Guitar effect processor on microcontroller

June 16, 2020

1 Introduction

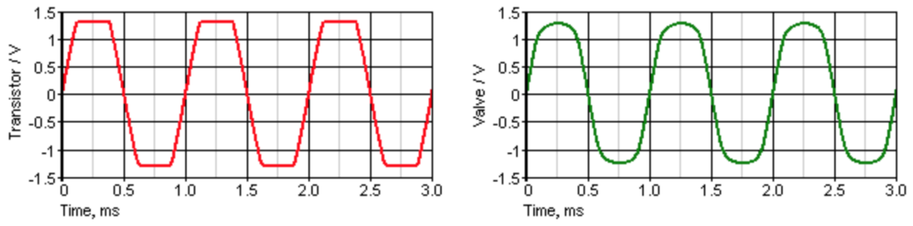
In this project, I have used different stm32 nucleo boards with the stm32 cube IDE and the DAC CS4344/ADC CS5343 board in order to program a guitar effect processor. I have used the tutorial [1] as a basis in order to try implementing the following effects: distortion, delay, chorus and reverberation. I managed to have a working prototype of every effect I wanted, both on a python framework and on the F413ZH microcontroller.
The distortion was a little bit different compared to the other effects, as it corresponds simply to a non-linear function. On the other hand, the delay was the main building block for both the chorus and reverberation algorithms.

2 Distortion

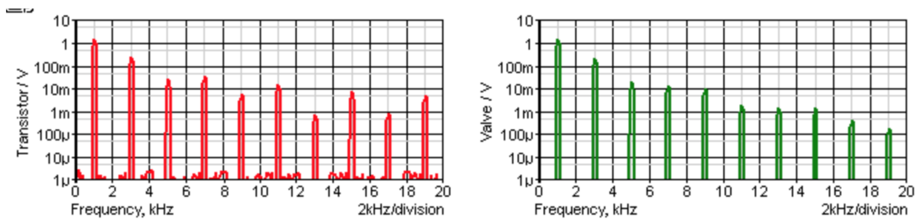
2.1 Theory

The distortion effect has first been used in the 1960s, and originally comes from the sound made by vacuum-tube powered amplifiers when they are pushed to their maximum. To mimic this analog effect digitally, the idea is to clip the signal, making it reach its maximum values. It can be done by soft clipping or hard clipping, which respectively flatten the signal softly or abruptly (see Figure 1(a)).

Soft clipping is obtained with valve amplifiers whereas hard clipping comes more often from transistor amplifiers [2]. Soft clipping is usually considered better for guitarists, as it creates less harsh sounds, but the valve amplifiers that make soft clipping are usually more expensive than their transistor equivalent. On figure 1(b) we can notice that hard distortion creates more powerful high order harmonics (for example 11th, 15th and 19th) than the soft distortion, which leads to a harsher sound. It makes sense since it has sharper edges and looks more like a square wave in the time domain, and the spectrum of the square wave contains all odd-number harmonics. Therefore, in our implementation, we expect to have high order harmonics with significant higher levels in the case of hard compression than in the case of soft compression.



(a) distortion in time



(b) distortion in frequency

Figure 1: Hard (in red) and soft (in green) distortion

2.2 Implementation

The distortion effect was implemented both in symmetrical soft clipping, described by equation (1) (from [3]), and simple hard clipping described in equation (2).

$$f(x) = \begin{cases} 2x & \text{if } |x| \in [0; \frac{1}{3}] \\ \frac{3-(2-3x)^2}{3} & \text{if } |x| \in [\frac{1}{3}; \frac{2}{3}] \\ 1 & \text{if } |x| \in [\frac{2}{3}; 1] \end{cases} \tag{1}$$

$$f(x) = \begin{cases} \frac{3}{2}x & \text{if } |x| \in [0; \frac{2}{3}] \\ 1 & \text{if } |x| \in [\frac{2}{3}; 1] \end{cases} \tag{2}$$

The first major problem I faced to do this distortion was that I could not get the 24 bit ADC and DAC to work properly: although the passthrough was working, the values registered by the microcontroller were not correct. After some time we decided with adrien to switch to 16 bits, using the same settings as in the lab [1].
Since we use values in 2's complement signed integer in 16 bits, equation (1) was adapted to take as input and output values in the range $[-32'768, 32'767]$, rather than in $[-1, +1]$. The most complicated part of the implementation was with the $\frac{3-(2-3x)^2}{3}$ function, because I had to take care of the possible overflows and arithmetic problems.
In the end this is the code for symmetrical soft clipping (equation 1):

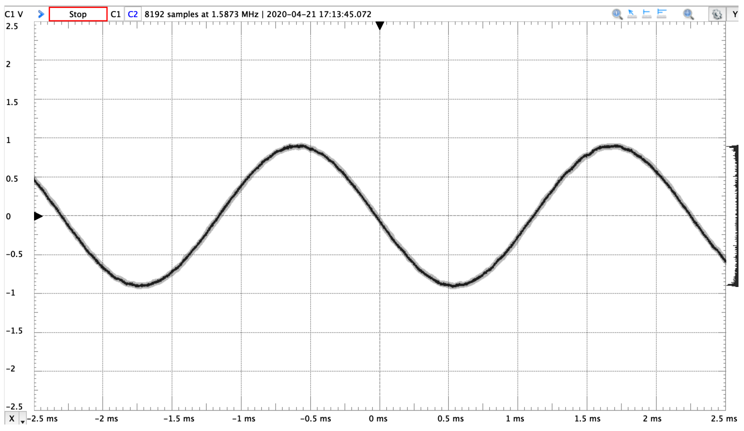
```
int16_t inline distort(int16_t x16){
    if (x16 > MAX_TH23) :
        return MAX_INT;
    else if (x16 < -MAX_TH23) :
        return -MAX_INT -1;
    else if (x16 > MAX_TH13 && x16 <= MAX_TH23 ):
        int32_t a=3*x16;
        float aprim=2*MAX_INT -a;
        float b = aprim/MAX_INT;
        float c = b*b;
        float d = (3 - c)/3;
        int32_t r= d*MAX_INT;
        return r;
    else if (x16 < -MAX_TH13 && x16 >= -MAX_TH23 ):
        int32_t a=3*x16;
        float aprim=2*MAX_INT + a;
        float b = aprim/MAX_INT;
        float c = b*b;
        float d = (3 - c)/3;
        int32_t r= - d*MAX_INT;
        return r;
    else :
        int32_t a=2*x16;
        return a;
    }
```

The code for hard distortion (equation 2) is :

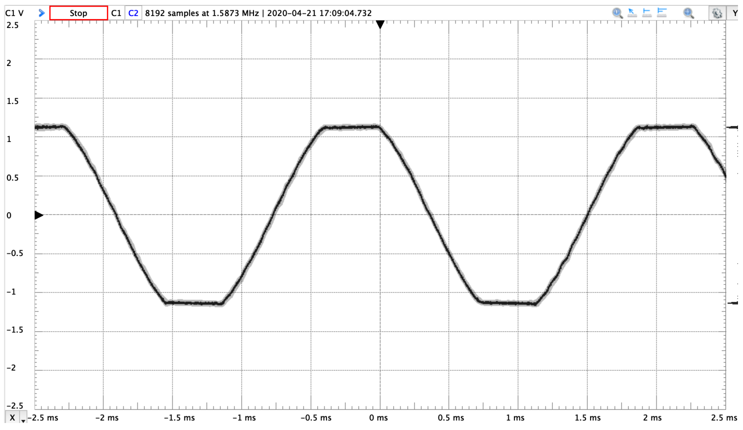
```
int16_t inline distort(int16_t x16){
    if (x16 > MAX_TH23) :
        return MAX_INT;
    else if (x16 < -MAX_TH23) :
        return -MAX_INT -1;

    else :
        int32_t a=x16/2*3;
        return a;
    }
```

For 16 bit data, we have `MAX_INT` $= 2^{15} - 1 = 32767$, `MAX_TH13 = MAX_INT` $\times \frac{1}{3}$ and `MAX_TH23 = MAX_INT` $\times \frac{2}{3}$.

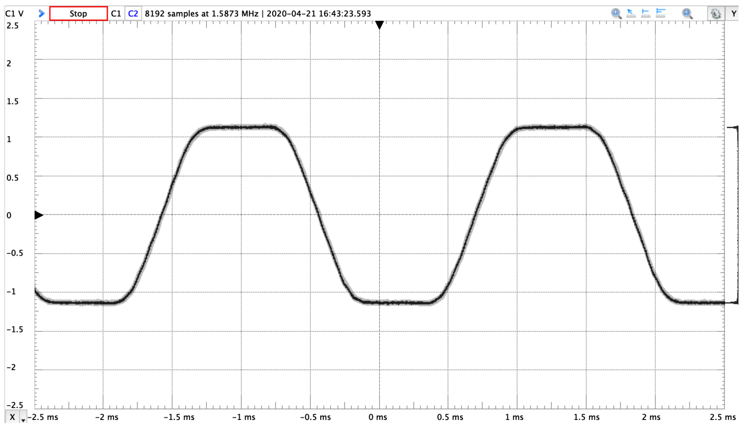
2.3 Results



(a) input



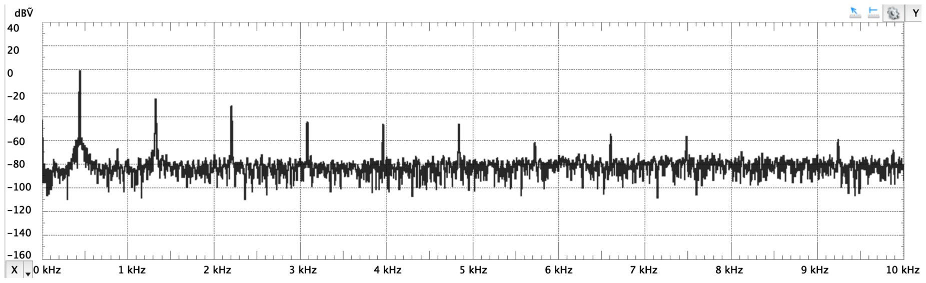
(b) Hard distortion



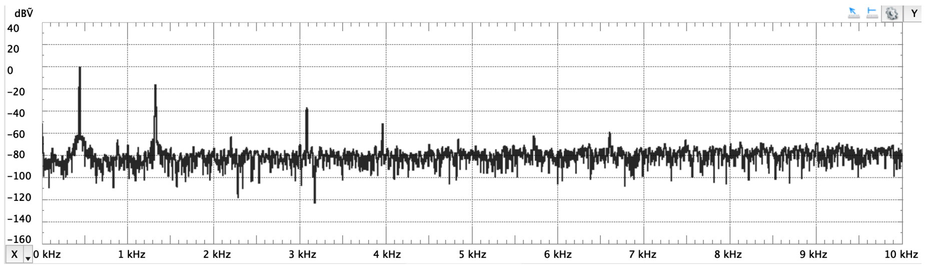
(c) Symmetrical soft distortion

Figure 2: Distortion Effect in time

In the end, the results are quite good: the measurement of the time and frequency effect on figures 2 and 3 are very similar to the theory of figure 1. We can notice that we have less harmonics and they are weaker when we use symmetrical soft clipping instead of hard clipping: the fifth harmonic is almost absent in symmetrical soft clipping, while it is quite strong in hard clipping, and the last real harmonic peak is the 9th harmonic in the symmetrical soft clipping, whereas we have a peak at the 21st harmonic in the case of hard clipping.
However, when listening to the sound outputs, it is not really obvious that the soft clipping creates less harsh sounds than hard clipping.



(a) hard clipping



(b) symmetrical soft clipping

Figure 3: Distortion Effect in frequency

3 Delay

The delay effect exists since the 1940s, and was originally created with tape loops. The goal is to create an echo to the original sound. In the book [3], Udo Zolzer describes 2 different kinds of delays, which can be described by the following equations:

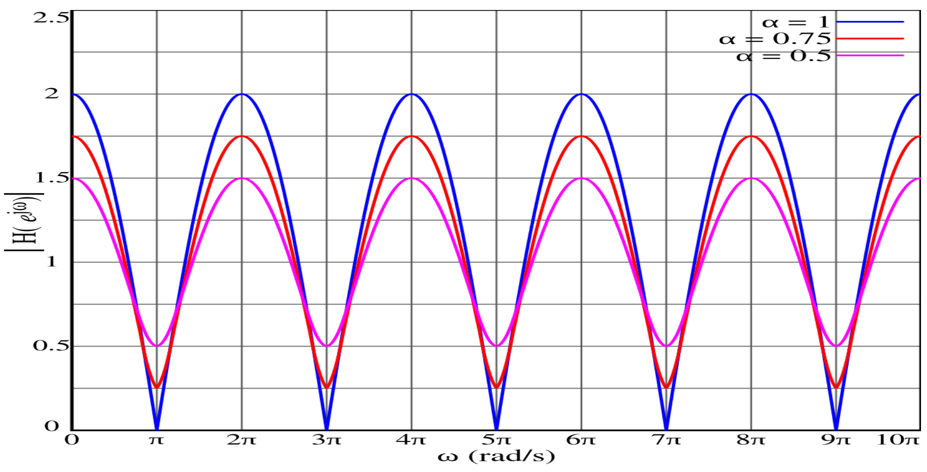
$$y(t) = x(t) + \alpha x(t - D) \tag{3}$$

and

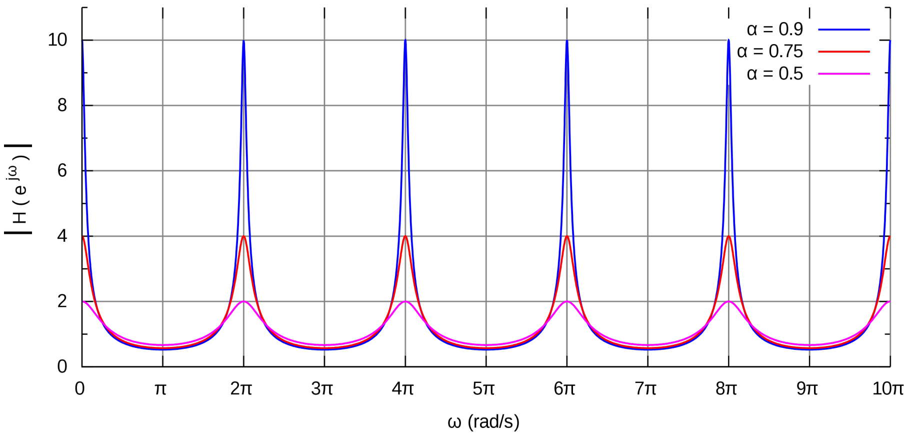
$$y(t) = x(t) + \alpha y(t - D) \tag{4}$$

Equation 3 describes the FIR delay, and equation 4 the IIR delay. If we calculate the z-transform corresponding to these delays, we obtain respectively $1 + \alpha z^{-D}$

and $\frac{1}{1-\alpha z^{-D}}$. Their magnitude frequency responses are $\sqrt{(1+\alpha^2)+2\alpha cos(wD)}$ (Figure 4(a)) , and $\frac{1}{\sqrt{(1+\alpha^2)+2\alpha cos(wD)}}$ (Figure 4(b)).



(a) feedforward (FIR) delay



(b) feedback (IIR) delay

Figure 4: Frequency responses of FIR and IIR delays

Those filters are periodic with a period of $\frac{1}{D}$, and they can create strange sound coloration.

3.1 First microcontroller implementation

The first delay type I implemented was the FIR delay, described by equation (3). At the beginning, I always obtained artefacts as shown on figure 5, spaced by 32.6ms, which was the buffer length.

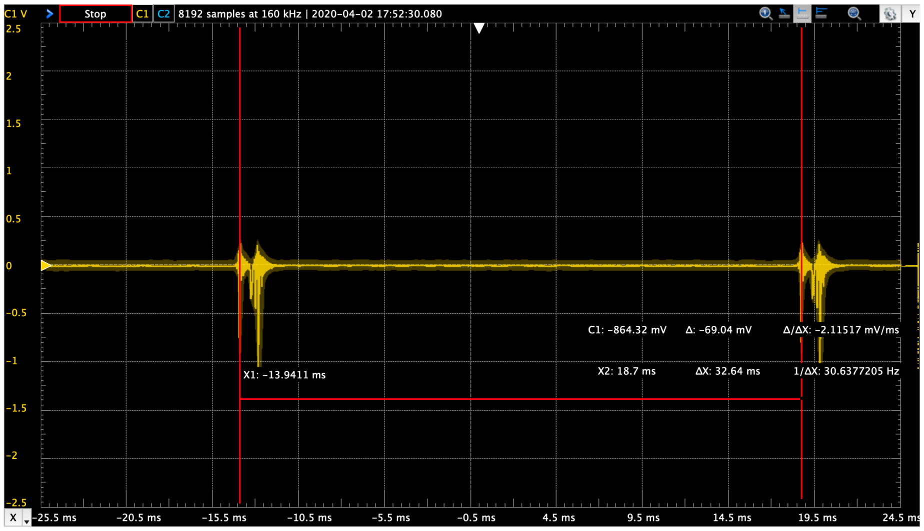
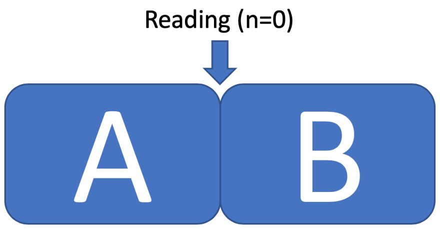
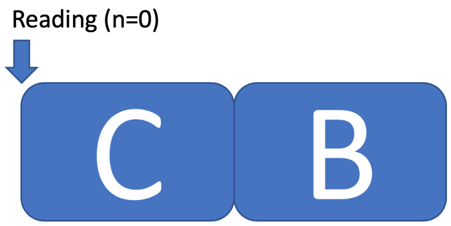


Figure 5: Buffer artefacts

The artefacts were due to the fact that the buffer indexing was not circular. The buffer gets filled by half buffers and sends an interruption at every half buffer, which means we are either in case 1 or case 2 described by figure 6.



(a) case 1



(b) case 2

Figure 6: Buffer filling

When we are in case 1, we simply have to take the sample at $n - D$ if D is the delay, but in case 2 we have to take the sample at $n - D + BufferSize$ while $n < D$, and then we can take the sample at $n - D$. This was implemented in the code by passing an extra argument to the process function to know whether we are in case 1 or case 2. You can also notice that the maximum achievable delay is only of the size of half a buffer, because if we go further than that we come back to the beginning of the current half buffer.
The code of the FIR delay function is :

```
void inline Process(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp) {

static float y=0;
static int16_t y16 = 0;

for (uint16_t i = 0; i < size; i += 2) {
    int16_t y = (int16_t) *(pIn+i);

    int16_t delaysig;

    if(!halfcomp){ \\case 1
        delaysig = *(pIn+i-M_DELAY);
    }
    else
    { \\case 2
        if (i >= M_DELAY){ // (pIn+i-M > pIn)
            delaysig = *(pIn+i-M_DELAY);
        }

        else{
            delaysig = *(pIn+FULL_BUFFER_SIZE+i-M_DELAY);
        }
    }
    y = y + delaygain*delaysig;
    y16 = (y/maxnum) * 0x7FFF;
    *pOut++ = y16;
    *pOut++ = y16;
}
```

You can also notice that I do a dynamic range control, by using a float as first output and then converting it back to 16 bit integer, so that I never get a saturating output or overflows.
The F072 microcontroller has 16kB RAM, which means we can store 8000 16 bit points in total, therefore 4000 points in every half buffer and 2000 points per channel. This means that the maximum delay achievable with this technique and the F072 microcontroller is of only 62.5 ms. Therefore we looked at another technique in order to implement the delay by decimating and reconstructing the sound, without using additional memory resources.

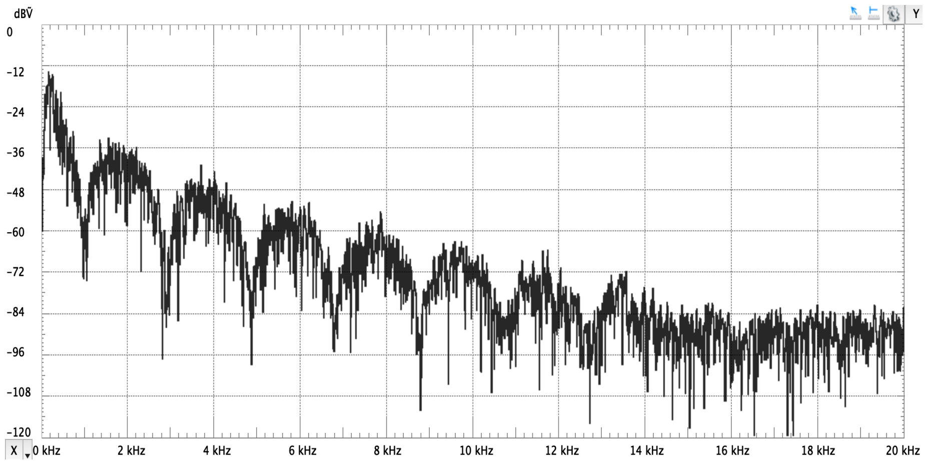
3.2 Offline Python decimation delay

The idea is to add a third buffer of the same size as the input and output buffers, decimate the input and store it in this buffer in order to achieve a 10 times longer delay. We will first downsample the signal 10 times to store it on the buffer, then upsample it 10 times to reconstruct it.
If $X(e^{jw})$ is the original fourier transform, the downsampled signal's fourier

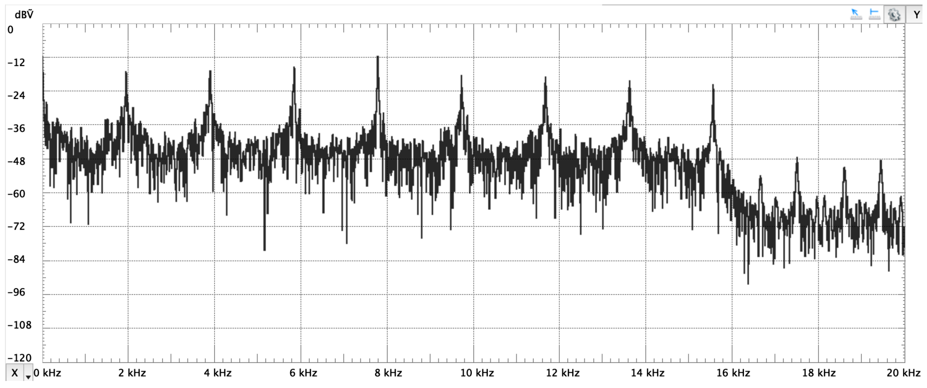
transform is $\frac{1}{N}\sum_{m=0}^{N-1} X(e^{j\frac{w-2\pi m}{N}})$ [4]. The fourier transform of the downsampled signal consists in shifted copies of the original fourier transform, shifted by multiples of $\frac{2\pi}{N}$, and the $\pm\frac{\pi}{N}$ interval is extended to $\pm\pi$. Since we downsample the signal 10 times, we have 10 shifted copies of the fourier transform, each shifted by multiples of $\frac{2\pi}{10}$. To prevent aliasing, we therefore need to apply a lowpass filter with frequency $\frac{\pi}{10}$, and then we take only 1 sample out of 10 to downsample.
If $X(e^{jw})$ is the original fourier transform, the upsampled signal's fourier transform is $X(e^{jwN})$ [4] (if the upsampling is done by adding zeros in between every input sample). The fourier transform of the upsampled signal is therefore the same as the original signal, except that the $\pm\pi m$ interval is compressed to $\pm\pi$ (remember that the fourier transform is 2π periodic). In our case, it means that we have to lowpass the signal after reconstruction by a frequency of $\frac{\pi}{10}$.
All this processing was performed offline using python and the scipy signal processing library. The processing was done separately for each buffer so that the algorithm could be moved to the STM board easily. However, since we use a 32kHz sampling frequency on the microcontroller, this means we have to perform the lowpass filter at 3200Hz, which deteriorates the sound quality, as we could hear using the python implementation. We therefore decided to change the microcontroller to one with more RAM, which allows longer delay.

3.3 Second microcontroller delay implementation

We first chose to use the microcontroller H743ZI, which has 1024kB RAM, 2048kB flash memory and a frequency of 480MHz. For some reason, I never managed to get the I2S communication to work properly on this device, after 2 weeks of trying to debug it. I used the analog discovery 2 oscilloscope to analyse the signals coming in and out, and the clock signals were working fine, but there was no data signal. Adrien also ordered the same microcontroller to try to find out why it was not working, without success yet.
With the F413ZH, the limit on the delay time actually does not come from the RAM, but from the fact that the size of the full buffer has to be an unsigned 16 bit integer in order to be used by the built-in DMA functions to transmit and receive packages, and therefore it has to be less than 65536. This gives us a maximum number of frames per buffer of 16383, which is around 512ms. Since we need to have a latency of the size of the frames per buffer, this delay is already too big for our guitar effect application, therefore I did not look into details how to further increase the buffer size.
I also managed to implement the IIR delay on the F413ZH, which has almost the same code as the FIR delay. The experimental comb filters with delays of 0.0005s (therefore 2000Hz periodic) can be seen on figure 7, and we can see they match well with the theory of figure 4.



(a) feedforward (FIR) delay



(b) feedback (IIR) delay

Figure 7: Experimental frequency responses of FIR and IIR delays

Delay is more a time domain effect than a frequency effect, since the goal is to create an echo of the sound. Therefore, it would have been interesting to observe the impulse response of this delay. I tried to do this, but having a short impulse at the output is complicated as there is a lowpass filter at the output of the ADC.
To reduce latency while having large delays, one solution would have been to have short buffers for input and output, but keep long buffers to do the delay.

4 Chorus

The goal of the chorus is to sound as if there were several instruments playing the same thing, almost at the same time, as a choir. To create a chorus effect, several copies of the input are delayed in the range 10 to 25ms with small and random variations of the delay time [3]. When the delay is very low, it is not perceived by our ears in the time domain, but in the frequency domain. This means that instead of hearing 2 truly separate sounds one after the other, we will hear a slight detuning of the sound. The chorus effect can be implemented using either the IIR or FIR delay. In the FIR case, the chorus effect can be written as the following equation :

$$y_n = x_n + g_1 x_{n-D_1 n} + g_2 x_{n-D_2 n} + g_3 x_{n-D_3 n} + ... \tag{5}$$

One common choice for D_n [5] is to use:

$$D_{kn} = F_k + \frac{D_k}{2}(1 - cos(2\pi \frac{f_k}{f_s} n)) \tag{6}$$

where F_k is the offset, D_k is the depth and f_k is the frequency of the change of delay. To create the chorus effect with a single branch, the offset is usually around 10ms, the depth around 15ms and the frequency less than 1 Hz. If we remove the offset and use a depth of 0 to 2ms, this creates the flanger effect.

4.1 Implementation

My first implementation of the chorus effect was done offline on python, while waiting for the new microcontroller. I recreated artificially the buffer filling system explained figure 6 on python, so that I would be able to adapt this code easily on the microcontroller. I obtained quite good results with an offset of 0.01, a depth of 0.015 and a frequency of 0.7Hz. The only problem is that we can hear a slight saturation.
I then implemented the chorus effect on the F413ZH board, using a lookup table for the cosine function. This is the final code of the process function:

```
int inline delaylen(uint16_t nglob){
    float delaylen =0;
    float a =COS_TABLE[nglob];
    float c = a/0x7FFF;

    delaylen = OFFSET + (DEPTH/2) * (1-c);

    int16_t mdelay = delaylen*SAMPLING_FREQ;

    return mdelay;
}

void inline Process(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp) {

    static uint16_t nglob = 0;
    static int16_t x_prev = 0;
    float y = 0;
    for (uint16_t i = 0; i < size; i += 2) {
        int16_t xn = (int16_t)(*(pIn+i) - x_prev);
        x_prev = *(pIn+i);

        int16_t delaysig;
        int16_t mdelay=delaylen(nglob);
// mdelay has to be an even number so that the delay stays in a single channel
        mdelay = floor(mdelay /2)*2;
        nglob=nglob+1;
        nglob %= COS_TABLE_LEN;

        if(!halfcomp){
            delaysig = *(pIn+i- mdelay);
        }
        else
        {
            if (i >= mdelay){ // (pIn+i-M > pIn)
                delaysig = *(pIn+i-mdelay);
            }

            else{
                delaysig = *(pIn + i - mdelay + FULL_BUFFER_SIZE);
            }
        }

        y = delaysig + xn;
        y = y / maxnum;
        int16_t y16 = y * 0x7FFF;
        *(pOut+i) = (int16_t)y16;
        *(pOut+i+1) = (int16_t)y16;
    }
}
```

5 Reverberation

5.1 Theory

When a sound is recorded inside a room, the recorded sound is the convolution of the direct sound with the room's impulse response. Different rooms have different impulse responses, which create different reverberation effects. A room impulse response is displayed on figure 8.

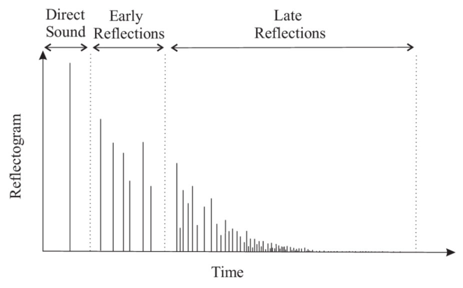


Figure 8: Room Impulse Response

As we can see, the impulse response is composed of the direct sound, the early reflections and the late reflections. The direct sound travels directly from the source to the listener without having bounced on a wall before. The early reflections are usually sparse as they have bounced on the walls only a few times before arriving at the listener, and they can improve sound intelligibility. On the other hand, we have a lot of late reflections, which give more the sense of space of the room[5]. In order to simulate the reverberation of a room, we have 2 main options: impulse response convolution, or feedback delay networks [6].

5.1.1 Impulse response convolution

The first option is to directly convolve the signal with the room's impulse response. The problem with this option is that to have a real reverberation effect, you need the room impulse response to be up to several seconds long. Using the time convolution formula $y_m = \sum_{n=0}^{N-1} h_n x_{m-n}$, at a sampling frequency of 32000Hz, we would need to perform 64000 multiplications and additions for every sample, in a time of $31\mu s$. This could be done with a 4 GHz processor, but not with our microcontroller.
However, it is also possible to perform the convolution in the fourier domain, and process the data by full buffers instead of processing each sample independently. This is thanks to the convolution theorem $f * g = \mathcal{F}^{-1}(\mathcal{F}(f)\mathcal{F}(g))$. The problem is that in the fourier domain we perform circular convolution, which means that part of the output is corrupted. Therefore we have to use different reconstruction tricks in order to avoid this. This is what is called the partitioned convolution algorithm [7].
Implementing this algorithm was the subject of my previous semester project, therefore I will try to focus here on the feedback delay network solution. Furthermore, in the case of implementing a reverberation effect, it is not very interesting since the only control over the reverberation we have with this technique is to change the length of the impulse response.

5.1.2 Feedback Delay Network

Another way to create the reverberation effect is to use a delay network in order to simulate a certain impulse response. The first idea was to simply use several different IIR delays. As we saw, the problem with this system is that if the delay is too small, we have a frequency coloration of the sound, as it corresponds to a comb filter (figure 4). Therefore, an allpass filter was needed.
This was done by Schroeder in [8], where he proposes a system to avoid having frequency effects when using delay networks. Schroeder's impulse response is $h(t) = -g\delta(t) + (1-g^2)(\delta(t-\tau) + g\delta(t-2\tau) + ...)$. The filter can be applied with

$$y_n = ay_{n-D} - ax_n + x_{n-D} \tag{7}$$

The corresponding frequency response is $H(w) = e^{-iwD}\frac{1-ge^{iwD}}{1-ge^{-iwD}}$, whose absolute value is equal to 1, as the ratio of 2 complex conjugate vectors is equal to 1.

Even with a single allpass filter, we still have the problem of flutter echos: when the same sound is repeated exactly periodically, it is not pleasing at all to the ear. This is caused by a lack of echo density. This problem is solved by Schroeder in [8] by using 5 allpass filters in series with different delays.
However, later on in [9], he developed a different system which is now a reference in the field of artificial reverberation. The idea is now to have a more realistic system, resembling more that of a true impulse response. The idea behind the system is that since real rooms have highly irregular frequency responses, it does not make much sense to insist on having a perfectly flat impulse response, with only allpass filters. Schroeder's reverberator is shown on figure 9.

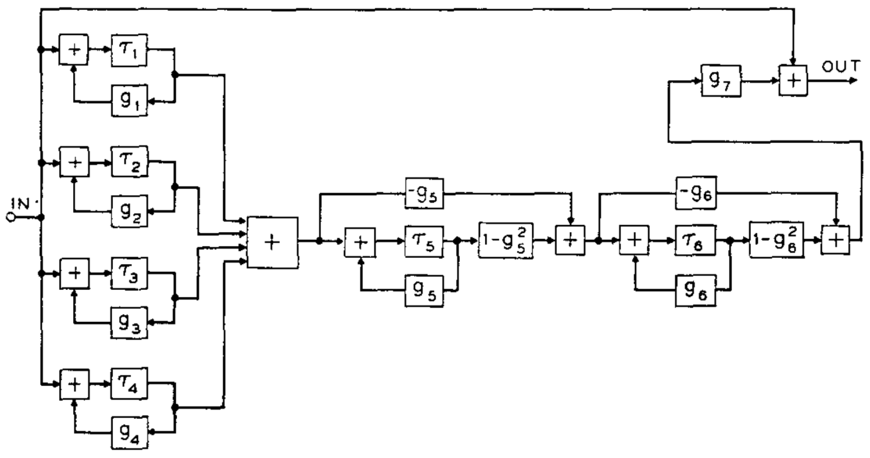


Figure 9: Schroeder's Reverberator [9]

This reverberator consists in 4 comb filters put in parallel, followed by 2 allpass filters in series. It allows for a wide choice of tuning parameters (mixing parameters and delay time), and can create a lot of different realistic reverberations. Another advantage it has compared to the simple allpass delay network is that you are able to control the amount of reverberation with respect to the direct sound, with gain 7 in the figure.

5.2 Implementation

The first reverberation implementation I did was on python, creating the reverberation described in [8]. The first step was to create an allpass filter using the equation 7, and then I had to connect 5 all-pass filters in series, with loop delays of 100ms, 68ms, 60ms, 19.7ms, and 5.85ms and gains of +0.7, -0.7, +0.7, +0.7 and +0.7. In the end, this reverberation works very well on the python simulation. On figure 10, I plotted the impulse response corresponding to this delay network.

This algorithm was then implemented on the F413ZH. It was not easy to deal

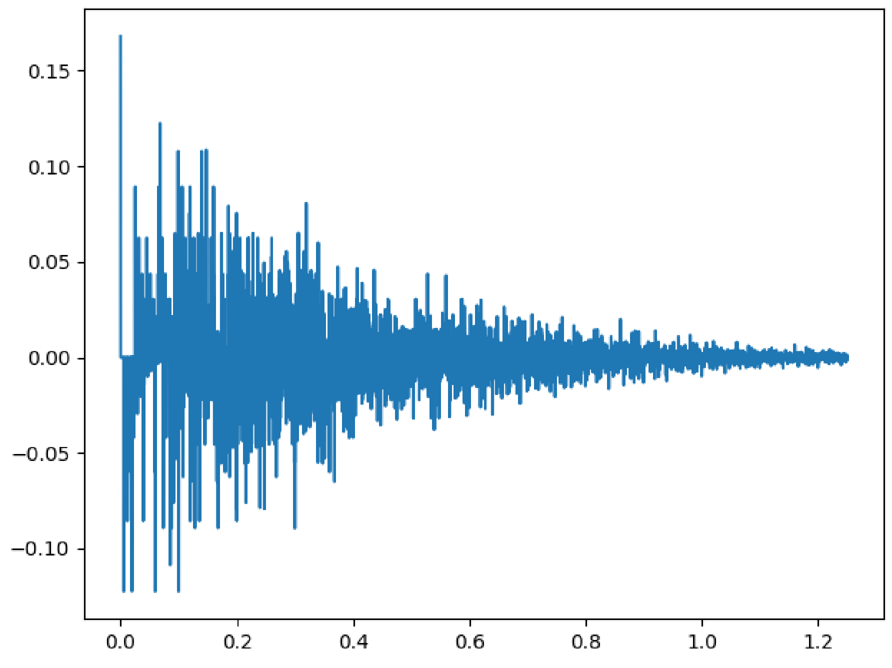


Figure 10: Impulse response corresponding to 5 allpass filters in series

with the different buffers to have several filters in series. The solution I found is displayed on figures 12(c) and 12(d). At each step, a new half buffer is filled in `dma_in`. It gets processed, and is then copied to an output buffer with global scope. This output buffer is used as input to the next processing function.

However, due to the speed of the microcontroller, I could only have up to 4 allpass filters in series. This already creates a quite realistic reverberation.

I also implemented the Schroeder reverberator described above, with the parameters Schroeder suggests in [9]: $\tau_1 = 30ms, \tau_2 = 37ms, \tau_3 = 42ms, \tau_4 = 45ms, \tau_5 = 5ms, \tau_6 = 1.7ms$, $g_5 = g_6 = 0.7$ and $g_n = e^{\frac{-3\tau_n}{T}}$ for $n = 1, 2, 3, 4$, with T the wanted reverberation time (I chose 1s). This reverberator was implemented on python, and the resulting impulse response is displayed on figure 11. The Schroeder reverberator was also implemented on the F413ZH, but once again I was a little bit short in computation power, and I could not do the last all pass filter. Comparing with the allpass filter reverberation, I think it sounds a little bit better, and less flat.

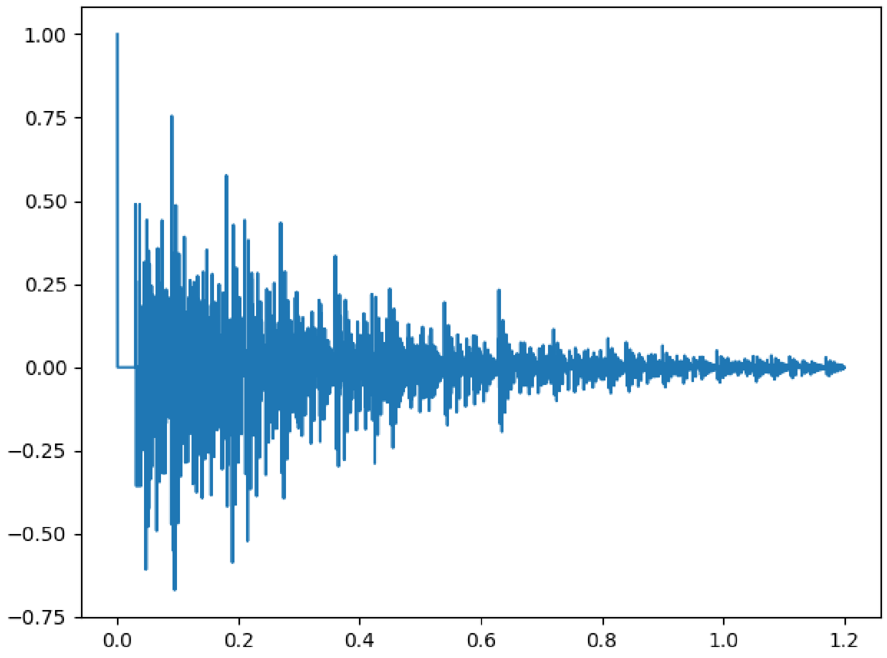
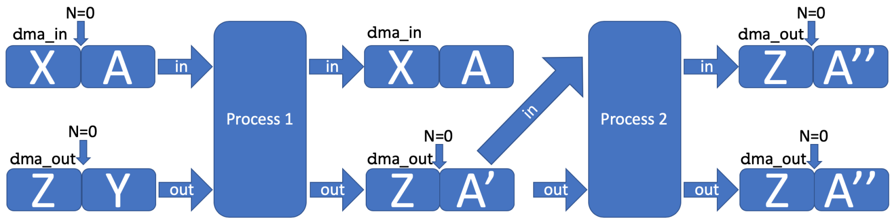
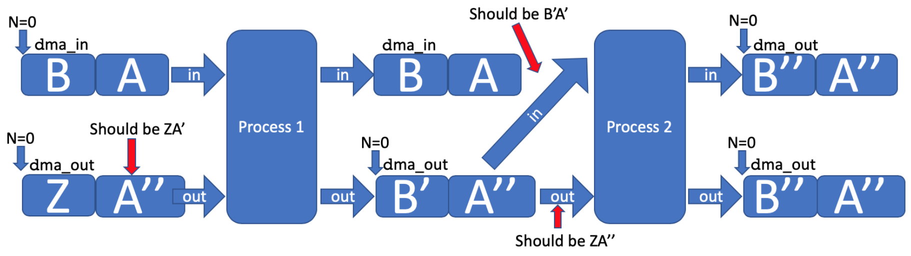


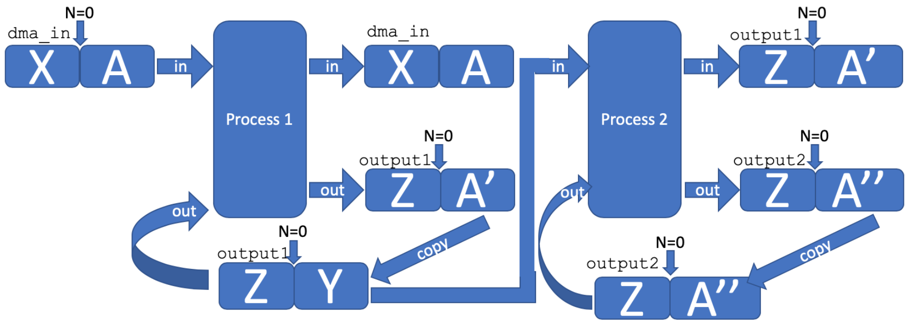
Figure 11: Impulse response corresponding to the Schroeder reverberator



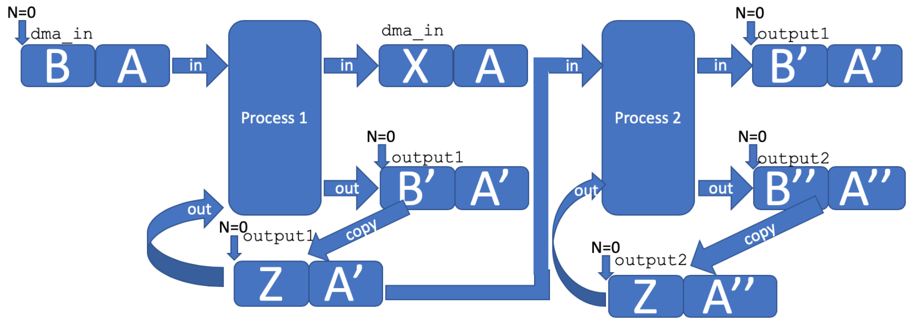
(a) First processing step of initial algorithm



(b) Second processing step of initial algorithm



(c) First processing step of solution algorithm



(d) Second processing step of solution algorithm

Figure 12: Buffer filling problem and solution

Here is the code for 4 allpass filters in series (everytime a half buffer is full, process, process2, process3 and process4 are called). The variables buffer, buffer2, buffer3 and buffer4 are global:

```
void inline Process(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp) {

    static int16_t x_prev = 0;
    static int16_t ynd=0;
    static int16_t xnd=0;
    static int16_t y=0;

    for (uint16_t j = 0; j < size; j += 2) {
        int16_t j2 = j/2;

        y = (int16_t)(*(pIn+j) - x_prev);
        x_prev = *(pIn+j);

        if(!halfcomp){
            ynd = buffer[j2-M_DELAY1/2+QUARTER_BUFFER_SIZE];
            xnd = *(pIn+j-M_DELAY1);
            y = xnd - y*GAIN1 + ynd*GAIN1;
            buffer[j2+QUARTER_BUFFER_SIZE] = y;
        }
        else
        {

            if (j >= M_DELAY1){
                ynd = buffer[j2-M_DELAY1/2];
                xnd = *(pIn+j-M_DELAY1);
                y = xnd - y*GAIN1 + ynd*GAIN1;
                buffer[j2] = y;
            }

            else{
                ynd = buffer[j2-M_DELAY1/2+HALF_BUFFER_SIZE];
                xnd = *(pIn+j-M_DELAY1+FULL_BUFFER_SIZE);
                y = xnd - y*GAIN1 + ynd*GAIN1;
                buffer[j2] = y;
            }
        }
    }
}

void inline Process2(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){

    static int16_t ynd=0;
    static int16_t xnd=0;
    static int16_t y=0;

    for (uint16_t j = 0; j < size; j += 2) {
        int16_t j2 = j/2;

        if(!halfcomp){
            y = buffer[j2+ QUARTER_BUFFER_SIZE];

            ynd = buffer2[j2-M_DELAY2/2+QUARTER_BUFFER_SIZE];
            xnd = buffer[j2-M_DELAY2/2+QUARTER_BUFFER_SIZE];
            y = xnd - y*GAIN2 + ynd*GAIN2;
            buffer2[j2+QUARTER_BUFFER_SIZE] = y;
        }
        else
        {
            y = buffer[j2];

            if (j >= M_DELAY2){
                ynd = buffer2[j2-M_DELAY2/2];
                xnd = buffer[j2-M_DELAY2/2];
                y = xnd - y*GAIN2 + ynd*GAIN2;
                buffer2[j2] = y;
            }

            else{
                ynd = buffer2[j2-M_DELAY2/2+HALF_BUFFER_SIZE];
                xnd = buffer[j2-M_DELAY2/2+HALF_BUFFER_SIZE];
                y = xnd - y*GAIN2 + ynd*GAIN2;
                buffer2[j2] = y;
            }
        }
    }
}

void inline Process3(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){

    static int16_t ynd=0;
    static int16_t xnd=0;
    static int16_t y=0;

    for (uint16_t j = 0; j < size; j += 2) {

        int16_t j2 = j/2;

        if(!halfcomp){
            y = buffer2[j2+ QUARTER_BUFFER_SIZE];
            ynd = buffer3[j2-M_DELAY3/2+QUARTER_BUFFER_SIZE];
            xnd = buffer2[j2-M_DELAY3/2+QUARTER_BUFFER_SIZE];
            y = xnd - y*GAIN3 + ynd*GAIN3;
            buffer3[j2+QUARTER_BUFFER_SIZE] = y;
        }
        else
        {
            y = buffer2[j2];

            if (j >= M_DELAY3){
                ynd = buffer3[j2-M_DELAY3/2];
                xnd = buffer2[j2-M_DELAY3/2];
                y = xnd - y*GAIN3 + ynd*GAIN3;
                buffer3[j2] = y;
            }

            else{
                ynd = buffer3[j2-M_DELAY3/2+HALF_BUFFER_SIZE];
                xnd = buffer2[j2-M_DELAY3/2+HALF_BUFFER_SIZE];
                y = xnd - y*GAIN3 + ynd*GAIN3;
                buffer3[j2] = y;
            }
        }
    }
}
void inline Process4(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){

    static int16_t ynd=0;
    static int16_t xnd=0;
    static float y=0;

    for (uint16_t j = 0; j < size; j += 2) {

        int16_t j2 = j/2;

        if(!halfcomp){
            y = buffer3[j2+ QUARTER_BUFFER_SIZE];

            ynd = buffer4[j2-M_DELAY4/2+QUARTER_BUFFER_SIZE];
            xnd = buffer3[j2-M_DELAY4/2+QUARTER_BUFFER_SIZE];
            y = xnd - y*GAIN4 + ynd*GAIN4;
            int16_t y16 = (y/maxnum) *0xFFF;
            buffer4[j2+QUARTER_BUFFER_SIZE] = y16;
        }
        else
        {
            y = buffer3[j2];

            if (j >= M_DELAY4){
                ynd = buffer4[j2-M_DELAY4/2];
                xnd = buffer3[j2-M_DELAY4/2];
                y = xnd - y*GAIN4 + ynd*GAIN4;
                int16_t y16 = (y/maxnum) *0xFFF;
                buffer4[j2] = y16;
            }

            else{
                ynd = buffer4[j2-M_DELAY4/2+HALF_BUFFER_SIZE];
                xnd = buffer3[j2-M_DELAY4/2+HALF_BUFFER_SIZE];
                y = xnd - y*GAIN4 + ynd*GAIN4;
                int16_t y16 = (y/maxnum) *0xFFF;
                buffer4[j2] = y16;
            }
        }
    }

    if(!halfcomp){
        for (uint16_t k = 0; k < size/2; k += 1) {
            *(pOut+2*k) = buffer4[k+QUARTER_BUFFER_SIZE];
            *(pOut+2*k+1) = buffer4[k+QUARTER_BUFFER_SIZE];
        }
    }
    else{
        for (uint16_t k = 0; k < size/2; k += 1) {
            *(pOut+2*k) = buffer4[k];
            *(pOut+2*k+1) = buffer4[k];
        }
    }
}
```

This is the code for Schroeder reverberator, in which we call process, process3 and process4 at every half buffer. The variables buffer, buffer3 and buffer4 are global:

```
void inline Process(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){

    static int16_t x_prev = 0;
    static int16_t ynd1=0;
    static int16_t xnd1=0;
    static int16_t ynd2=0;
    static int16_t xnd2=0;
    static int16_t ynd3=0;
    static int16_t xnd3=0;
    static int16_t ynd4=0;
    static int16_t xnd4=0;

    static int16_t buffercomb1[HALF_BUFFER_SIZE];
    static int16_t buffercomb2[HALF_BUFFER_SIZE];
    static int16_t buffercomb3[HALF_BUFFER_SIZE];
    static int16_t buffercomb4[HALF_BUFFER_SIZE];

    static float y=0;

    static int16_t y16 =0;
    for (uint16_t j = 0; j < size; j += 2) {
        int16_t j2 = j/2;

        y = (int16_t)(*(pIn+j) - x_prev);
        x_prev = *(pIn+j);

        if(!halfcomp){
            ynd1 = buffercomb1[j2-M_DELAY_COMB1/2+QUARTER_BUFFER_SIZE];
            xnd1 = *(pIn+j-M_DELAY_COMB1);
            buffercomb1[j2+QUARTER_BUFFER_SIZE] = xnd1 + ynd1*GAIN1;

            ynd2 = buffercomb2[j2-M_DELAY_COMB2/2+QUARTER_BUFFER_SIZE];
            xnd2 = *(pIn+j-M_DELAY_COMB2);
            buffercomb2[j2+QUARTER_BUFFER_SIZE] = xnd2 + ynd2*GAIN2;

            ynd3 = buffercomb3[j2-M_DELAY_COMB3/2+QUARTER_BUFFER_SIZE];
            xnd3 = *(pIn+j-M_DELAY_COMB3);
            buffercomb3[j2+QUARTER_BUFFER_SIZE] = xnd3 + ynd3*GAIN3;

            ynd4 = buffercomb4[j2-M_DELAY_COMB4/2+QUARTER_BUFFER_SIZE];
            xnd4 = *(pIn+j-M_DELAY_COMB4);
            buffercomb4[j2+QUARTER_BUFFER_SIZE] = xnd4 + ynd4*GAIN4;

            y16 = buffercomb1[j2+QUARTER_BUFFER_SIZE]
            +buffercomb2[j2+QUARTER_BUFFER_SIZE]
            +buffercomb3[j2+QUARTER_BUFFER_SIZE]
            +buffercomb4[j2+QUARTER_BUFFER_SIZE];
            // faster than xnd1 + ynd1*GAIN1 + xnd2 + ynd2*GAIN2 +
            xnd3 + ynd3*GAINC3 + xnd4 + ynd4*GAINC4;
            buffer[j2+QUARTER_BUFFER_SIZE] = y16;
        }
        else
        {
```

```
if (j >= M_DELAY_COMB1){
    ynd1 = buffercomb1[j2-M_DELAY_COMB1/2];
    xnd1 = *(pIn+j-M_DELAY_COMB1);
    buffercomb1[j2] = xnd1 + ynd1*GAIN1;
}
else{
    ynd1 = buffercomb1[j2-M_DELAY_COMB1/2+HALF_BUFFER_SIZE];
    xnd1 = *(pIn+j-M_DELAY_COMB1+FULL_BUFFER_SIZE);
    buffercomb1[j2] = xnd1 + ynd1*GAIN1;
}


if (j >= M_DELAY_COMB2){
    ynd2 = buffercomb2[j2-M_DELAY_COMB2/2];
    xnd2 = *(pIn+j-M_DELAY_COMB2);
    buffercomb2[j2] = xnd2 + ynd2*GAIN2;
}
else{
    ynd2 = buffercomb2[j2-M_DELAY_COMB2/2+HALF_BUFFER_SIZE];
    xnd2 = *(pIn+j-M_DELAY_COMB2+FULL_BUFFER_SIZE);
    buffercomb2[j2] = xnd2 + ynd2*GAIN2;
}


if (j >= M_DELAY_COMB3){
    ynd3 = buffercomb3[j2-M_DELAY_COMB3/2];
    xnd3 = *(pIn+j-M_DELAY_COMB3);
    buffercomb3[j2] = xnd3 + ynd3*GAIN3;
}
else{
    ynd3 = buffercomb3[j2-M_DELAY_COMB3/2+HALF_BUFFER_SIZE];
    xnd3 = *(pIn+j-M_DELAY_COMB3+FULL_BUFFER_SIZE);
    buffercomb3[j2] = xnd3 + ynd3*GAIN3;
}

if (j >= M_DELAY_COMB4){
    ynd4 = buffercomb4[j2-M_DELAY_COMB4/2];
    xnd4 = *(pIn+j-M_DELAY_COMB4);
    buffercomb4[j2] = xnd4 + ynd4*GAIN4;
}
else{
    ynd4 = buffercomb4[j2-M_DELAY_COMB4/2+HALF_BUFFER_SIZE];
    xnd4 = *(pIn+j-M_DELAY_COMB4+FULL_BUFFER_SIZE);
    buffercomb4[j2] = xnd4 + ynd4*GAIN4;
}
```

```
            y16 = buffercomb1[j2] +buffercomb2[j2]+buffercomb3[j2]+buffercomb4[j2];
            buffer[j2] = y16;
        }
    }
}

void inline Process3(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){

    static int16_t ynd=0;
    static int16_t xnd=0;
    static int16_t y=0;

    for (uint16_t j = 0; j < size; j += 2) {
        int16_t j2 = j/2;

        if(!halfcomp){
            y = buffer[j2+ QUARTER_BUFFER_SIZE];
            ynd = buffer3[j2-M_DELAY3/2+QUARTER_BUFFER_SIZE];
            xnd = buffer[j2-M_DELAY3/2+QUARTER_BUFFER_SIZE];
            y = xnd - y*GAIN3 + ynd*GAIN3;
            buffer3[j2+QUARTER_BUFFER_SIZE] = y;
        }
        else
        {
            y = buffer[j2];

            if (j >= M_DELAY3){
                ynd = buffer3[j2-M_DELAY3/2];
                xnd = buffer[j2-M_DELAY3/2];
                y = xnd - y*GAIN3 + ynd*GAIN3;
                buffer3[j2] = y;
            }

            else{
                ynd = buffer3[j2-M_DELAY3/2+HALF_BUFFER_SIZE];
                xnd = buffer[j2-M_DELAY3/2+HALF_BUFFER_SIZE];
                y = xnd - y*GAIN3 + ynd*GAIN3;
                buffer3[j2] = y;
            }
        }
    }
}
void inline Process4(int16_t *pIn, int16_t *pOut, uint16_t size, bool halfcomp){
    static int16_t ynd=0;
    static int16_t xnd=0;
    static int16_t xn=0;
    static int16_t y16 = 0;
    static float y=0;

    for (uint16_t j = 0; j < size; j += 2) {
        int16_t j2 = j/2;

        if(!halfcomp){
            xn = buffer[j2+ QUARTER_BUFFER_SIZE];

            ynd = buffer4[j2-M_DELAY4/2+QUARTER_BUFFER_SIZE];
            xnd = buffer[j2-M_DELAY4/2+QUARTER_BUFFER_SIZE];
            y = xnd - xn*GAIN4 + ynd*GAIN4;
            int16_t y16 = (y/maxnum) *0xFFF;
            buffer4[j2+QUARTER_BUFFER_SIZE] = y16;
        }
        else
        {
            xn = buffer[j2];

            if (j >= M_DELAY4){
                ynd = buffer4[j2-M_DELAY4/2];
                xnd = buffer[j2-M_DELAY4/2];
                y = xnd - xn*GAIN4 + ynd*GAIN4;
                int16_t y16 = (y/maxnum) *0xFFF;
                buffer4[j2] = y16;
            }

            else{
                ynd = buffer4[j2-M_DELAY4/2+HALF_BUFFER_SIZE];
                xnd = buffer[j2-M_DELAY4/2+HALF_BUFFER_SIZE];
                y = xnd - xn*GAIN4 + ynd*GAIN4;
                int16_t y16 = (y/maxnum) *0xFFF;
                buffer4[j2] = y16;
            }
        }
    }

    if(!halfcomp){
        for (uint16_t k = 0; k < size/2; k += 1) {
            *(pOut+2*k) = buffer4[k+QUARTER_BUFFER_SIZE];
            *(pOut+2*k+1) = buffer4[k+QUARTER_BUFFER_SIZE];
        }
    }
    else{
        for (uint16_t k = 0; k < size/2; k += 1) {
            *(pOut+2*k) = buffer4[k];
```

```
                *(pOut+2*k+1) = buffer4[k];
            }
        }
}
```

6 Conclusion

In this project, I managed to create the saturation, delay, chorus and reverberation effects on different STM32 boards. It was very interesting to learn about those audio effects, as I knew what they sounded like but not much about how they were created. I was also happy to learn a bit more in details about reverberation created with feedback delay networks, as reverberation was the subject of my previous semester project but I could not really go into details of delay networks at the time. I also learnt a lot about coding in C, which I had seen in first year but had not practiced much after that, and about coding audio processing in python.
I was a bit surprised at how slow I was to code the effects in C, compared with the python simulations. The problem in low level languages is that very small mistakes can take a lot of time to debug, and sometimes you can get stuck a long time.
There was also quite a lot of hardware issues, since we ordered a H7 microcontroller in order to have more RAM for the delay, but in the end we could not get the H7 to work and had to switch to the F413. The right channel of the ADC/DAC board also broke, and I thought the problem was coming from my code.
At the end, I was happy to have an extra week to finish the code, and I am very happy to have been able to make each effect work on the microcontroller. I had expected to finish the effects processor quite fast, and eventually be able to make a small digital synthesizer. It would have been nice to be able to finish something like this, and could be a sort of continuation of this project.

References

[1] A. H. Eric Bezzam and P. Prandoni, "Digital audio effects tutorial."

[2] R. Elliot, "Soft clipping - useful or a waste of time?."

[3] U. Zolzer, *DAFX: Digital Audio Effects.*

[4] K. A. Paolo Prandoni, Frederike Dumbgen, *COM303 : Digital Signal Processing, lecture 19.*

[5] M. G. Christensen, *Introduction to Audio Processing.* Springer International Publishing, 2019.

[6] V. Valimaki, J. D. Parker, L. Savioja, J. O. Smith, and J. S. Abel, “Fifty years of artificial reverberation,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, pp. 1421–1448, 2012.

[7] E. A. C. G. A. Farina, “Implementation of real-time partitioned convolution on a dsp board,” in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, New Paltz, New York*, pp. 19–22, 2003.

[8] M. R. Schroeder and B. F. Logan, “” colorless” artificial reverberation,” *IRE Transactions on Audio*, no. 6, pp. 209–214, 1961.

[9] M. R. Schroeder, “Natural sounding artificial reverberation,” *Journal of the Audio Engineering Society*, vol. 10, no. 3, pp. 219–223, 1962.