Implementation of sound effects in DSP

Alfredo Ricci Vásquez - Juan Carlos Bucheli García

1 Introduction

Sound is one of the physical phenomena that has intrigued human race during all history. Since ancients civilizations humans have explored different ways to manipulate sound in order to generate new sounds. Today, thanks to the digitalization of the world, humans have open an immense sea of possibilities to manipulate sound. In this new digital world, devices such as Digital Signal Processors (DSP), help humans to modify any sound signal, in a relatively easy way. For this reason, the aim of the project is to implement some algorithms to create sound effects on a DSP platform. To accomplish this goal the project is divided in two big parts. The first one is a rough implementation of the effects in Matlab, where it is possible to, as a first approach, try how they would be described over a high level language where different real time considerations can be neglected. The second part, is the implementation of the effects on C, so they can run on the DSP platform. The sound effects implemented were: *Delay, acho, amplitude modulation* and *flanger.*

2 Sound effects, and implementation on Matlab

In order to accomplish the final goal of creating the sound effects on the DSP, it is necessary to begin with some basic implementations of the effects on *Matlab.* There are three importante concepts to understand before starting to implement the effects, these are: *Long buffer*, *Sample rate*, and *Block processing.* Most sounds are continuos in time, but in order to make a digital process to this sound, the sound wave has to be discretized in x samples per second. This amount of samples per second of the digital signal is called *sample rate.* Depending on the sample rate is possible to achieve a higher or lower quality of sound, but a higher quality implies higher storage requirements . The *long buffer* is a sequence of data, which stores the original signal after it is processed in some time around 500ms (the way to decide this time will be shown later). The reason to have this long buffer is that in general sound effects are the combination of the original sound with a previously recoded version of it., so using a buffer gives the possibility to reach previous information. The first way to implement a buffer is having a fixed size vector, in which the last positions corresponds to the most recent sounds, and the oldest sounds are stored in the first positions, so every time a sound block is processed all the elements in the array are moved an amount of positions corresponding to the *block length. Block processing* corresponds to a signal processing in which the signal is processed not in a sample by sample basis, but by bigger blocks of n samples, where n is the *block length.*

2.1 Effect 1: ***Delay***

The Delay is the simplest sound effect that can be implemented, and is the basis of lots of other effects. A delay consists in, delaying the original sound, then applying a specific gain into it, and finally adding this sound to the original sound. An example of how sound is affected by delay is shown below:

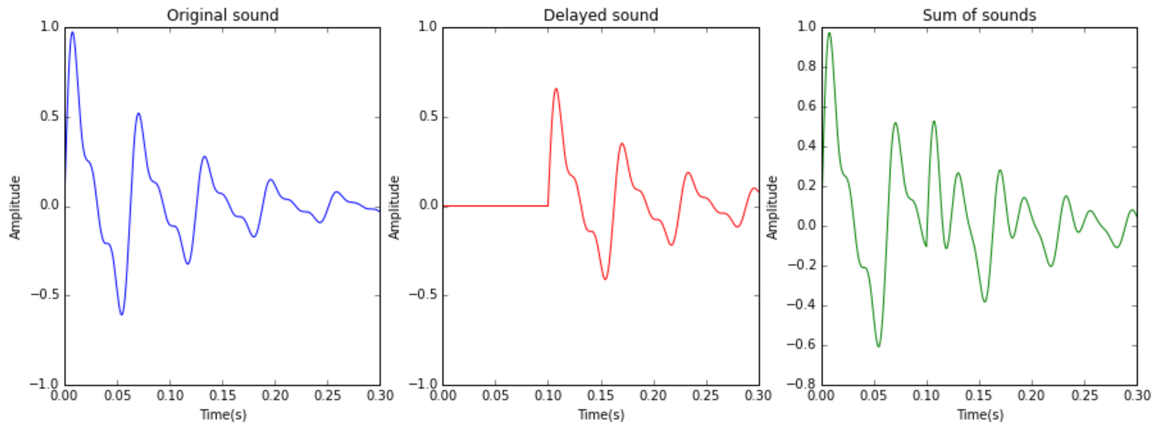

Figure 1: Delay sound

In the image is possible to see the general idea behind the delay. Te blue plot shows an arbitrary original sound, then the red sound shows the original sound, with a delay, and a gain lower than 1 and finally the green plot shows the result of adding the two previous sounds. To implement an algorithm to create this effect, is necessary to do the exact same process than the one shown above, but with the digital signal. The next block diagram shows how the effect is implemented.

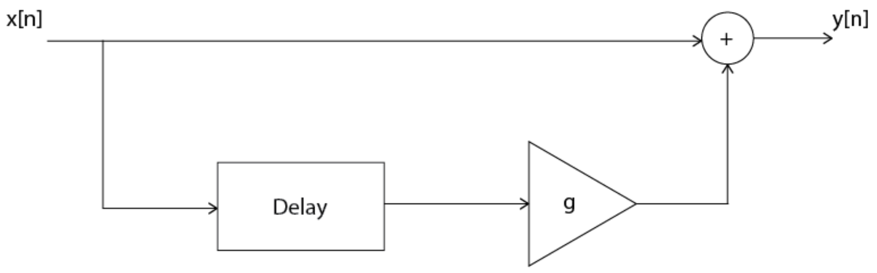

Figure 2: Delay block diagram

To implement this algorithm, it is necessary to use the long buffer created, because there it is possible to find all the previous information of the sound, so the delayed sound is stored in there. in this way, the sequence of steps to create the dalay effect is

1. To begin, it is necessary to have an input signal, which size is the *block size*.
2. All the items in the *long buffer* should be shifted to older positions, making space for the input signal to come in.
3. The input signal should replace the first (newest) positions in the *long buffer*.
4. The samples corresponding to the *delay*, are taken, multiplied by a gain factor, and finally added to the input signal. The samples corresponding to the *delay*, are the the ones located between the position given by the product of the delay (in s), with the *sample rate*, to the same product adding the *block size*. ($P_0 = D \times S_r$ and $P_f = D \times S_r + B_s$)
5. Finally the output signal is returned.

   The following diagram shows how this process works

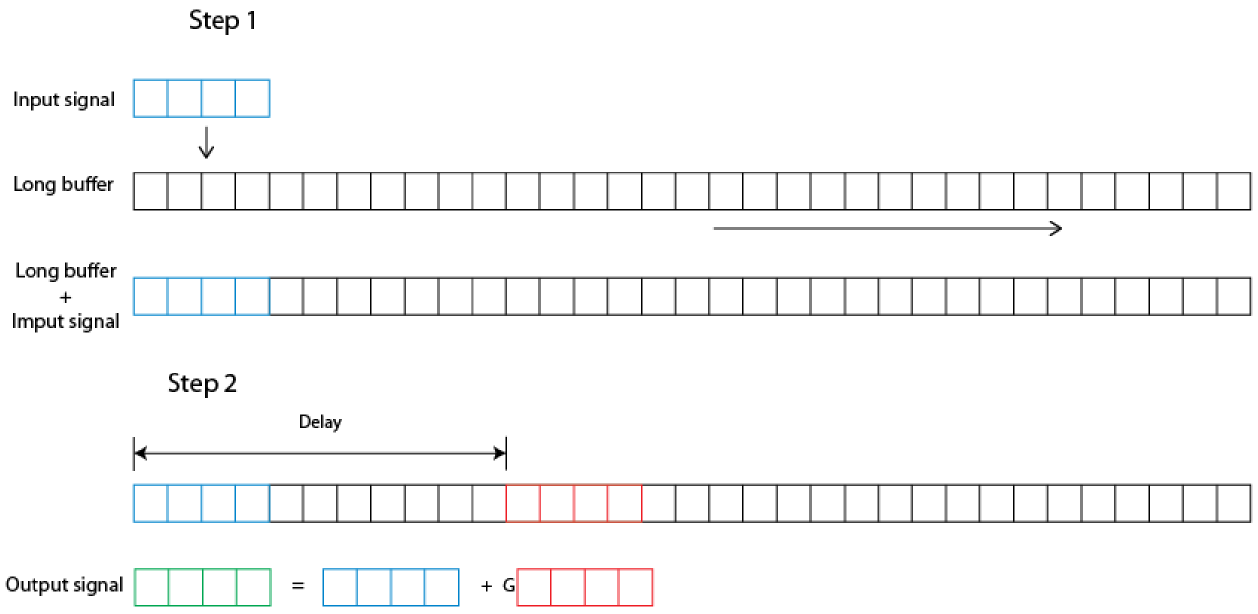

Figure 3: Delay

2.2 Effect 2: *Echo*

The implementation of the echo, has the same general structure as the implementation of the delay, but with the possibility of having more than one delay. In a general form, the echo consists of adding the original sound to N original sounds with a certain delay and gain for each one. It is possible to think of the delay as a specific case of the echo where $N = 1$. An example of the echo with $N = 3$ is shown below.

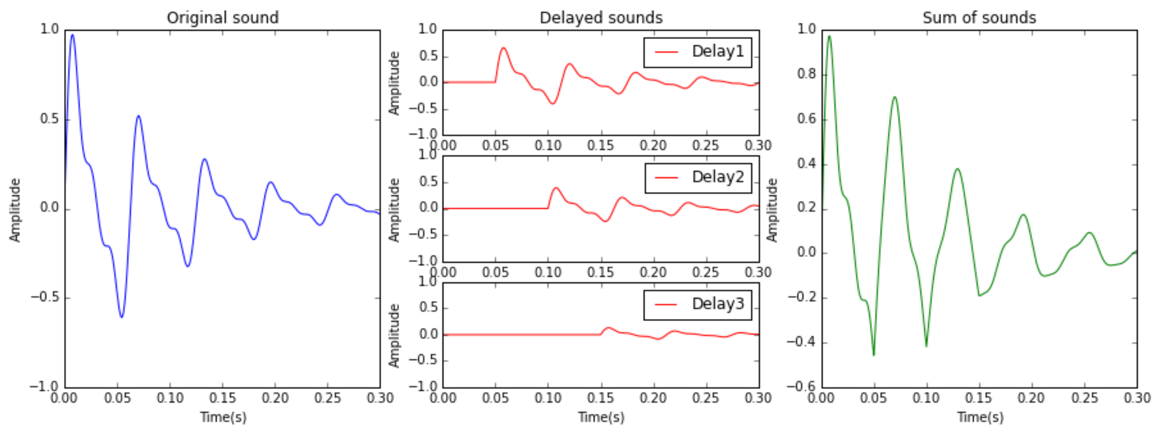

Figure 4: Echo sound

In the previous plot each of the red plots is a delay of the original with different gain (decrease as the delay is longer). The result is the green plot. In order to create an algorithm that generates this effect the signal should be edited the same way. The block diagram of the algorithm is the next.

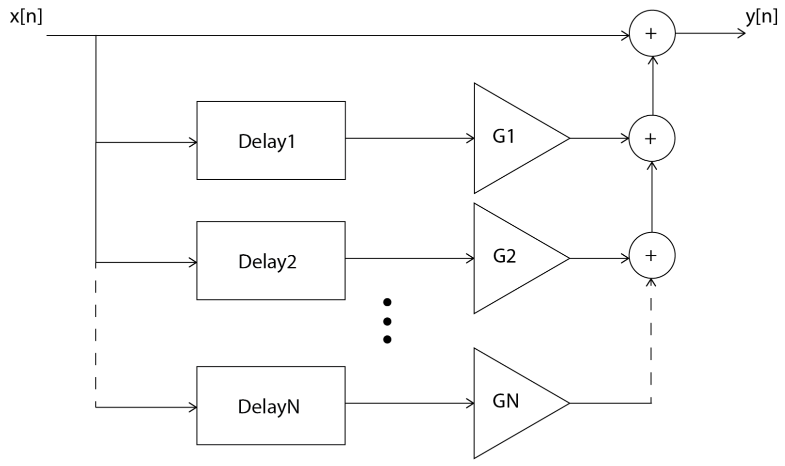

Figure 5: Echo block diagram

Following this idea, to create the echo is necessary to take the information of the sound within ranges of 50-400ms before. All this information is stored in the long buffer. The steps to create the echo are almost the same as for the *delay*, but with the difference that instead of adding to the input signal only one delayed signal with one gain factor multiplying it, multiple delayed signals with different gains are added. In order to reduce the parameters that the user is required to input, the delays and gains are function of the *main delay* (D_0) and *main gain* (G_0)(the first one), and its position in the following way:

$$D(m) = (m+1)D_0 \tag{1}$$
$$G(m) = {G_0}^{-(m+1)} \tag{2}$$

Where m is goes from 0 to N, where N is the total number of *delays*. The next image shows how to implement this idea.

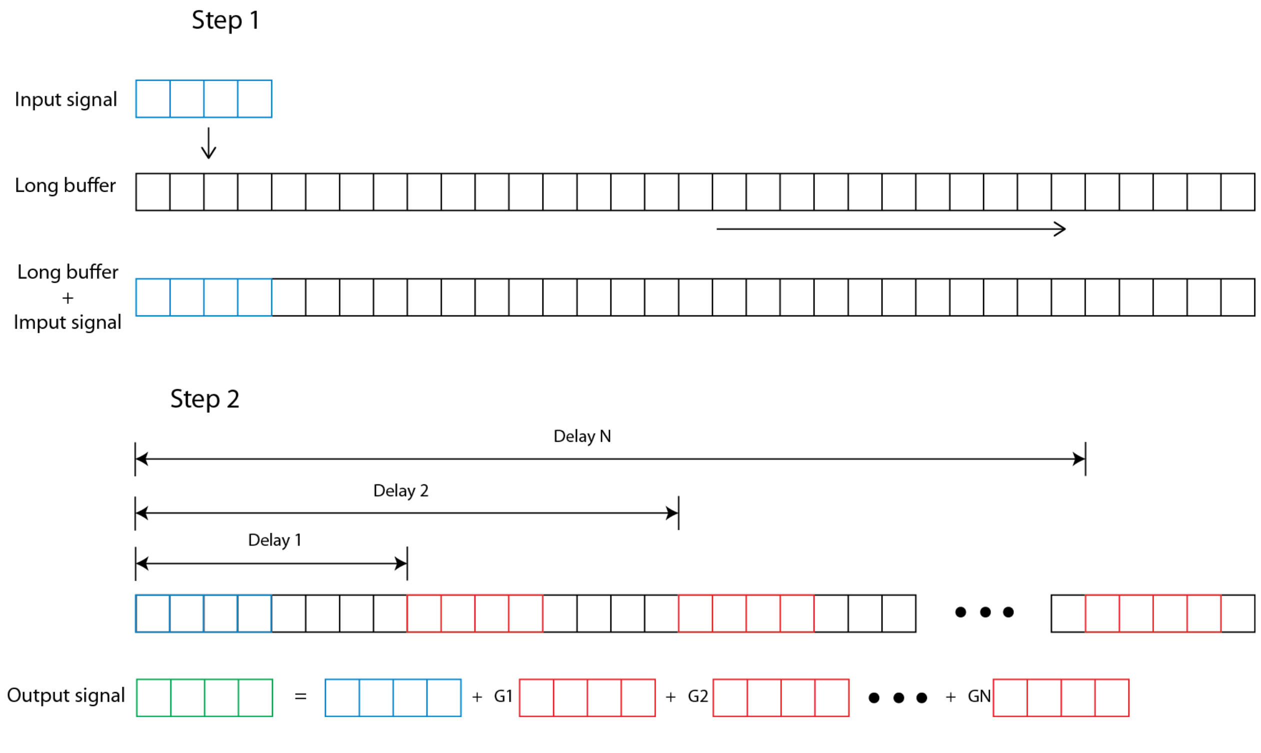

Figure 6: Echo

2.3 Effect 3: *Amplitude modulation*

After implementing the echo, the next effect to consider is the amplitude modulation. This effect consists on the same echo created before, but with the difference that the gain applied to the delayed signals varies over time in a periodic way. The gain as a function of time follows the next equation:

$$G(t) = 0.5\sin(\omega t) + 0.5 \tag{3}$$

Were *omega* is an user defined parameter that determines the frequency of the effect. The next diagram shows how the gain is a function of time, using the same original sound of the previous cases.

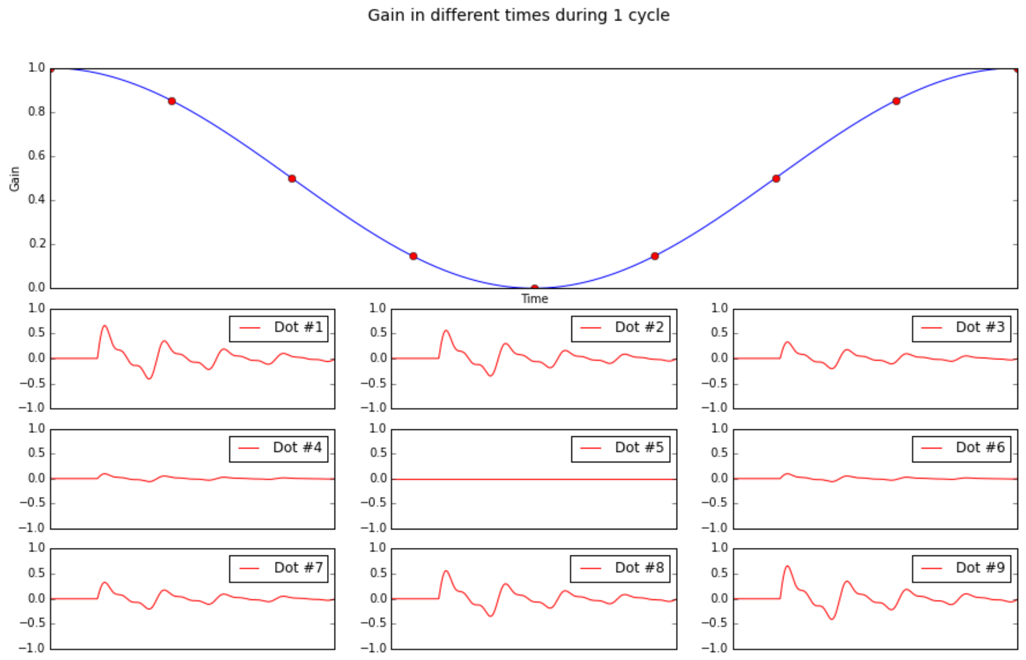

Figure 7: Amplitude modulation effect

In order to implement the effect, it is necessary to use a discrete time step, equivalent for the equation shown above. This is:

$$G(n) = 0.5\sin(\gamma n) + 0.5 \tag{4}$$

In this case,

$$\gamma = 2\pi \times \frac{f \times B_l}{S_r} \tag{5}$$

Were f is an arbitrary frequency given by the user (in Hz) and determines how "fast" or "slow" does the gain will change from zero to its maximum value; B_l is the *block length* and S_r is the *Sample rate*.

2.4 Effect 4: *Flanger*

The last effect to be implemented is the *flanger*. The basic idea behind the flanger is to create constructive and destructive interference, by adding up the original sound with a delayed sound, but with a varying in time delay. With this variation in time, the different frequencies of the sound spectrum will add up, or subtract by interference, creating a *wha-wha* sound, but with out the need of filtering the signal. In order to create the desired effect it is very important that the constructive and destructive interference happens. The interference happens when to sinusoidal waves are summed but with different phase (given by the delay), so in other to create interference the delay should vary around a period of the sound. Assuming a central frequency of $440Hz$:

$$T = \frac{1}{440Hz} \approx 2ms \tag{6}$$

This means that the delay should oscillate in the around 10^{-3} to 10^{-2} seconds.The delay as a function of time should follow the next equation:

$$D(n) = D_0(0.5\sin(\gamma n) + 0.5) \tag{7}$$

Where γ corresponds to the same value shown in the previous section. The next image shows the behavior of the delay as a function of time.

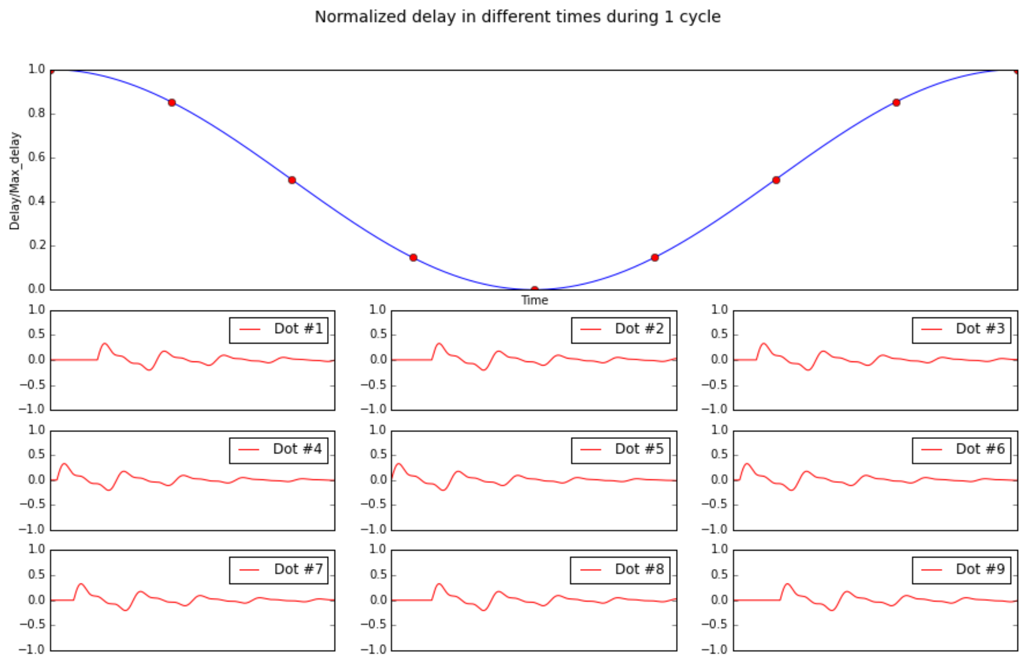

Figure 8: Flanger effect

3 DSP Implementation

3.1 Circular buffering

The way that the buffer was implemented in *Matlab* presents a disadvantage when trying to implement it on the DSP. Every time a new audio block is read, all the elements in the buffer vector have to change positions corresponding to the *block length*. This is a very inefficient way of buffering, and this is why the *circular buffer* is introduced. Circular buffering consists in not necessarily having the oldest samples in one end of the buffer and the newest in the other end, but just replacing the oldest blocks by new block, and changing the index for the current position. To accomplish this, the modular division has to be used. The next image shows how the *circular buffering* works.

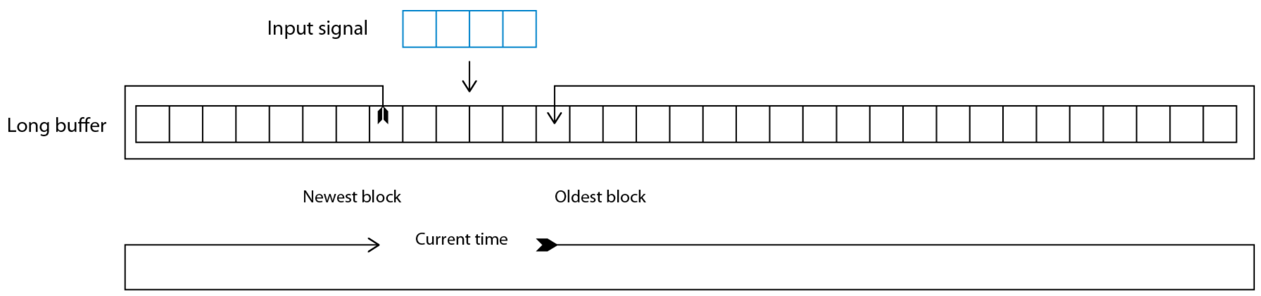

Figure 9: Circular buffering

3.2 Generalization of effects

In order to implement the previous effects into the DSP several considerations and constraints had to be taken into account based on the architecture of the given digital signal processor and the flexibility of C language. First of all, ideally the best way to implement the set of effects is by generalizing them so as to describe them by a limited set of coefficients. This was possible thanks to the use of C Structures and by reducing the degrees of freedom of the algorithm itself. Having in mind the basic delay block diagram of figure 2, if we assume the gain and the delay as being able to vary over time only by a sinusoidal shape, then the amount of information in order to describe every effect can be greatly reduced. As any sinusoid, the only required parameters are its amplitude, frequency, center (or DC offset as it is often referred to) and phase. By describing every effect by a set of different echo paths, each one of them containing independent sinusoidal gain and delay the mentioned description was transcribed into C language according to the following code:

```
struct varying
{

  float amp;
  float DC;
  float freq;
  float phase;
};

struct echo_path
{
  struct varying gain;
  struct varying delay;
};

struct effect
{
  unsigned int n_effects;
  struct echo_path effects[];
};
```

This way, every effect can be passed as an argument only through a pointer carrying information about a struct of type effect. Furthermore, a general and uniquely defined function does the job of loading samples from the buffer pointed by x, processing them and transferring them to an output intermediate array y:

```
void apply_effect(float *x, float *y, struct effect* _effect, unsigned int cur_callback)
{
    int m;
    int n;
    int delay;
    float amplitude;

  for(m=0;m<(*_effect).n_effects;++m)
    {
      delay = (int)(   (0.5*sin(2*PI*FRECUENCIA_BASE*(*_effect).effects[m].delay.freq*
          cur_callback + (*_effect).effects[m].delay.phase)+0.5) * (*_effect).effects[m].
          delay.amp + (*_effect).effects[m].delay.DC   );

      amplitude =  (   (0.5*sin(2*PI*FRECUENCIA_BASE*(*_effect).effects[m].gain.freq*
          cur_callback + (*_effect).effects[m].gain.phase)+0.5) * (*_effect).effects[m].
          gain.amp + (*_effect).effects[m].gain.DC   );

    for(n=0; n<DSP_BLOCK_SIZE; ++n)

    {
      y[n] = y[n] + amplitude*x[(LONG_BUFFER_SIZE+n+curPositionInBuffer-delay)%
          LONG_BUFFER_SIZE];
    }
  }
}
```

Where the aforementioned circular indexing is used in order to access the buffer of recorded samples. Also, as clear by the implementation, the sinusoids associated to gain and delay (referred to as amplitude and delay in the code) are independent. It is important to point out that given the algorithm description above (using the variable *callbacks* which is increased deterministically at a rate of 500 Hz), the maximum sinusoidal recommended frequency in the current version of the code would be 250 Hz which does not give any problem for the effects that are to be implemented for the current project.
Afterwards, when considering the DSP architecture, the algorithm was implemented using an interrupt-driven description. This gives the possibility of assigning priority to certain events according to what is considered to have real time requirements. The basic three interrupts used were based on the *framework* code available at the course webpage giving the highest priority to *process*, continuing with *keyboard* and meaning the lowest priority would go to an interrupt associated to the timer module. The job of each interrupt as thought for the current algorithm is described below:

- ***Process***: is the basic most important function of the code. This function is associated to the interrupt being triggered very time a new bock of samples is available. When a new block is available, it copies it into the buffer using circular indexing and applies the effect pointed to *curEffect* and saves it into the audio out array of structs. Also, it increases the callbacks counter which is used in order to keep track of the time feeding the sinusoids. This function has the highest priority and should run completely within a lapse of time determined by the number of samples per block times the sampling period.

- ***Keyboard***: This short function has the only task of starting the timer in order to update a new effect to the pointer *curEffect*. It is associated to the interrupt triggered by a button press and the main reason why there is a timer required is to solve the problem associated to the bouncing of the button state after pressing it.

- ***Timer***: This function is in charge of generating (or reading) a new effect according to the newly entered parameters and assigning it to the *curEffect* pointer. It is triggered some time after a button press. Furthermore, the reason it has the lowest priority is that it is hard for us to notice the amount of time it takes for the DSP to generate a new effect since the button press (as long as the audio processing interrupt is being triggered correctly every time it is supposed to). As the audio process is given higher priority, the timer interrupt will change the current effect pointer while the

ADC is sampling and when the process interrupt is triggered, it is going to apply the new effect correspondingly.

3.3 Interacting with the user

One of the challenging parts of the implementation on the DSP was the interaction between the user and the DSP. It was necessary to have the possibility of changing the effect and the parameters of some the effects using only 4 buttons, and 6 LEDs. The best way found to control the effects was using two buttons to circule around the effect or the parameter, and two buttons to increase or decrease the value of the parameter. The LEDs will show in what effect the user, in what parameter or the value of the parameter, depending on the last pressed button. For instance if the last pressed button was the one to circule around the effects, the LEDs will show in what effect the user is. The next image show the function for each button.

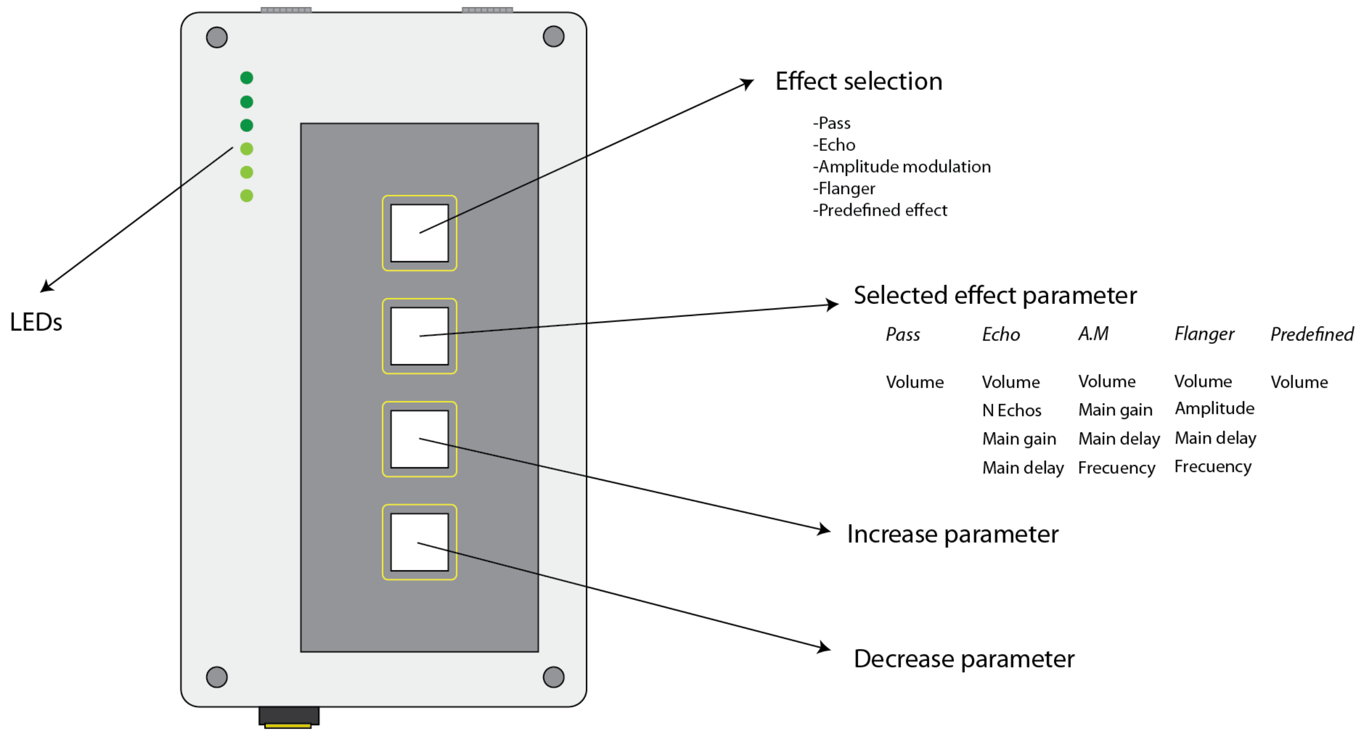

Figure 10: Flanger effect

4 Further implementations

4.1 Interpolation in flanger

The flanger effect presents one problem in the DSP implementation. The root of the problem is that the delay in some sense is a time measurement as a function of time. As a general idea, the delay is shifting as a sinusoidal function of time with a certain frequency f, and the long buffer is shifting positions every time a *sample block* has been processed, but always in the same direction. This means that when the delay is moving from a low value to a high value, some of the data in the buffer will not be processed, and when the delay is moving from a high value to a low value, some data will be processed twice. This can be seen in the next image:

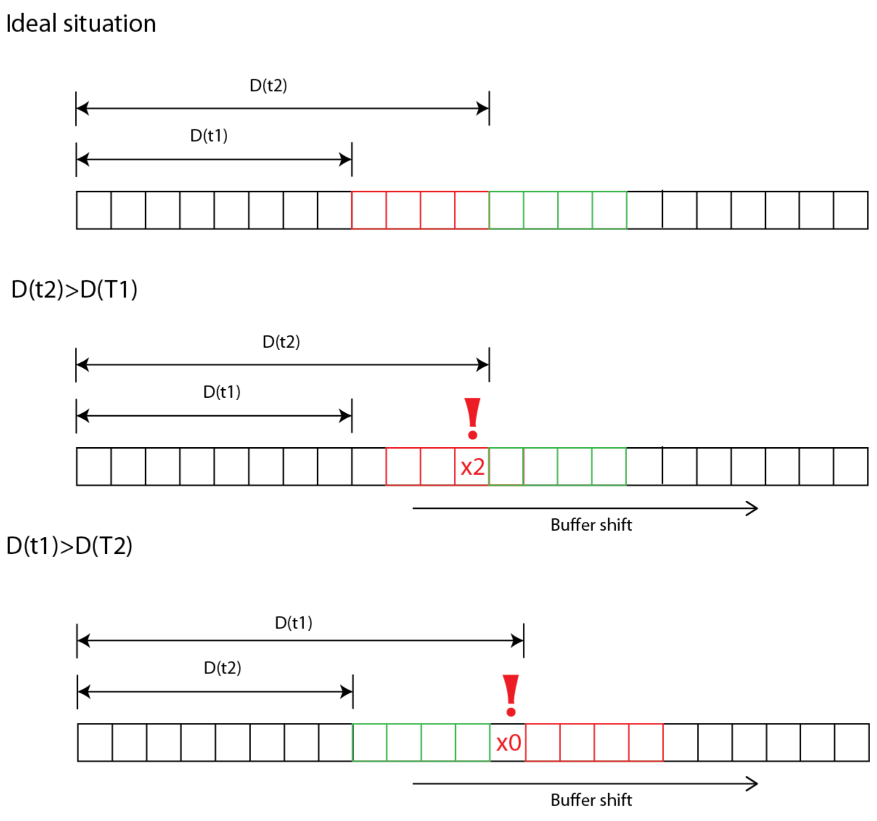

Figure 11: Flanger problem

4.2 Recursive sine function

The sine and cosine functions are very inefficient in the way they were implemented for the project, this is because every time a sine or cosine value is needed, it has to be numerically calculated, for this reason implementing a recursive way of implementing trigonometric functions will be important. To begin, the definition of discreet sine is the next:

$$f(n) = \sin(\gamma n) \tag{8}$$

But this can be expressed as the imaginary part of the complex exponential in the next way:

$$\sin(\gamma n) = \Im\left(e^{j\gamma n}\right) \tag{9}$$

Using the properties of the exponential function,

$$e^{j\gamma n} = e^{j\gamma(n-1)}e^{j\gamma} \tag{10}$$

This shows that in order to find the value of the complex exponential in the present time, there is only need to use the previous value. This gives a condition of starting to find values with $n = 1$, and using $n = 0$ with a known value as j (if the sine is needed to start in 1) or 1 (if the sine is needed to start in 0). As *gamma* is a constant determined by the user, the value of $e^{j\gamma}$ can be computed only one time (while γ remains constant).