Digital instrument effects using a DSP
Course ETIN80

Marcus Månsson - mob12mma@student.lu.se
Erik Söderberg - dat12eso@student.lu.se
Guoda Tian - gu7550ti-s@student.lu.se

Supervisor Mikael Swartling

March 24, 2017

Contents

1 Project description

The project scope was to design and implement an instrument effects library on a digital signal processor. The effects that were chosen were: an echo-effect, an adaptive gain controller and an overdrive effect. The initial steps consisted of simulating all three algorithms in MATLAB in order to understand the basics of the effects and also get acquainted with possible difficulties. After a successful simulation, all effects were implemented on a *SHARC ADSP-21262* using the IDE *Visual DSP++ 5.0* in the programming language C.

1.1 Hardware description

The hardware setup consisted of one DSP connected to a motherboard with 4 buttons. The interaction between the computer and the circuit was done using an emulator connected via USB. The setup can be seen below.

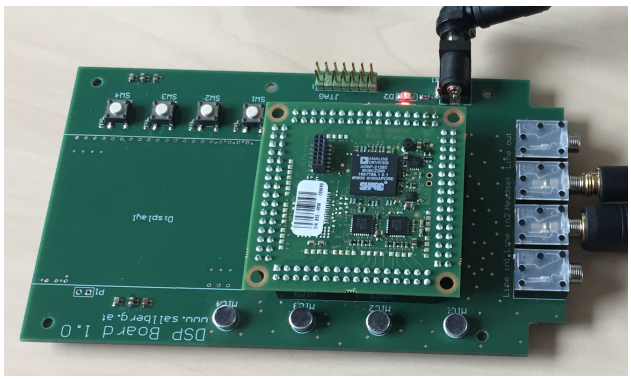

Figure 1: Hardware setup

2 Research

All research was done before implementation started. The main focus of the research was to find a theoretical basis for the chosen algorithms.

2.1 Echo filter

2.1.1 FIR filter

The echo filter was initially implemented using a first-order FIR filter. An FIR filter is a filter whose impulse response is of *finite* duration hence having no feedback of the processed audio. The test audio was recorded using a laptop and was then run through our MATLAB FIR algorithm. Initially an audio buffer was allocated to the size of the number of samples in the input audio plus the number samples of the desired delay. This is due to the fact that the

output audio consists of N overlapping copies of the original audio. Since the test FIR filter was of order one, only one extra delay length was needed for the allocation. The echoed audio was multiplied by a small attenuation factor to get a more realistic echo sound. The block diagram of a first order FIR filter can be seen in Figure 2 and the attenuation factor is visualized as parameter "a".

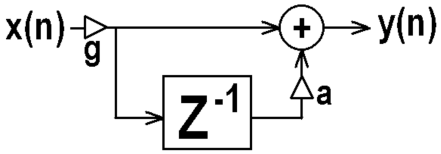

Figure 2: First-order FIR filter

2.1.2 IIR filter

As the name suggests, IIR (Infinite Impulse Response) audio filters have infinite impulse responses due to constant feedback of the output audio back to the filter. In practice however, this is not entirely true due to the finite precision of the DSP float values which each audio sample is stored as. Each audio sample read from the input is added together with an attenuated previous audio sample stored the delayed position away from the current position in the audio buffer. This new combined sound sample is fed to the output and also fed back to the current location in the audio buffer, added to whatever sound sample present in that location and finally stored. The attenuation factor that the previously stored audio sample is multiplied by is smaller than one and larger than zero so that the output cannot diverge, see Figure 3. Since no real-time processing is possible in MATLAB, all preparatory simulations on the IIR filter was done by chopping a pre-recorded audio track into 32 sample segments to mimic the sample size of the DSP. These segments were then iterated through and the IIR filter algorithm was performed on all of them with the result stored in an output sound track which could then be played once all segments had been processed.

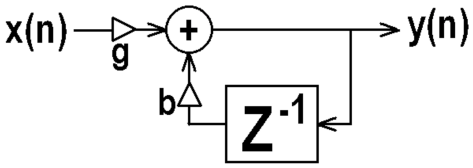

Figure 3: First-order IIR filter showcasing the feedback- and delay loop that is attenuated a factor b and added to a future input value.

2.2 Adaptive Gain Controller (AGC)

In this section a brief description of the adaptive gain controller (AGC) will be given. AGCs play an important role in both communication and acoustic systems. Both of these systems suffer from fluctuations of the input signal amplitude, therefore an AGC is required to enhance very low amplitudes, or suppress excessively high signals, in order to maintain the stability of the energy of output signal. In this project the AGC was placed as the last filter (after the distortion filter and the echo-filter), suppressing both background noise and the excessively high signal powers. The structure of the AGC consisted of mainly two parts, namely, power measurement and the gain scaling. The block diagram is given in figure (4), see in [1].

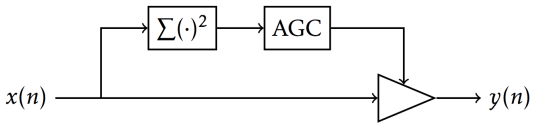

Figure 4: Block diagram of the AGC

2.2.1 Power measurement

Power measuring is the first step in the AGC system, the mathematical model of which is presented as follows:

$$Q(n) = \frac{1}{N} \sum_{k=0}^{N-1} x(n-k)^2 \tag{1}$$

$$P(n) = \alpha P(n-1) + (1-\alpha)Q(n), \tag{2}$$

where $x(n)$ in equation (1) represents the input signal $Q(n)$ is the average signal power of the N points which on the DSP, N = 32. $P(n)$ in equation (2) is denoted as the output signal power passing the first-order filter.

Based on the mathematical model above, N = 32 sample points of the input signal were read and the average power of these were then calculated. Then, in order to smooth the variance of the average power, $Q(n)$ was passed into a first-order feedback filter. Finally, the output signal $P(n)$ was generated for the second step gain scaling.

2.2.2 Gain scaling

After the power measurement, the gain factor was assigned based on the graph in figure 5 As can be seen in the graph, there exists two cut-off amplitudes, namely, -50dB and -20dB for the lower and higher amplitude respectively. If the amplitude of the signal is lower than -50 dB, the amplitude is set to zero in order to completely suppress the low level background noise. If the signal

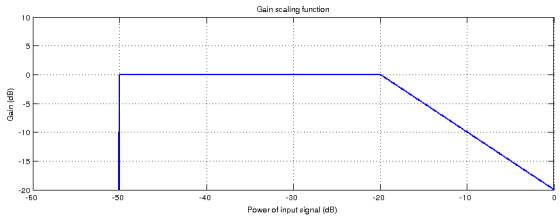

Figure 5: Gain scaling function for input audio used on the DSP platform

amplitude lies between -50dB and -20dB, the amplitude is set to 1 in order to let the signal pass through. If the signal amplitude is too high, a mathematical function $y = -x - 20$ is utilized to suppress the excessively high signal.

2.2.3 Simulation results for the AGC

The simulation results from the AGC code ran in MATLAB is shown in figure 6. In the figure, the red line represents the original signal while the blue line is denoted as the output signal after the AGC. It is clear that AGC significantly reduces the background noise and the input signals with excessively high amplitudes.

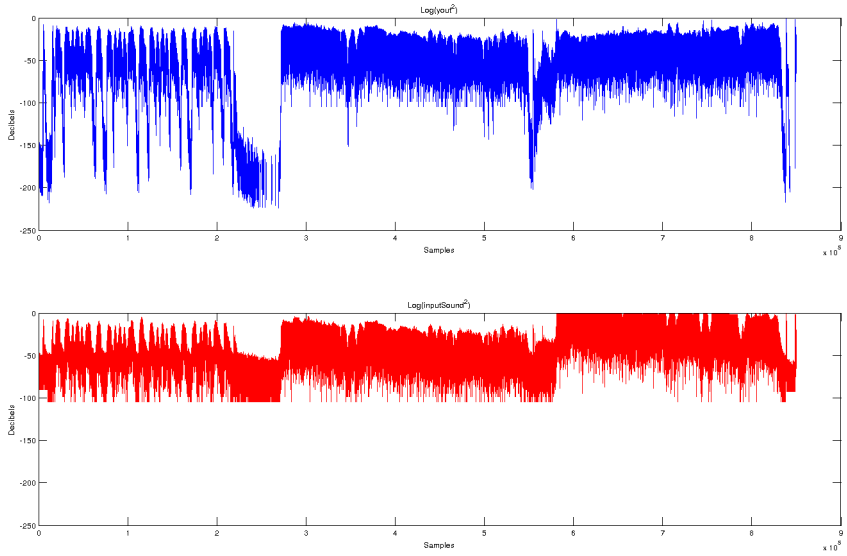

Figure 6: Simulation results for AGC.

2.3 Distortion and overdrive filter

In music, the distortion and overdrive effect is produced by clipping of the audio signal due to the power of the signal being greater than the maximum output power of the amplifier. When this happens the part of the signal that is outside the maximum output power gets clipped resulting in a distorted sound, see Figure 7. In the frequency domain this clipping will yield harmonics at a higher frequency than the input signal. The difference between distortion and overdrive is that the clipping is softer for overdrive resulting in a warmer and not as "dirty" sound as with distortion.

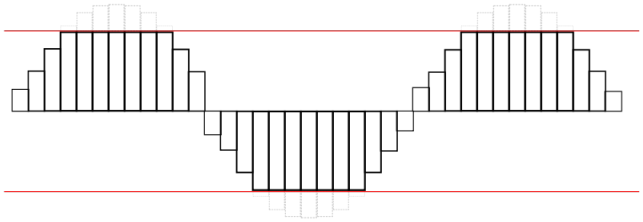

Figure 7: Clipping of discrete amplitude samples.[2]

To create a digital distortion or overdrive a non-linear function needs to operate on the input signal to perform the desired clipping [3]. The function originally used in this project looked like equation 3.

$$f(x) = \frac{x}{|x|}(1 - e^{(\alpha x^2/|x|)}) \tag{3}$$

This function was implemented in MATLAB and made to work in cascade with the other filters and then transferred to the DSP. Due to the relatively heavy calculations (several divisions by a variable etc.) and many steps required to perform this filtering this implementation was abandoned in favor of the less demanding overdrive algorithm from equation 4 below.

$$f(x) = \begin{cases} 1, & \text{if } x > (2/3) \\ (1/3) * (3 - (2 - 3x)^2), & \text{if } (2/3) \geq x > (1/3) \\ 2x, & \text{if } (1/3) \geq x > 0 \\ -2x, & \text{if } -(1/3) < x \leq 0 \\ (1/3) * ((2 + 3x)^2 - 3), & \text{if } -(2/3) < x \leq -(1/3) \\ -1, & \text{otherwise} \end{cases} \tag{4}$$

This algorithm only contains some divisions by a constant, multiplications and additions. The divisions 1/3 and 2/3 are pre-calculated at the startup of the DSP and then used as constants during runtime for a small increase in efficiency.

3 Problems

3.1 Memory allocation

When working with low-level languages such as C, there is always the potential for index-out-of-bounds errors. Since the real-time implementation on the DSP used circular indexing of the buffer we encountered a lot of problems when adding and reading samples. Either we would hear an erroneous sound from the output or the program would crash, setting the DSP in a default mode. We solved this problem by testing the algorithm for the circular indexing and also by drawing the buffer on paper and stepping through simple examples.

3.2 Button bouncing

The buttons hosted by the motherboard proved to give what is called button bouncing. This phenomenon can be described as the circuit detecting multiple button presses upon pressing a button only once. Since we implemented one button for toggling between different lengths of the echoing it was irritating to obtain "random" lengths of the echoing upon pressing the button. This was solved by using an already existing timer function in the given framework file. Each time we pressed a button we called the timer function. It then started a clock counter and for each time it detected a new press it would reset the counting. Only when the counter was allowed to count to a specified number, the button would be registered. In this way small bounces and fast clicks would be taken care of and not affect the outcome in an unwanted way.

3.3 Global variables

Our first approach to work with the audio buffer was by implementing all functionality in one file and declaring the audio buffer in the main function. Later when we decided to modularize the code and outsource the functions in different files we started to encounter problems with output. In this stage we had declared the audio buffer as a static variable in the header file. This meant that each separate file created an own copy of the buffer rather than modifying the same central buffer. This of course resulted in an unwanted output. We solved this problem by declaring the buffer as *"extern"* which means that the declared name is preserved and hence works as a global variable.

3.4 Sample frequency problem

We wanted to use 44.1kHz as sampling frequency since this is the sample frequency used in mp3. This was not an option available to us so we tried to add it ourselves but failed. We then changed to 48kHz which was an available frequency but this made our audio buffer for the echo too large to store in memory due to our echo being one second long. We then switched to 8kHz in order to fit a one second echo but this severely reduced the sound quality. Finally we

reduced the length of the echo to a maximum of 0.3 seconds and increased the sampling frequency back up to 48kHz increased the sound quality drastically and also made for a more usable echo, one second was too long.

3.5 Calculation heavy distortion

When we ran the original distortion function on the DSP it was very harsh and not at all pleasant. We had to implement two additional functions in order to make it work, one called sign(float) and one called maxi() which slowed down the program some. It also contained a lot divisions by a variable which is a computationally heavy action to perform in real-time. In order to work around this we implemented the much simpler algorithm in equation 4.

3.6 Circular indexing

The circular index is widely used when implementing IIR filter. First of all, we wrote the circulation operation ourself by using module function and several if statements. However, afterwards, we enhanced the code by using the circindex function provided by the DSP library. There are three parameters in the circindex function, namely, current position of the pointer, the step, and the buffer size. It is worth noticing that the step can be both positive and negative.

4 Final remarks

All mentioned effects were successfully implemented on the DSP. Even though we experienced some minor difficulties during the testing on the actual guitar, we are all in all satisfied with the result. Furthermore we agree that the work flow of this project was very efficient. Starting with theoretical research followed by statical simulation was indeed a successful approach before starting to work with the DSP. It made the progressions smoother and less prone to errors and difficulties. Working with this project has taught us how to write and optimize real-time low-level code and how different audio effects can be implemented.

If we would have had more time to work on this project we would have liked to improve our overdrive function so that it worked better for high frequency audio. We would also have liked to look more into audio effects such as different real-time configurable equalizers. Finally we would have liked to refactor our code and outsource our AGC algorithm to a separate file.

References

[1] http://www.eit.lth.se/fileadmin/eit/courses/etin80/lectures/slides-lecture2.pdf
page loaded on 03.01.17.

[2] By David Batley (user:h2g2bob) - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1764293
picture fetched on 03.01.17.

[3] http://users.cs.cf.ac.uk/Dave.Marshall/CM0268/PDF/10_CM0268_Audio_FX.pdf
page loaded on 03.01.17.

The source code for the project can be found at https://gitlab.control.lth.se/FRT090-2016/group-i