Assignment 1 – Audio Effects

# Introduction

In this assignment, you are going to implement a real-time audio effect. You are going to run code that you write in C# under the Windows operating system. Although Windows is far from being a real-time operating system, modern PC’s are so fast that they have no difficulty keeping up with the rate that audio information is generated, provided that a small buffer is used.

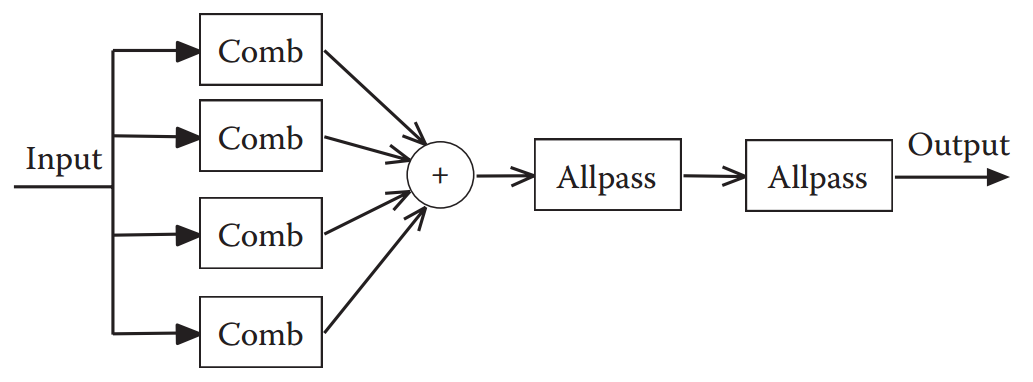
You have two alternatives for effects to implement – an audio **flanger** or a **reverb** (using Schroeder’s algorithm from 1961). The flanger is more difficult to implement, so unless you are up for a challenge, chose to implement the reverb algorithm. Bonus marks (described at the end) will be assigned for implementations of a flanger.

# Background

The theoretical underpinning for the effects will not be described in this document. Instead, it is found in .pdf files included in the Assignment 1 folder of the website. While the theory is relatively complex, the implementation is somewhat straight forward (although there are a few tricky bits).

As a quick summary, the basic implementation of Schroder’s reverb is shown below:

Reverb



In terms of equations, there are two parts to this implementation:

Comb Filter:

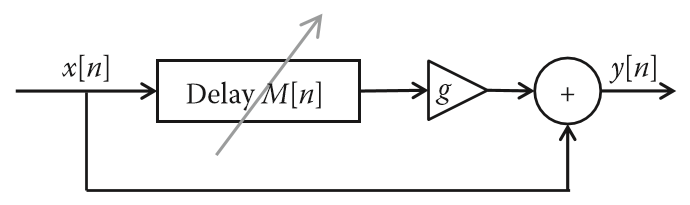
y[n] = x[n – d] + gy[n - d]

where g is a scale (gain) between 0 and 1. Details for selecting this scale factor is included in the .pdf. y[n] is the output of the filter for the nth sample. x[n] is the input to the filter for the nth sample. So x[n – d] refers to an input sample that occurred d samples in the past and y[n – d] refers to a filter output that occurred d samples in the past. This means that you will have to keep track of past samples by creating buffers for the outputs of the comb filters and the input. Note that the amount of delay for these filters is in the order of tens of milliseconds. Given that each current buffer (for 128 sample buffer) is about 3 milliseconds in length, you will need to create delay buffers that can store multiple current buffers. For an example of how to do this, have a look at my code for the delay effect, which has delay buffers for both the input and the output.

All Pass filter:

y[n] = -gx[n] + x[n - d] + gy[n – d]

The basic implementation of a flanger is shown below:



There is only one block to implement, a variable delay time. The amount of delay is changed in real-time using a low frequency oscillator (LFO). Typically, this ramps the length of the delay up and down over a second or two.

In terms of an equation, we have:

y[n] = x[n] + gx[n – M[n]]

where M[n] is a variable delay amount that depends on n. Note that this can be (and should be – see below) a non-integer value to give a smooth sounding effect.

# Implementation

In general, you will need a buffer for the output, a buffer for the input, and delay buffers for both the input and the output (for the reverb) and just the input for the flanger. The code supplies you with the input and output buffers when the function AsioDriver\_BufferUpdate but you have to create the delay buffers yourself. These are simply large arrays that are updated each time the function is called. Typically, you will need delay buffers large enough to handle multiple calls to the function, so you will create a continuous record of input or output samples over multiple calls.

For all delay buffers – for both the flanger and the reverb, the buffers should be implemented as what is called a circular array (so that the index at the end of the buffer wraps around to the beginning). This is most commonly done using the modulo (%) operator. That way, once the end of the buffer is reached, the next sample goes into the beginning of the buffer and the indexing is continuous. For more details, do a quick search on a “circular buffer”.

For the flanger, an LFO that changes the value of the delay is quite easy to implement. All you have to do is increment the value of the delay that you are using each time you update the buffer. When the delay value reaches the maximum that you wish to allow, simply begin decrementing the delay. In the case of a flanger, a maximum delay of about 300 samples (roughly 100 milliseconds) works quite well. If you increment once per buffer update (every 3 milliseconds for 128 sample buffer), you will achieve maximum delay after 3 mS \* 300 or 900 mS. Thus, a full sweep up and down would take 1800 mS or 1.8 seconds.

For the flanger, the hardest thing to implement is the fractional sample delay. The problem is that the delay time only changes once per buffer, and you want a continuous change in delay time OVER the length of the buffer. Otherwise, the change in delay time will become very noticeable. So the way to think about it is that you have one additional delay step at the END of the buffer relative to the beginning (since you increment the delay at the end of the buffer (or the beginning of the next)). Therefore, we interpolate between samples from the delay buffer to determine the actual output value at each sample. For the sample at the beginning of the output buffer, the resulting delayed sample is selected as the sample from the delayed buffer with the index given by the LFO. For the sample at the end of the output buffer, the resulting delayed sample is selected as the sample from the delayed buffer with the index given by the LFO - buffer length - 1. For the sample halfway in the output buffer, the resulting delayed sample is selected as the interpolated value halfway between the sample with index LFO - buffer length / 2 and LFO - buffer length / 2 - 1. The interpolation position varies linearly from 0 at the beginning of the buffer to 1 (the next sample) at the end of the buffer.

The sample code for the delay effect should be sufficient to get you started in creating delay buffers, because it uses both an input and output delay buffer. Note that for clarity in my implementation, I did NOT use a circular buffer – I have an if statement that explicitly checks for a wrap of the buffer index – but it is more common to simply use the circular buffer.

# Assessment

Overall, each assignment is worth 5% of your final mark. They will be graded out of 5 as follows:

Working code with a sample output as a wave file (the creation and recording of the file is already in the code – you just have to change the filename). /4

A BRIEF (less than 1 page) description of how your implementation works. /1

Simply submit the resulting visual studio project (in its entirety, including the ASIO framework code so that I can run it directly from your submission) and your less than 1 page writeup in pdf or Word format. The project can be worked on in groups of no more than two. As long as both names are on the writeup, you can both submit to the same dropbox location – i.e. ONE submission per group.

The due date for the assignment will be date of the midterm exam (Feb. 27).