EC581 DSP Projects: Lab Project #1

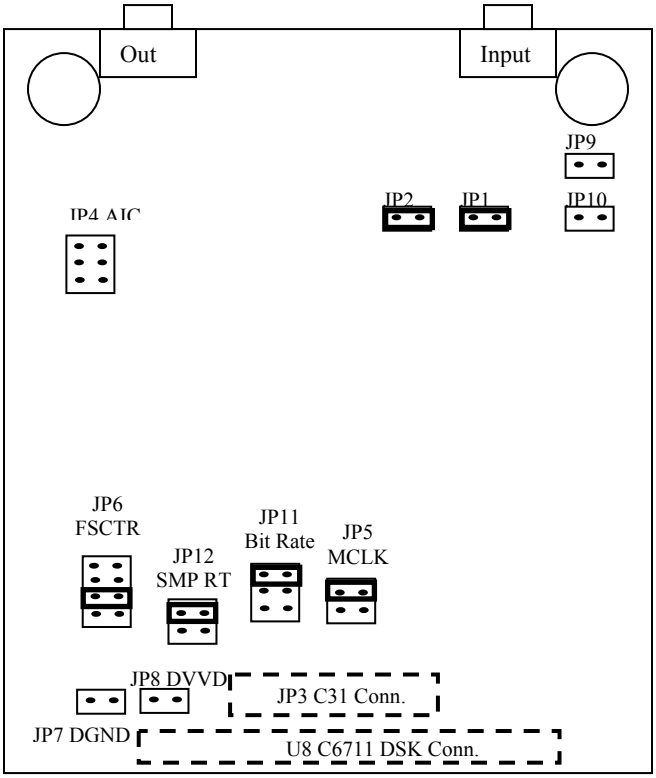
Dept. of Electrical and Computer Engineering
Rose-Hulman Institute of Technology

TMS320C67x C Familiarization and Audio Sampling

Last Modified on 17– Mar-2004 (may)
Original by KEH

I. Introduction

This lab follows right after the TI lab9. Now that you have a working input and output it's time to play. First copy all the TI lab9 files into a new lab01 directory. Rename everything to lab01 and compile and run. It should work.



No need for a microphone, the built-in microphones should work. However you may need to change some jumpers. The figure to the left shows the default position of the jumps. Make sure all but jumpers JP11 and JP12 are in the positions shown. Be sure JP11 and JP12 are one below the positions shown. That is one position closer to the DSK connector. Is puts JP12 in the bottom position and JP11 in the middle position.

Lab Report Part 1a

Devise a test to measure the frequency response of your system. The sampling rate should be 48 kHz. Verify it. The frame-based processing we are doing introduces a delay in the system. Estimate the delay length based on your code. Verify this number by making measurements.

Lab Report Part 1b

Try modifying the interrupt service routine (ISR), ProcessBuffer, to play the left channel only. Next modify the ISR to play both channels, but with left and right channels interchanged. Include your modified ProcessBuffer listings (just ProcessBuffer() only, to save paper!) in your lab report as **Listing L1.1a.**

Lab Report Part 2b.

Also investigate how many bits are really needed to represent intelligible speech. Do this by masking out lower-order bits of the right channel, as well as all the bits of the left channel, and speak "secret words" very quietly in the microphone (cupping it in your hand). Find out how well your lab partner can understand the secret words that were spoken. Remember, don't let your partner know what words you are speaking ahead of time, as "hindsight is 20/20", especially in the case of aural word recognition! You may be surprised to find that audio samples consisting of only 4 or maybe even 2 bits permit

reasonably good intelligibility. However, more bits will be needed if the speaker to be not only understood, but also recognized. Note how the quantization noise becomes more noticeable, as you go to a smaller number of bits per sample.

Important Note: For this study to be meaningful, it is important that you operate the A/D converter over its entire dynamic range, so that even the most significant bit positions vary as you speak. Otherwise, when you mask out all but the uppermost few bits, these remaining bits will be either all zeros or all ones, depending the current polarity of your audio waveform. Therefore, I suggest that you begin your study (with all 16 bits present) by adjusting the gain of your mixer box high enough to cause the audio output to nearly saturate (distort) on voice peaks. You should continue to speak at this same level as you begin your study, masking out lower-bits. You will want to do the masking on ints, not floats, so be sure to work on the samples before they are turned to floats.

Tabulate your test results for 16 bits, 8 bits, 6 bits, 4 bits, 2 bits, and 1 bit of quantization, using five different test words for each quantization level (you will need 30 similar-length test words in all). Indicate what percentage of the five spoken words were recognized correctly at each level of quantization. *This table must appear in your lab report as* ***Table L1.1b.*** Also, include the modified ISR listing that provides 4-bit quantization as **Listing L1.1b**.

II. Audio Reverberation System

Your next task is to modify ProcessBuffer to emulate the analog audio reverberation system of Fig. 1. To convince yourself that this is the proper block diagram of an analog reverberation system, imagine that you clap your hands near the microphone. This clap (impulse) will immediately be heard in the loudspeaker. Then 0.5 second later, you will hear the first echo (a clap that is 0.75 as strong as the original clap). Then 0.5 second after that, you will hear a clap that is 0.75 as strong as the one before it, etc.

Start by creating a project. Copy the necessary files from the Lab1b subdirectory. Create a new project named **lab1c.**.

Helpful Hints: Mainly the ISR needs to be changed, however you will also have to define some global variables. Use a globally defined (defined outside of the *main()* function) one-dimensional integer array called **gDelay[ ]** to create a digital "circular delay buffer" so it can be accessed by the interrupt routine. You are sampling at 48000 samples per second, so you will need a buffer length of N = 0.5 / (1/48000) = 24000. Memory size may be an issue here. If so, check ask for the **Advanced Memory Management** lecture on how to move this buffer to SDRAM.

You should use a global index variable, **index**, which is initialized to in the main program. It is a good idea to initialize all of the elements in the **gDelay[ ]** array to zero to avoid a sudden burst of random noise when the program is first started. ZeroBuffers() is a good place to put it. In the interrupt routine, the output of the delay line is taken out of the position "index" in the array:

delay_line_output = gDelay[index];

However you will have BUFFSIZE samples to remove, so you'll have to use a loop. Then the calculated delay line input value (which, according to Figure 1, is the present input sample + 0.75 * Delay_line_output) must be placed into this position in the array **gDelay[index]**,

gDelay[index] = present_input_sample + delay_line_output*3/4;

At the end of the ISR, the index variable must be incremented by **BUFFSIZE** so that it points to the next oldest value in the **gDelay[ ]** buffer to ready it for the next sample interrupt. If the end of the array (**index = N**) has been exceeded, the index value must be wrapped back to 0.

Lab Report Part 2.
Modify your program to implement an audio reverberation unit with (a) 0.1 second delay and a 0.5 decay rate, and (b) a 0.1 second delay and a 0.25 decay rate. You will be asked to demonstrate your reverberation program to the lab instructor before the start of next week's lab period for all three of these settings. Be sure to include the commented listing of your reverberation program as **Listing L1.3** (which will be signed by the instructor upon successful demonstration) in your lab report.

Figure 1. Analog Reverberation System to be Emulated Digitally

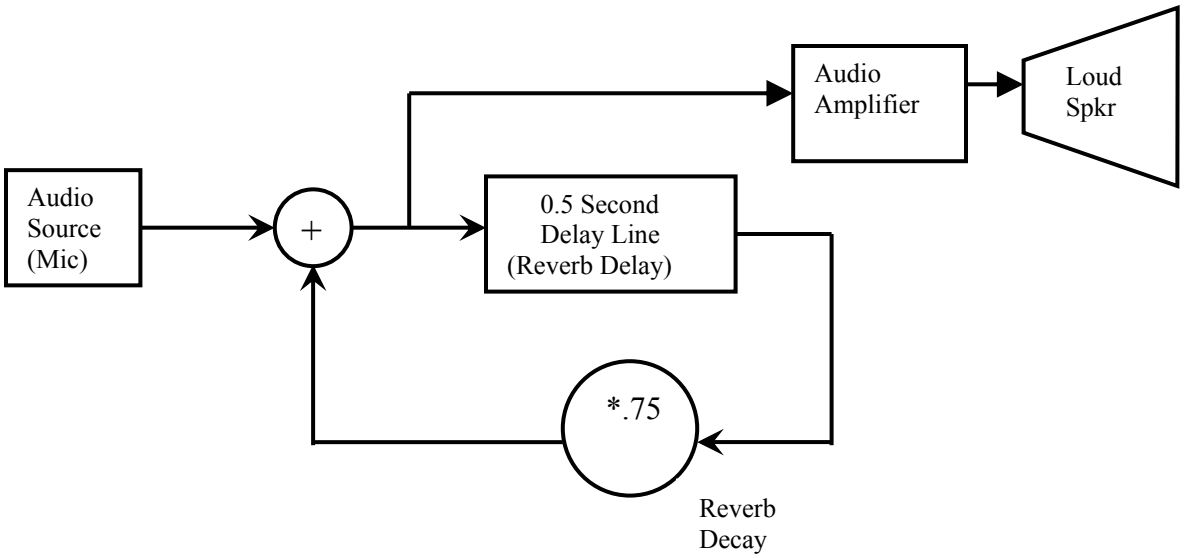
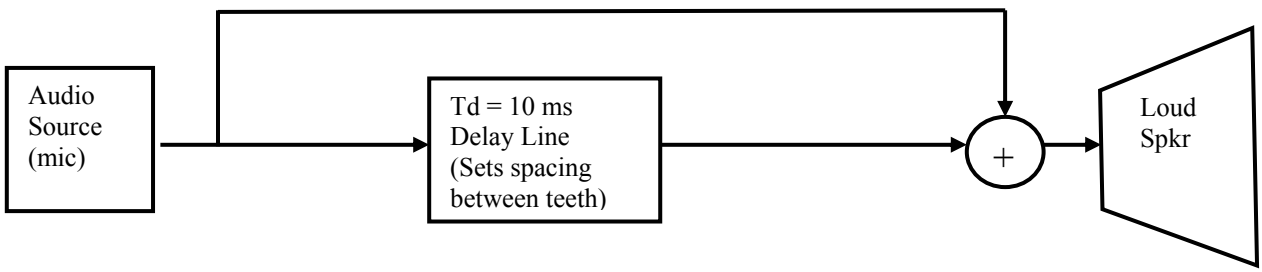


Figure 2. Analog Audio Comb Filter to be Emulated Digitally



III. Audio Comb Filter

Finally we shall digitally emulate a different kind of analog system called a "comb filter", which is shown in Fig. 2. Recall that the reverberation system implemented in the previous section was a "feedback" system, with an infinitely long, though exponentially decaying, impulse response. In this system, no feedback paths are present, resulting in a system with a finite impulse response.

Note that the sinusoidal steady state magnitude transfer function for this analog system can easily be found using the Fourier transform "Delay Theorem":

$$\left|\frac{Y(f)}{X(f)}\right| = \left|1 + e^{-j2\pi f T_d}\right| = \left|1 + \cos(2\pi f T_d) - j\sin(2\pi f T_d)\right|$$

$$= \sqrt{(1 + \cos(2\pi f T_d)^2 + \sin(2\pi f T_d)^2}$$

$$= \sqrt{2 + 2\cos(2\pi f T_d)} = 2\left|\cos(\pi f T_d)\right|$$

(Note, the last step used the trig identity: cos(2a) = 2cos(a)^2 - 1)

The above result for the system gain passes through minimum values of 0 (corresponding to destructive interference at the summing junction) when the argument of the cosine function equals **π**, **3π**, **5π**, **7π** etc. Thus, gain nulls occur at frequencies given by

2π*fnull*Td = π, 3π, 5π, 7π,... => fnull = 1/2Td, 3/2Td, 5/2Td, 7/2Td,...

Likewise, the above result for the system gain passes through peak values of 2 (corresponding to constructive interference at the summing junction) when the argument of the cosine function equals **0**, **2π**, **4π**, **6π**, etc. Thus, gain peaks occur at frequencies given by

2*pi*fpeak*Td = 0, 2π, 4π, 6π,... => fpeak = 0, 1/Td, 2/Td, 3/Td,...

Thus for Td = 0.01 seconds, the gain nulls should occur at 50 Hz, 150 Hz, 250 Hz, etc., and the gain peaks should occur at 0 Hz, 100 Hz, 200 Hz, 300 Hz, etc.

Lab Report Part 3

In a separate subdirectory **Lab1d** implement the audio comb filter project. Once again use your template. Use a sampling rate of 48 kHz. Note that the delay buffer size is now much smaller. Connect an oscilloscope to the analog output of the DSK and a function generator to the input of the DSK to verify that the peaks and nulls occur at the proper frequencies. Attach a listing of your source code (signed by the instructor as you demonstrate the reverb program) as **Listing L1.4**. Include a table in your lab report showing the predicted and the measured peaks and nulls over a frequency range of 0 – 1 kHz. Label this table **Table L1.4**.

The higher the sampling frequency, the less time the DSP chip has to do the required calculations before the next sample interrupt occurs. Use the "CPU Load Graph" to see how much of the CPU you are using. Note the percentage in your report.

Part IV. Audio Flanger

Imagine that your comb filter's delay buffer size is slowly swept (varied), so as to cause the delay line's delay time (Td) to gradually decrease from 10 ms down to 1 ms over the course of about a second. Then, over the course of another second, assume the delay is allowed to gradually increase from 1 ms back to 10 ms. Let this cycle repeat forever. This would result in a continuously varying comb filter whose "frequency response teeth" gradually separate and come together, separate and come together, etc. Therefore, any sustained musical note being played through the system would take on a "shimmering, ethereal" quality, as the teeth of the comb march past its dominant frequencies. This is the principle behind the special effect heard on many popular music recordings known as "audio flanging".

Write a flanging program that implements such a variable delay comb filter. When you play a sine wave (from a signal generator) into the system, you should see it go up and down in amplitude as the teeth of the variable comb filter pass through it. Aurally, it should "shimmer"! Also, try playing a low frequency (50 Hz) square wave, which is a "spectrally rich" signal with many harmonics spread across the audio spectrum. You should hear different harmonics emphasized at different times as your flanger processes this signal.

Include your program listing (signed by the instructor when it is successfully demonstrated) as **Listing L1.5**.

A critical problem in implementing an audio flanger involves how to initialize the extra memory, as you increase the length of the delay buffer, so as not to cause any noticeable discontinuity (pop) in the processed audio output signal. Try to determine a suitable way to minimize this "pop" as the buffer lengthens.

V. Performance

If time permits, see if you can improve the performance of your program. Bring up the "CPU Load Graph" and see how much of the CPU you are using now. What things can you do to decrease the load? Try them. Record your results in your report.

VI. Extras

If you do only what I've asked so for I'll give you maybe 7/10 for your lab grade. Now that you have a basic working setup, think of some other interesting things to do. It's time to be creative. Thinking of and doing something interesting is an easy way to get a full 10/10!