffer, using DMA. The samples are then available for processing by the ADSP-BF533 processor, which is what you are responsible for doing. The processed data is then placed in the transmit buffer, and is passed back to the AD1836 which will be responsible for outputting the information to the speaker ports.

Below are descriptions of how the files in your project perform these operations:

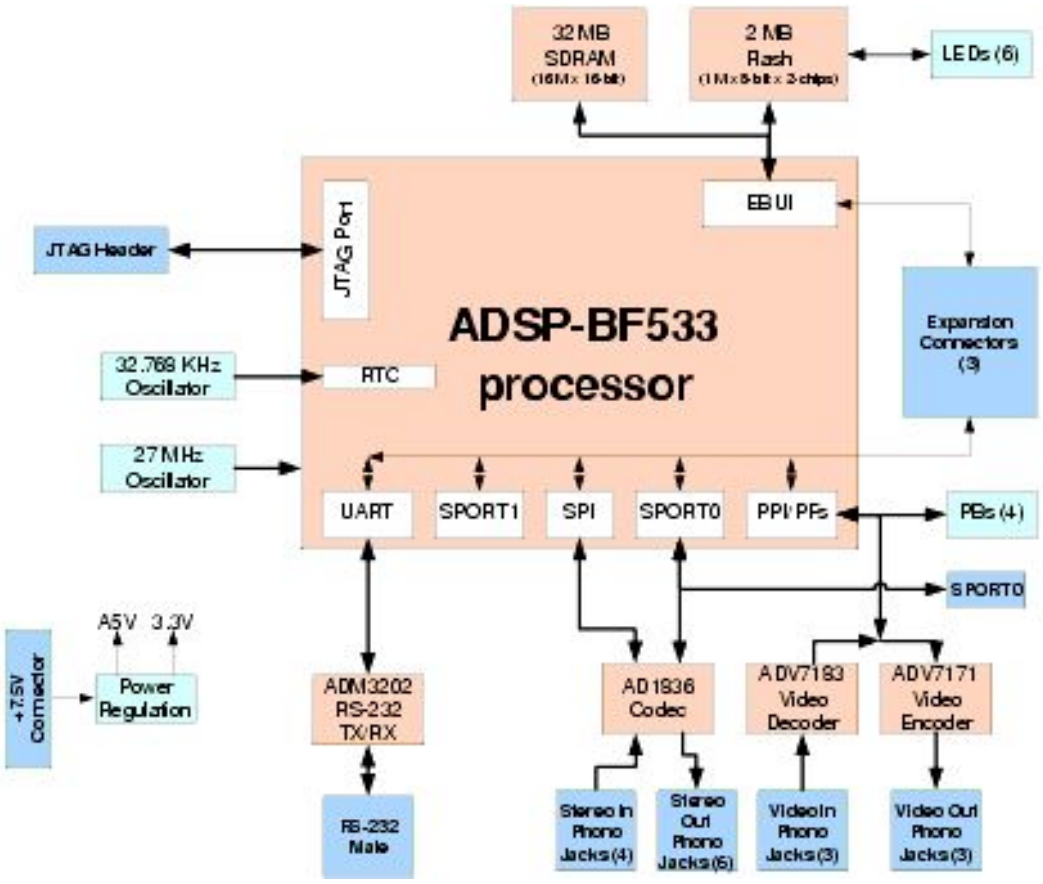

Figure 2: System Architecture (taken from the ADSP-BF533 user manual)

- **main.c**: This file contains the main program, which is run after initialisation of the DSP. The file will set up the necessary hardware (ie. the AD1836 codec) to receive and transmit data from the relevant ports. After initialisation, the `main()` function goes into an infinite waiting loop.

- **Initialize.c**: This is called by the `main()` function during initialisation, and is used to set all the relevant *hardware* parameters to their correct values.

- **ISR.c**: When the receive buffer is full, an interrupt service routine is called which will in turn call the function `Process_data()` to perform the relevant signal processing. Note that for the implementation of the echo filters the size of the receive buffer has been increased such that even though the codec operates at 48 kHz, interrupts are only generated for every 6th sample received, effectively reducing the sampling rate to 8kHz.

- **Process_data.c**: Contains the `Process_data()` routine, called every time an audio sample is received (or more precisely when the input buffer is full). This function calls the routine `EchoFilter` and outputs the data to the speakers.

- **_Labtasks.c**: Contains the routine `EchoFilter` which implements the various

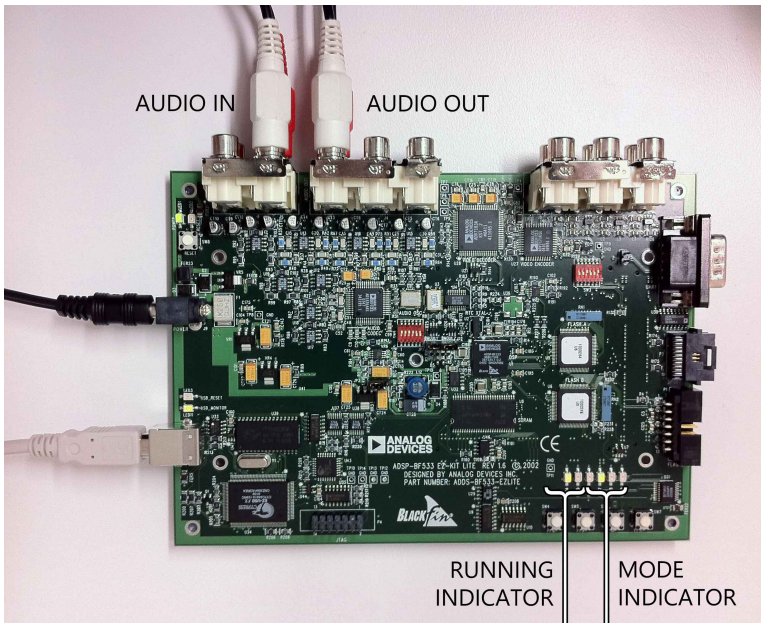

Figure 3: Hardware setup

echo filters. *This is the only file you need to modify in order to program the echo filters.*

- **SPWS2-echo.h**: The header file containing prototypes and macros AND global variable definitions.

The code of `Process_data.c` and `_Labtasks.c` is given in Appendix 3.

The software is set up so that the speaker output is changed every 3 seconds between the original audio signal, and audio signals with echoes added. For testing and demonstration purposes you can change these settings. E.g. if you only want to test the filter implemented in question **b)** in part B, you can change

```
// Play 3 seconds of signal with echo a)
else if (temp <= 12){
    iChannel0LeftOut = (int) (loa);
    iChannel0RightOut = (int) (loa);
}
```

to

```
// Play 10 seconds of signal with echo b)
else if (temp <= 26){
    iChannel0LeftOut = (int) (lob);
```

```
    iChannel0RightOut = (int) (lob);
}
```

You will now hear a repeated pattern of 3 seconds of the original signal, half a second of silence, and 10 seconds of the signal filtered through the echo filter $H_2(z^D)$.

2 Coding Hints

For those of you who are less familiar with c, here are some coding hints that may help you throughout the laboratory:

- Global variables can be modified from anywhere in your project, and so you should be careful not to give your local variables identical names. Most variables that you will need have been defined as global rather than as function inputs and outputs for simplicity. For the notch filter implementation most of the global variables you will need to have access to for this lab have been defined in `_LabTasks.c`.
- Variables defined as `float` may lose resolution if multiplied or divided by a variable of type `int`. Any such multiplications should be typecast as floating point by typing `(float)` immediately before it.
- `Static` local variables are initialised only once when a function is first called, and retain their value upon subsequent calls to the same function. For this reason, they are useful for "remembering" the state of your program between events, without the need for using global variables (the advantage being that these can only be accessed within the local host function, and so will not be confused with other static variables of the same name in other functions). They are useful for interrupt driven applications such as these ones.
- In c an array with `N` elements is indexed from `0` to `N-1`.
- All elements of an array can be initialised at the time of declaration.

  `static float x[100]={0.0};`

  initializes all elements of `x` to 0.0.
- In real time applications the signals are obtained samples by samples. In filtering applications a number of past values of the signals must be remembered and stored away. Assume that `N` values of a signal must be stored in an array `x`. There are two common ways of storing a signal in an array.
  - The most recent sample (current value) is stored in `x[0]`, the previous value in `x[1]`, the one before that in `x[2]` and so on. When a new sample arrives, the old sample values are updated

```
x[N-1]=x[N-2];
x[N-2]=x[N-3];
```

until

`x[1]=x[0];`

Note that the order of the update is important. This update is easily carried out in a `for` loop. The most recent sample is then place in `x[0]`, e.g.

`x[0]=newsample;`

– The drawback of the approach above is that all variables have to be updated when a new sample arrive. This can be time consuming if a large number of variables need to be stored. An alternative is to store the incoming values in a circular buffer. Let `current` be a counter keeping track of where the next value should be stored. Then when a new samples arrive

```
x[current]=newsample;
current++;
current=current%N;
```

Here current counts from 0 up to N-1 and starts at 0 again. % is the modulus operator in c which gives the remainder after integer division.

3 Files for echo generation

3.1 Process_data.c

```
#include "SPWS2-echo.h"

void Process_Data(void)
{
    static int i = 0;

    LeftInput = (float)iChannel0LeftIn;
    RightInput = (float)iChannel0RightIn;

    // run echo filters
    EchoFilter();

    int time;
    time = (i / 4000) % 28;
```

```
ledreset();

// Play 3 seconds of the original signal
if (time <= 5) {
    iChannel0LeftOut = iChannel0LeftIn;
    iChannel0RightOut = iChannel0RightIn;
    led(0, 1);
}
// Play half a second of silence
else if (time <= 6) {
    iChannel0LeftOut = 0;
    iChannel0RightOut = 0;
}
// Play 3 seconds of signal with echo a)
else if (time <= 12) {
    iChannel0LeftOut = (int)loa;
    iChannel0RightOut = (int)loa;
    led(1, 1);
}
// Play half a second of silence
else if (time <= 13) {
    iChannel0LeftOut = 0;
    iChannel0RightOut = 0;
}
// Play 3 seconds of signal with echo b)
else if (time <= 19) {
    iChannel0LeftOut = (int)lob;
    iChannel0RightOut = (int)lob;
    led(2, 1);
}
// Play half a second of silence
else if (time <= 20) {
    iChannel0LeftOut = 0;
    iChannel0RightOut = 0;
}
// Play 3 seconds of signal with echo c)
else if (time <= 26) {
    iChannel0LeftOut = (int)loc;
    iChannel0RightOut = (int)loc;
    led(3, 1);
} // Play 1 seconds of silence
else if (time <= 27) {
    iChannel0LeftOut = 0;
    iChannel0RightOut = 0;
```

```
    }
    else {
        i = 0;
    }

    i++;
}
```

3.2 _LabTasks.c

```
#include "SPWS2-echo.h"

// Input samples
float LeftInput;
float RightInput;

// Output samples
float loa, lob, loc;

// Declare any global variables you need

void EchoFilter(void)
{
    // TODO: Implement echo filter (a)
    loa = LeftInput;

    // TODO: Implement echo filter (b)
    lob = LeftInput;

    // TODO: Implement echo filter (c)
    loc = LeftInput;
}
```

4 Files for notch filter

4.1 _LabTasks.c

```
#include "SPWS2-notch.h"

#define SAMPLE_RATE 24000.0
#define PI 3.14159265359

// Input samples
float LeftInput;
float RightInput;

// Corrupted samples
float LeftInputCorrupted;
float RightInputCorrupted;

// Filtered samples
float LeftOutputFiltered;
float RightOutputFiltered;

// TODO: 0. Define your own global coefficients for filtering

void FilterCoeff(void)
{
    // TODO: 1. Initialise the filter coefficients
    //   You should write this function so the filter centre frequency and
    //   bandwidth can be easily changed.

}

void AddSinus(void)
{
    // TODO: 2. Add the sinusoidal disturbance to the input samples

    LeftInputCorrupted = LeftInput;
    RightInputCorrupted = RightInput;
}

void NotchFilter(void)
{
    // TODO: 3. Filter the corrupted samples

    LeftOutputFiltered = LeftInputCorrupted;
```

```
    RightOutputFiltered = RightInputCorrupted;
}
```