View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTMENT OF ELECTRONIC ENGINEERING

FINAL PROJECT

MULTIEFFECTS PROCESSOR

REPORT

Student: Cristian Gil Morales

Supervised by: Albert Masip Alvarez
Manuel Lamich Arocas

December 2013

I want to thank to all the people who have encouraged me during the realization of this project.

I would like to thank my supervisors Albert Masip Alvarez and Manuel Lamich Alvarez for their guidance and support.

I would like to thank all my English teachers and friends who have corrected this project in English.

Finally I want to thank my family, friends and girlfriend, who always believed in me.

Thank you.

Index

Introduction

Programming

Conclusions

INTRODUCTION

Acronyms

There are many elements / components / programs / etc with very long names in the present document, and read them a lot of times it can be uncomfortable for the reader.

For readability, the following acronyms are used in replacement of these names. It is also shown their meaning.

Likewise, it sometimes recalls their meanings along the document to avoid losing the reading thread continuously.

PC: Personal Computer.
DSP: Digital Signal Processor.
RTDX: Real Time Data Exchange.
LFO: Low Frequency Oscillator.
AM: Amplitude Modulation.
DSK: DSP Starter Kit.
FFT: Fast Fourier Transform.
DFT: Discrete Fourier Transform.
ASIC: Application-Specific Integrated Circuit.
FPGA: Field-Programmable Gate Array.
TI: Texas Instruments.
DSK: DSP Starter Kit.
USB: Universal Serial Bus.
IDE: Integrated Development Environment.
VLIW: Very Long Instruction Word.
MFLOPS: Million of FLating-point Operations Per Second.
MIPS: Million of Instructions Per Second.
MMACS: Million of Multiplictions Per Second.
JTAG: Joint Test Action Group.
ADC: Analog-Digital Converter.
DAC: Digital Analog Converter.
FS: Sampling Frequency.
FM: Maximum Frequency.
McBSP: Multichannel Buffered Serial Port.
EDMA: Enhanced Direct Memory Access.
GUI: Graphical User Interface.
CCStudio: Code Composer Studio.
VC++: Microsoft Visual C++.
API: Application Programming Interface.
SDK: Software Development Kit.
MFC: Microsoft Foundation Classes.
SDI: Single Document Interface.
MDI: Multiple Document Interface.

Introduction

Music is transmitted increasingly among the population. This does that the people have more contact with the audio waves and its treatment.
Nowadays it is normal that a lot of people play musical instruments. Hence they end up having the need to treat the audio waves which they generate for give another sense or personality to their songs, transmitting new and unique emotions.

Throughout the history, the search of new sounds has not ended. The musicians, in their attempt to innovate and achieve new styles and sounds to reach their personal mark, have always used all of their ingenuity in anywhere.
To create a new effect, firstly it has to find the desired effect. After that, it has to make an artifact which allows to generate this effect, using purely analog techniques.
Examples are the characteristic distortion sound of the valve amps, the Fuzz pedal from Jimi Hendrix, the Delay effect discovered accidentally by Ritchie Blackmore, the Flanger effect used by The Beatles, etc.

New technology throughout the years has done possible to do digital signal processing in an economic way, but mainly easier.

The present project of the multieffect pedal is born to the need to investigate in the world of audio processing in a more professional level for who seek new sounds with their musical instrument.

This multieffect pedal generates the most used sound effects in the music world in real time with the input signal.

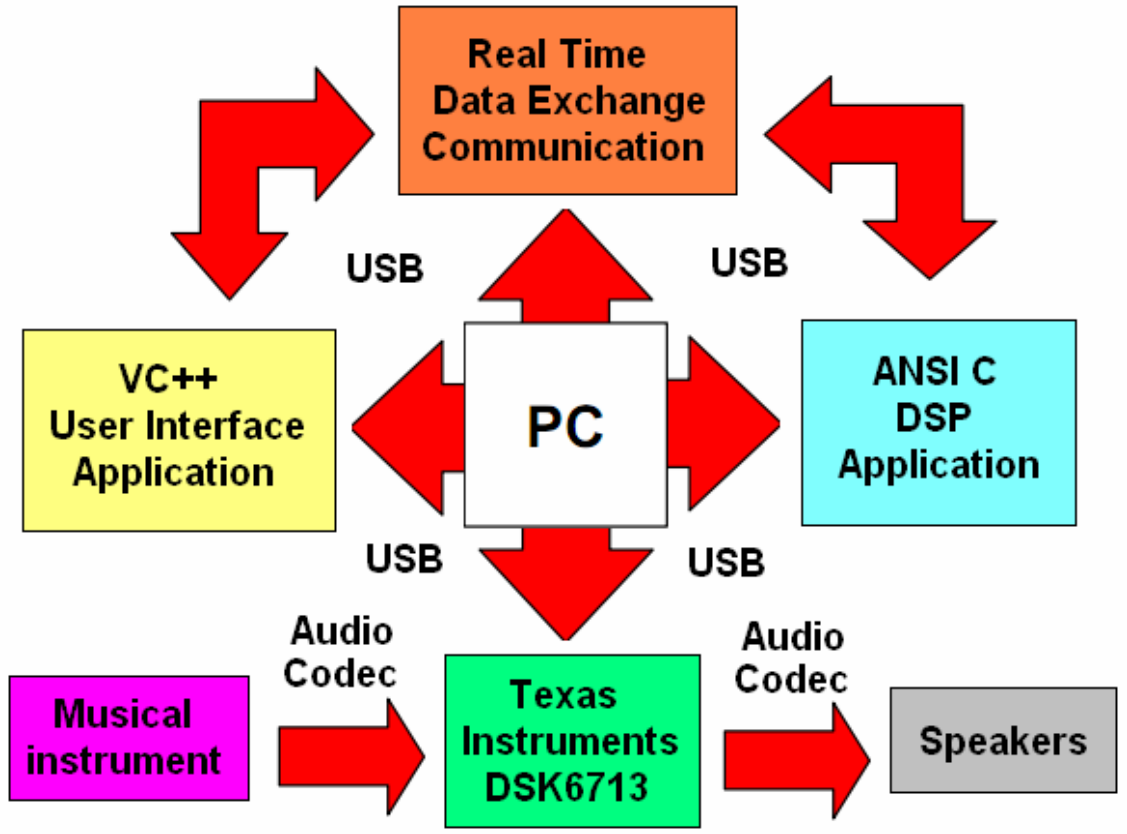
Elements like Digital Signal Processors (DSPs), appeared in the eighties, are systems based on processors or microprocessors. They have a group of optimized instructions to apps that require a lot of numeric operations in very high velocity in front of the big and expensive analog circuits.
Due to this reason, the DSPs are especially useful for the processing and representation of signals in real time. Now, complex operations like signal filtering or the Fourier transform, are easy to implement in a simple work file.

In addition to this processing in real time, it is necessary an interface which is the bridge between the user and the DSP. It allows to change among all the effects and change their configuration in a easy and simple manner.
Therefore the communication between both units is constant and uninterrupted.

The coordination between the DSP and the interface is carried out by the communication protocol Real Time Data Exchange (RTDX). Its libraries are included in the programming software of the DSP (which are distributed with this last one).

This is useful to obtain a powerful homemade multieffect processor with a decent quality, acceptable for any user.

Onwards it explains the hardware TMS320C6713 DSP, its peripherals, its programming code and the interface programmed in Visual C++ that controls it.


Real Time
Data Exchange
Communication
USB
USB
VC++
User Interface
Application
PC
ANSI C
DSP
Application
USB
USB
Audio
Codec
Audio
Codec
Musical
instrument
Texas
Instruments
DSK6713
Speakers

Motivations and objectives

Motivations

I proposed to do this project because two years ago I became interested in music in a deeper sense and I also play the electric guitar regularly on my own.

Without option to go to a music school, it is complicated the fact of learning the secrets of this INFINITE world, especially if I am working or busy with other issues.

Recently I bought an amplifier which incorporates various sound effects, but I do not know what the differences between them are, how to set them, the best way to connect them, etc.
I would have to find out more about the issue, but I do not have the necessary time to enter into this whole wide world.
Moreover, after discussing this topic with my professor, I decided to do this project to initiate me into this world definitely.

Objectives

Initially the project was thought to have a finished product and use it at home with my musical instrument and amplifier. This required a hardware interface to control it with the feet, so it would be possible playing guitar while it changes/configures the effects, like in the professional world.

Therefore, the objectives to be achieved in this project are:

- Know the audio effects and the audio treatment in a deeper way.
- Developing an application using a DSP microprocessor.
- Receiving audio with a codec from outside (by instrument, PC, iPod...).
- Implement a group of algorithms to use them in real time.
- Sending the audio result to speakers with the codec.
- Managing the algorithm variables through an external interface.

After this project, I hope to have much knowledge on this subject and hence be able to apply it in my life as a musician.

Audio effects

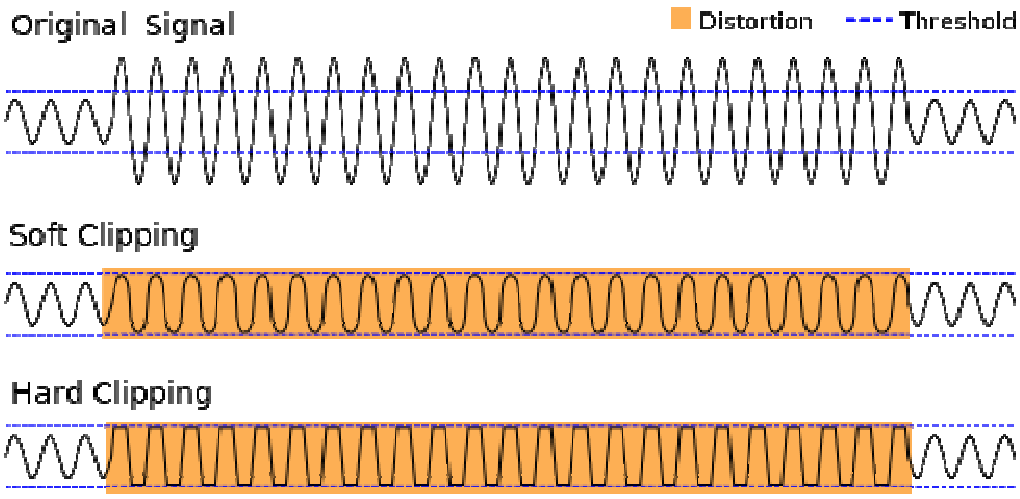
For this project it creates a total of 13 effects classified into 4 groups according to the method used to create them:

- **Gain effects:** Distortion of the original signal with gain variation.
  - **Overdrive:** Low distortion.
  - **Distortion:** Medium distortion.
  - **Fuzz:** High distortion.
- **Dynamic Range Effects:** Alters the signal using filters and thresholds.
  - **Compressor:** Compress the dynamic range of the signal.
  - **Expander:** Expands the dynamic range of the signal.
  - **Noise Gate:** Eliminates the signal range under a threshold.
  - **AutoWah:** Applies a dynamic band pass filter.
  - **Panning:** Swing of the signal between speakers (stereo spekaers are needed).
- **Modulation Effects:** Alters the signal with a Low Frequency Oscillator (LFO).
  - **Chorus:** Simulates two musicians playing an instrument in unison.
  - **Flanger:** Produces a swept comb filter effect.
  - **Tremolo:** Oscillates the output volume.
- **Repetition effects:** Simulation of the environment with repeated signals.
  - **Delay:** Original signal plus delayed signal.
  - **Reverb:** Simulates the acoustics of a room.

Gain effects

The gain effects create "warm", "gritty" and "fuzzy" sounds by "clipping" an instrument's audio signal, which distorts the shape of its wave form and adds overtones.

It affects to the gain level of sound. Depending on the characteristics, it succeeds altering various harmonics and it dulls the fundamental note in different levels.

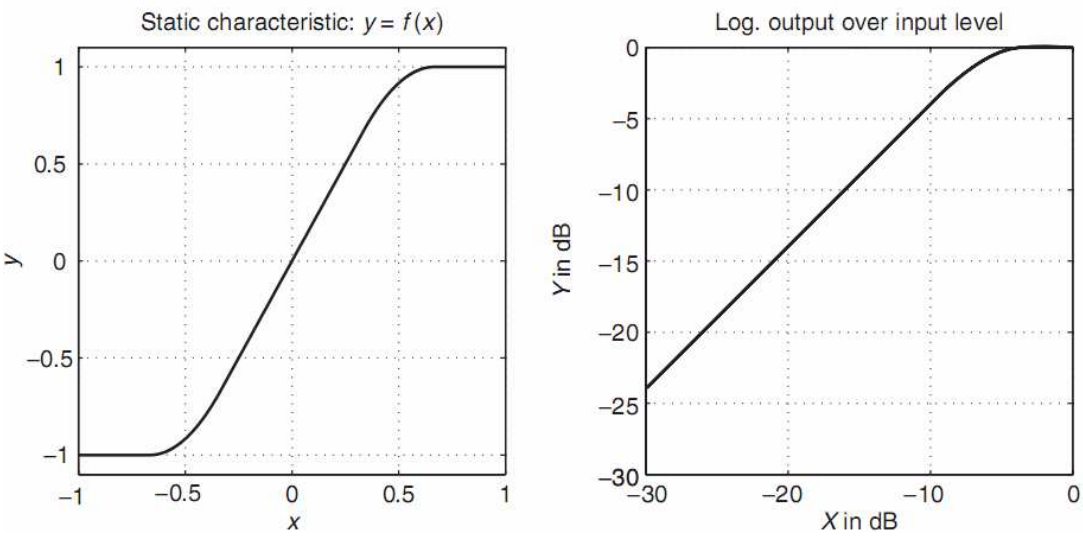


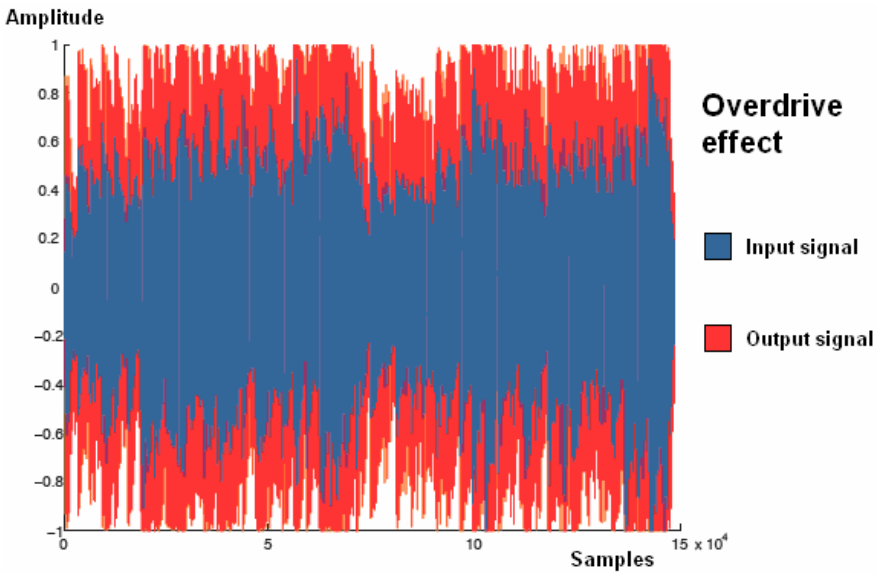
"Clipping" an instrument's audio signal produces distortion

Overdrive

The overdrive effect appears when the optimum threshold of the input signal is exceeded.
The signal is saturated with a low distortion and it produces natural harmonics, which enhance the sound body.

It affects mainly to the second harmonic in a moderate way, resulting irregular peaks in the signal.
The sound is more "dirty", but the main note is distinguishable and the saturation is not much heavy. Hence the natural sound remains.



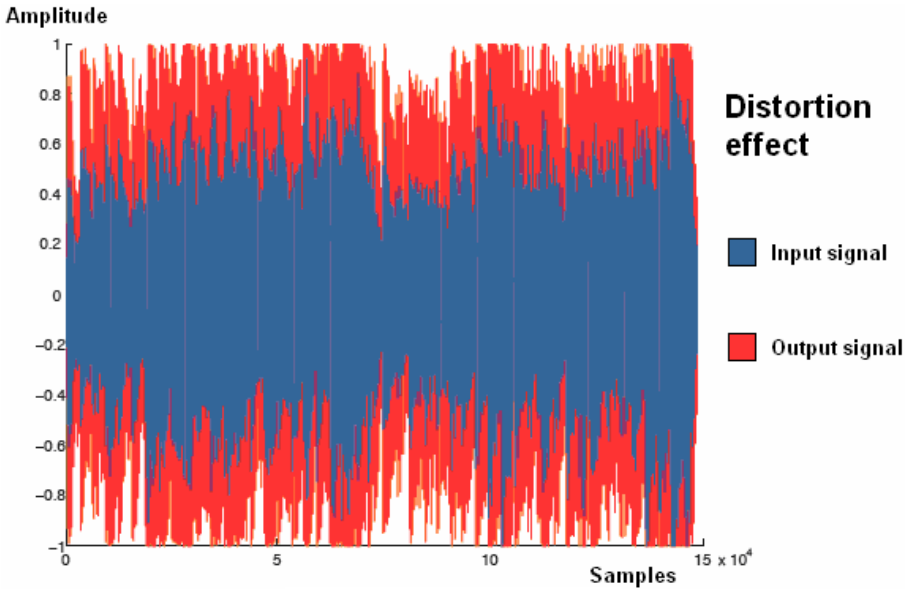


Example with a +-1 Volt signal

Distortion

The second and third harmonic are enhanced. The signal is highly saturated, the main note is indistinguishable and the wave is totally irregular. It highlights the medium-high tones, but with more distortion, and there are notable lows too.
However it is pleasant to hear these enhanced peaks, although the sound is theoretically dirty.

The distortion effect operates along a wider tonal area than it does the overdrive effect.

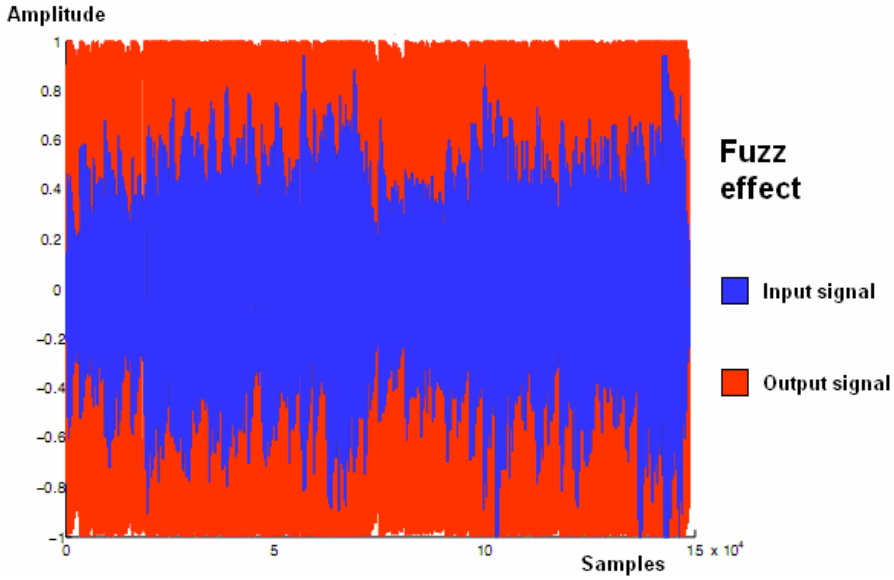


Example with a +-1 Volt signal

Fuzz

The second harmonic is enhanced in a highly way, and the signal peaks are generated in more quantity than the overdrive effect. The sound is focused on lows and middle tones, but not in treble tones.

The sound is heavier, with little clarity and it has a completely non-linear behaviour.



Example with a +-1 Volt signal

Dynamic range effects

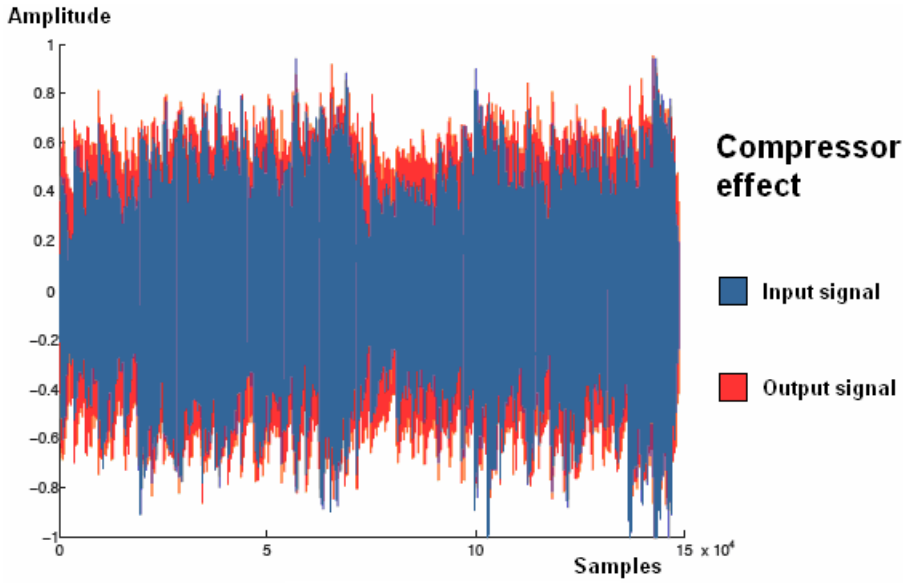
The dynamic range of an audio signal is the range between the softest and loudest parts of the signal. These effects are done to adjust the dynamic range of audio input signal. This is due to an increase the perceived loudness and to highlight of the main parts of the sound. At the same time, it ensures that the softer sounds are not lost in the mix.

These effects alter the frequency content of an audio signal by boosting or weakening specific frequencies or frequency regions.

Compressor

The signal compression is a process which modifies the audio signal to level all its amplitude throughout the signal. The signal is increased or decreased according to a math equation.

The difference between the most and the least extensive parts of the signals with an applied compression is reduced, as consequence the sound volume is equalised.
Regarding the threshold, the signal is reduced heavily. It does not include the higher volume in the low parts, namely, this effect does not amplify the volume. It is only reduced.



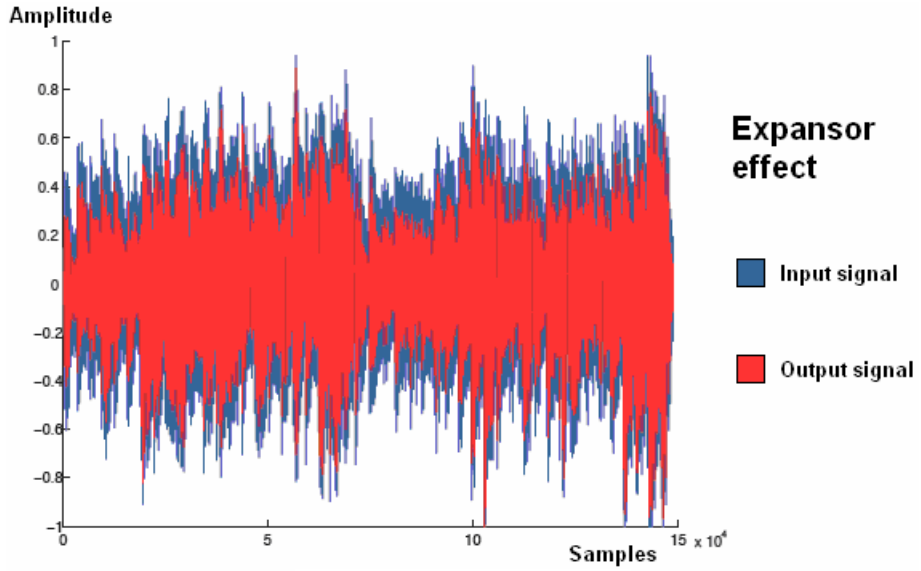
Example with a +-1 Volt signal

The use of the compression is necessary for professional recorders.

Expander

The expansor is a dynamic process which is used like a professional recording filter. It does the inverse process of a compressor, since it increases the dynamic range of the audio signal.

It has several control parameters. The threshold determines the level at which the expander starts working. The attack time to start the effect when the signal passes below the threshold.
The relaxation time or decay stops the effect when the signal returns to be above the threshold. And the last one is the expansion ratio which indicates the expansion level when the signal exceeds the threshold, for example a ratio of 1:2, 1:4, etc.



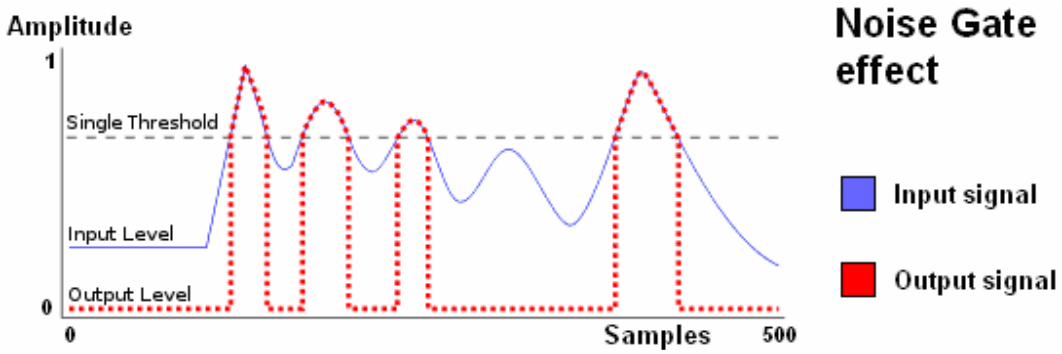
Example with a +-1 Volt signal

It is very used to reduce the background noise which is not wanted in the professional recordings.

Noise Gate

A noise gate can be considered as an extreme compressor with a infinite slope (in fact, the relation 1:10 is enough).
This effect consists in the complete muting of the signal below the defined threshold. The noise gate is typically used to eliminate the noise by setting the threshold just above the level of the background noise. So the signal only passes when its level is above the predefined threshold.

This results in an overall cleaner sound.

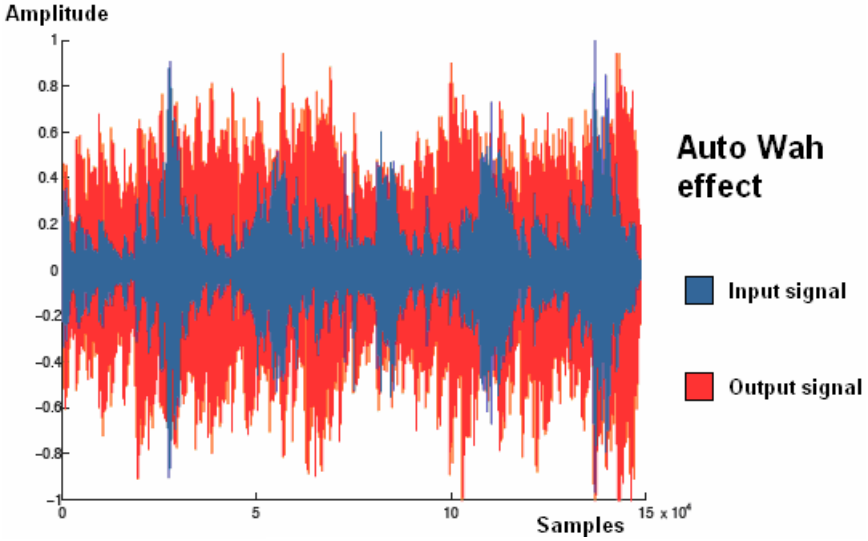


Example with a 1 Volt signal

Auto Wah

The wah-wah effect alters the tone of the signal to create a distinctive effect, mimicking the human voice. The effect sweeps the peak response to a band pass filter to create the sound.

The auto-wah effect is the wah-wah effect with a cosine signal which oscillates the cut frequency for the band pass filter.

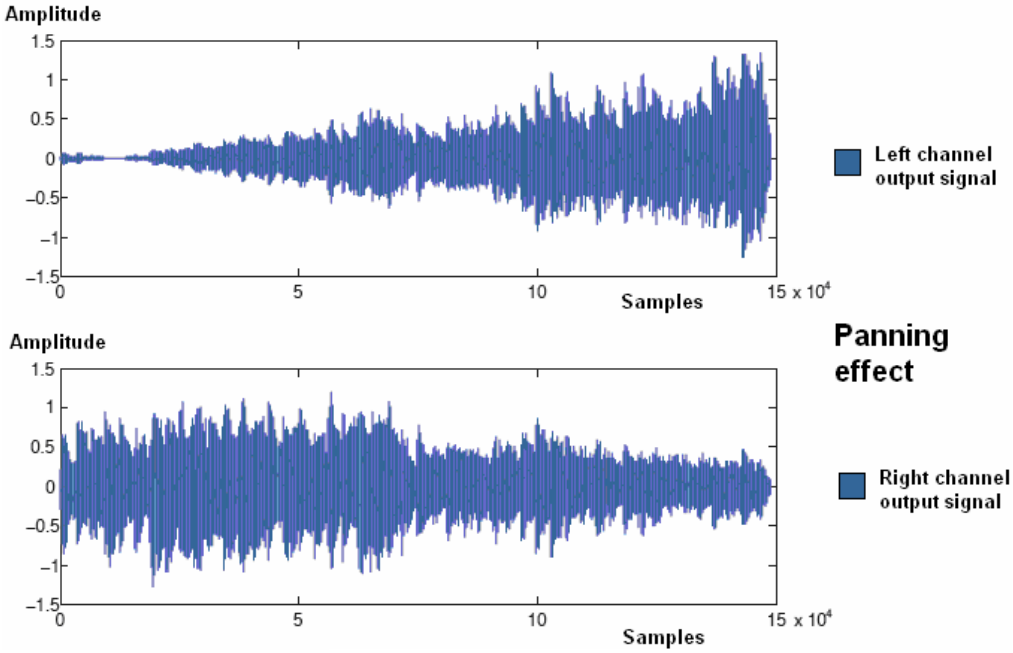


Example with a +-1 Volt signal

Panning

The panning effect is the spread of the signal (either monaural or stereophonic) into a new stereo or multi-channel sound field.
It is used to create the impression that the audio source is moving from one side of the soundstage to the another.

For the stereo output, one channel is multiplied by a cosine signal and another channel is multiplied by a sine signal, both with the same frequency and amplitude.



Example with a +-1.5 Volt signal

Modulation effects

The modulation effects are used to add motion and depth to the sound. They typically delay the input signal a few milliseconds and use a LFO to modulate the delayed signal.

A low frequency oscillator (LFO) is an artefact which generates a wave with a frequency which can be modified in low levels. These levels are so low than they cannot be heard, therefore they are only used for modulation purposes. The LFO parameters include speed (or frequency) and depth (or intensity) controls.
The LFO may also be used to modulate the delay time in some effects.

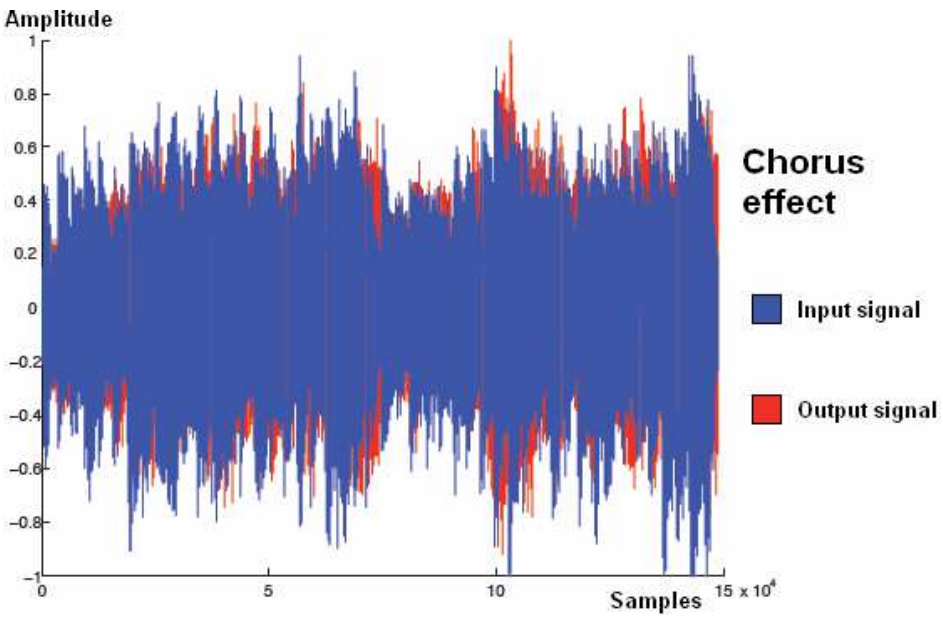
The original signal is often called the "dry" signal and the processed signal is called the "wet" signal.
Some modulation effects include feedback parameters, which add part of the effect output back into the input.

Chorus

The Chorus effect is qualitatively similar to the Flanger effect. It simulates the effect of several sound sources producing nearly the same sound, like a choir does with multiple singers in unison.

Electronically, it is achieved using small random variations of the time delay and it uses several delay channels which are recombined in stereo to produce a very rich sound.

Modulation rates are longer than the Flanger effect, typically 0.1-0.5 Hz, with similar delay times of 1 to 50 milliseconds. Unlike Flanger effect, the Chorus effect often employs amplitude modulation to simulate the way that the singers vary the volume in time.

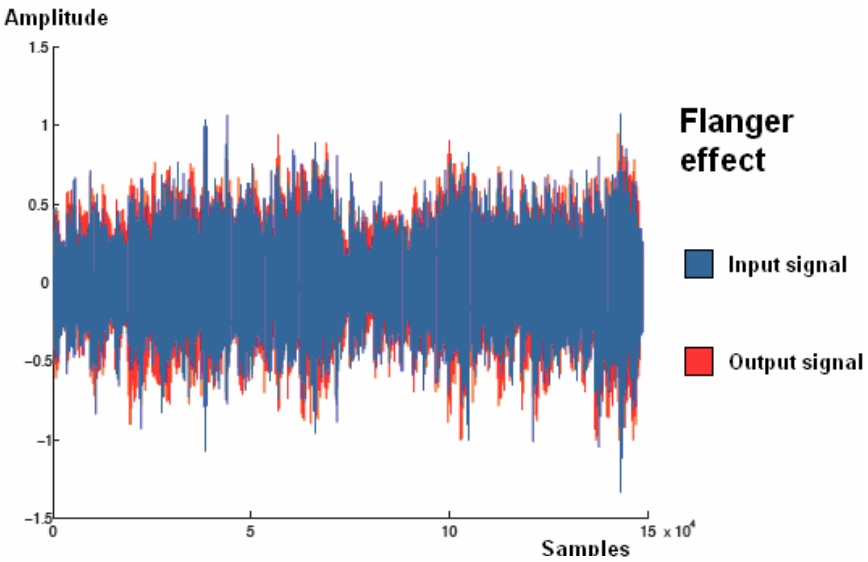


Example with a +-1 Volt signal

Flanger

When time-delayed and direct signals are combined, a comb-filter effect is created. Generally, this is undesirable. However, the effect can be used to "spice-up" certain sounds. If the delay time is constantly slightly altered, a rich sweeping filter is created. This is known as flanging.

In addition, the depth of the effect can be controlled through changing the balance between the delayed and direct signals. For flanging, the delay time is in the range of 5-35 milliseconds.
The modulation rate (which changes the delay time) is in the range of 1-10 Hz.

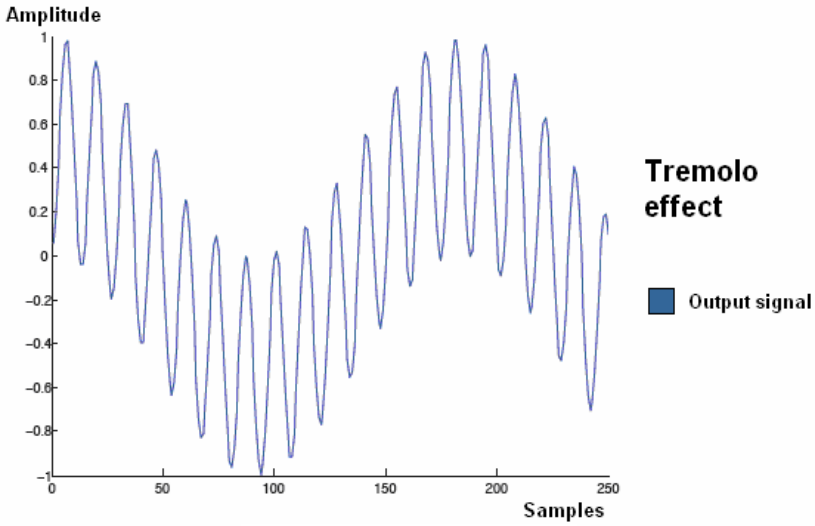


Example with a +-1.5 Volt signal

Tremolo

The tremolo effect changes the amplitude of the signal.
To obtain this effect, it multiplies the input signal by a periodic signal, usually a sinusoidal signal (with a LFO). In terms of modulation, this is the amplitude modulation (AM).

It also causes small phase changes which primarily affect low waves. Sometimes it is confused with the vibrato effect. The vibrato effect alters the frequency in function of time, not the volume.



Example with a +-1 Volt signal

Repetition effects

The most effects which are used in the modern music production are time-based.

These ones are achieved by mixing a original signal with a delayed copy of itself. Unlike the dynamics range and modulation effects (they are often used to enhance recorded sound without being obvious to the listener), the repetition effects are used to creatively alter the sound of the source.

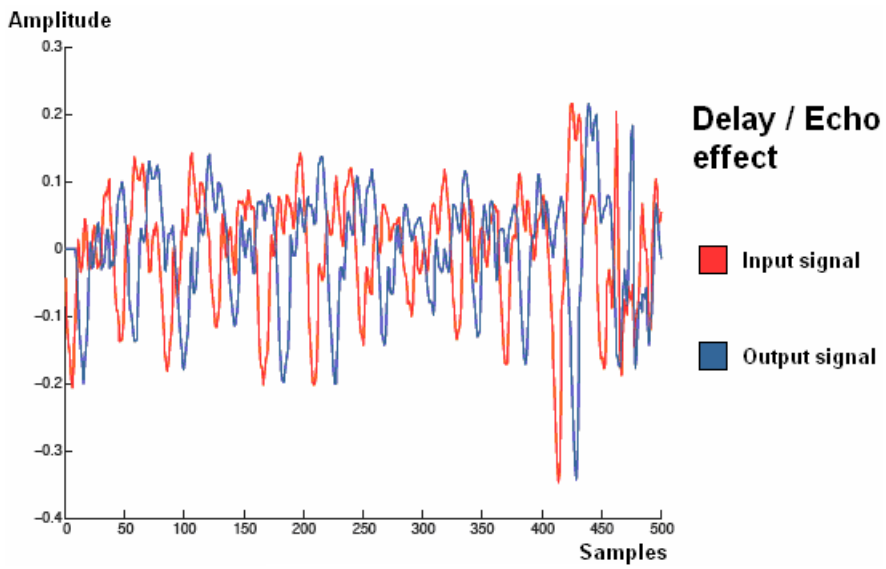
It can create many effects by delaying the input signal in variable amounts. In order to hear the delay, the delayed signal needs to be combined with the original signal.
For improve these effects, it is important to control the mix balance between the two signals.

It can also control the ratio of the modified signal (wet) and the original signal (dry).

Delay/Echo

The Delay effect records an input signal to an audio storage medium, and then it reproduces back after a period of time. The delayed signal can be played back multiple times to create the sound of a repeating.
After that, the processed signal is mixed with the original one.

The difference between the Delay and Echo effects is established simply by the amount of delay of the input signal and its repetition.
Therefore, in the Echo effect, the output signal (wet and dry signals together) are perceived by the ear like a new signal respect the original signal, since the delay is small. But with the Delay effect, the delay is very long, and the ear can distinguish the mix of both signals.

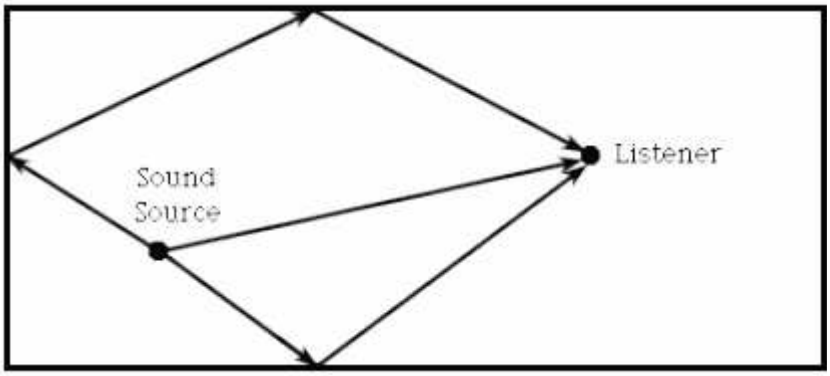


Example with a +-0.3 Volt signal

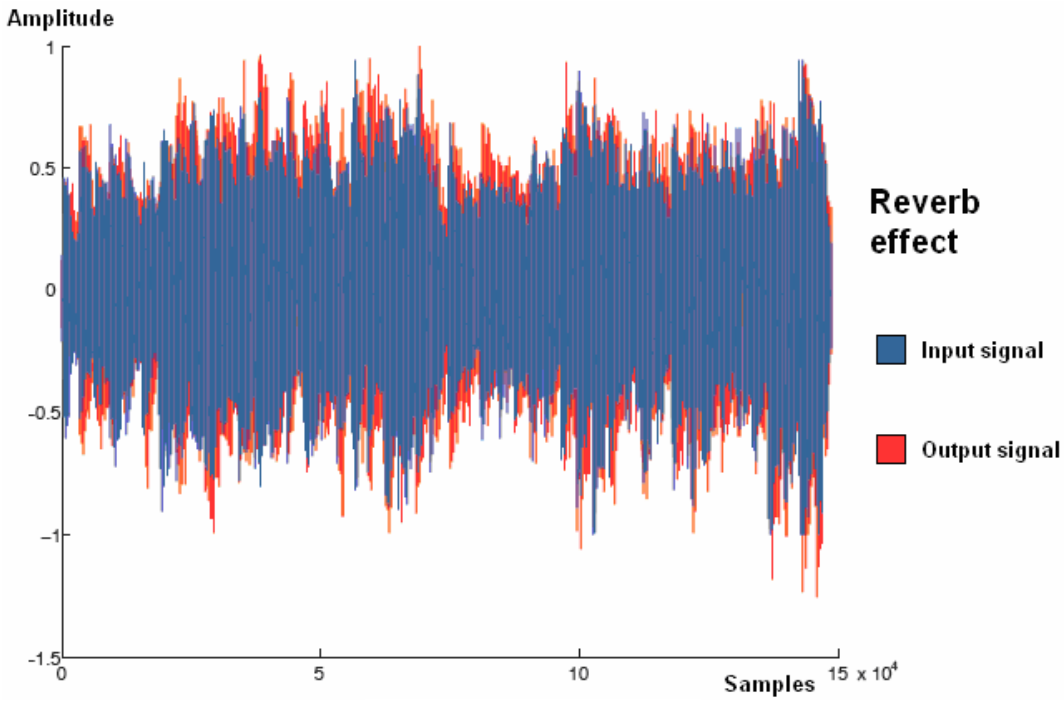
Reverb

Inside a room, it receives the sound through two channels: the direct sound and the reflected sound.

The reverb is a phenomenon produced by the reflection. The reflection consists in a permanence of the sound even when the original wave is gone.



This effect is more notable in big rooms with little absorbency (without curtains or any flexible object). Consequently, this effect is less notable in small rooms with a lot of absorbency (with curtains and other flexible objects).
This is part of the light system.



Example with a +-1 Volt signal

The effects order

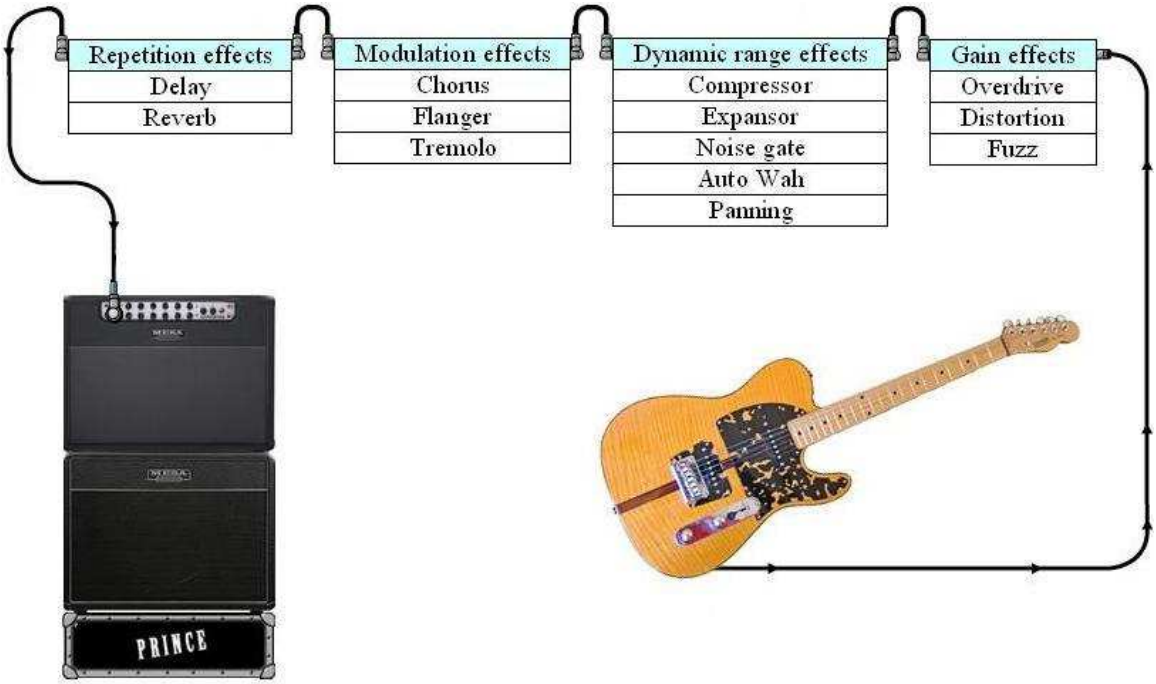
This project has the possibility to use multiple effects simultaneously. It can use at most one effect for each group together, namely, between zero or four effects at the same time.
Mix two effects of the same group is not sense because they could generate a lot of unnecessary noise.

Based on how these effects are generated, it is important to generate some of them before than other ones to achieve a minimum background noise and avoid unnecessary distortions.

This does not mean that it is not possible use the effects with another different order. Actually there are musicians who try to use other strange orders to find a special sound.
But the order proposed here is the most common and it helps to eliminate the background noise and distinguish the different used effects in the final result.

The correct order of the effects proposed is exactly the same order which these effects are been described: first of all the gain effects, after the dynamic range effects, then the modulation effects and finally the repetition effects.

In the next picture it can see better the proposed order:



For example, if the delay effect is generated first and then the distortion effect, the second effect distorts the first one. When the delay is distorted, it is not possible to distinguish the delay effect very well.
Conversely, if the distortion effect is applied first and then the delay effect, the result distorts the input signal. And once the input signal is distorted, then it delays.
Theoretically this one is the optimal result.

This example can be applied to any possible configuration among the different effects.

Retrospective: The history of the sound effects

A brief view in the history

As it can read below, in parallel and with few distance of time, there were several precursors who began conducting the first trials in searching of a different or particular sound.

The first amplified guitars appeared during the Swing era in the early 1930s.
At that time, the bands with many members and usually with wind instruments, dominated the show. Naturally, the guitarists wanted to grab some of those solos for themselves. But the natural sound of early amplified guitars was thin, reedy, thoroughly, anticlimactic and it did not quite be in the environment of the orchestras of the time.
It is not surprise that guitarists quickly began looking for ways to pump up their sound.

The very first guitar effects were built into instruments themselves.

In the 1930s, Rickenbacker made a clunky Vibrola Spanish guitar with motorized pulleys that jiggled the bridge to create a vibrato effect.

In the 1940s, DeArmond manufactured the world's first standalone effect, a type of tremolo.
Many guitarists looked for a way to reproduce the natural reverb and echo who they enjoyed during soundchecks in empty halls.
Although it is funny, the first effect of this kind was achieved when the guitarist Duane Eddy outfitted a 500 liters metal water tank with a speaker at one end and a microphone at the other to create an artificial echo chamber for recording. Of course, this idea was not to use it on stage.

By the late 1950s, many amplifiers incorporated built-in tremolo, vibrato, echo and reverb effects. It began to emerge guitarists who used them a lot.
Guitarists like Chet Atkins, Luther Perkins and Roy Orbison used these ones to produce the now-classic Rock 'N' Roll and the "slapback" echo sound on stage. The tape-based echo units, such as the Watkins Copicat, influenced heavily the sound of British beat rock.

In the 1960s, the early standalone guitar effects were powered with vacuum tubes. They were bulky, expensive, fragile and not very practical for live performance.
Then, the transistor became widely available. For the first time, engineers were able to create affordable portable standalone effects, such as the Uni-Vibe Jimi Hendrix used on his song "Machine Gun".

By the late 1970s, the manufacturing of affordable solid-state effects had exploded, creating a whole new gear market that continues to thrive today.
Several stompbox preamplifiers were produced to emulate the overdriven valve amplifier tones.

The mid-1980s saw the rise of digital effects pedals built into single systems called multi-effects pedals. These electronic racks contained pedals that could activate several effects at a time.
Some of these effects were digital representations of classic overdrive, reverb and wah-wah effects, with the addition of compression, pitch shifters, octave doublers and other modern effects.

In the 1990s, multi-effect racks and floor units became prevalent, with options for switching between a wide range of overdrive sounds, in addition to other popular effects. It was also created several custom made amplifiers, producing one type of sound extremely well. Punk music called the most distorted sounds as possible.
The early 2000's saw an explosion of digital modelling as companies tried to offer popular products for home recordings, bedroom players and semi-pro musicians.

The Global Financial Crisis in the late 2000s forced most companies to curtail development and focus on high volume, low cost items. The outstanding exception was a small company, Fractal Audio, who produced the Axe-FX. This amp emulator is arguably the first device to convince experienced professionals for valve tone.

Nowadays, the amplifiers are made with hundreds of effects built in. Computer chips make it possible to carry the sound effects to anywhere with little or no setup time. All preprogrammed and ready to go. In many cases, one pedal can operate all of these effects, therefore the combinations are almost endless.

The history of some sound effects

Distortion: A man named Link Wray was the first to find this effect in 1958, when two valves in his amp were loosened. Reggie Young, guitarist star of Nashville and Memphis studies, used to remove one of the bulbs of the power stage to distort his amp.
Another important precursor of overdriven sound was Chet Atkins. In the 50s, he used a small preamp of transistors with the size of a pack of cigarettes to saturate his tube amp. Perhaps he was the first precursor of compact pedals.

Wah-Wah: It is an effect which has been widely used, ranking the second after the distortion in popularity. Its creation was also the result of an accident. In 1963, the trumpeter Clyde McCoy commissioned to Vox (important company of amplifiers) to simulate the muted effect of his trumpet.
Trying to reach this result, the Vox technicians found the Vox Wah-Wah effect which achieves a very similar sound to its name. The company put it in a pedal and it went on sale at the time that Jimmy Hendrix began recording his first album.

Octave: Roger Mayer, a sound technician of the '60s, wanting to modify the octave fuzz, he developed this one.
This effect adds an acute note (in the upper octave) of the original one that gives the instrument.

Flanger: From the Beatles, it was common to "double tracking" effect was achieved with two recording machines, the first one dephased with another one.
To dephase, it rubbed with the hand against the fins of rolls of tape of one of them, hence the name os this effect: flange.

Chorus: Around 1977, Roland, with his division of Boss effects, released this processor.
The chorus effect was the most used effect in the '80s.
This effect is generated taking the input signal, slowing it about 20 milliseconds and modulating it. So it can say that this effect is a derivative of the flanger, but cleaner.

PROGRAMMING

MATLAB simulation

Before programming the audio effects in the DSP Starter Kit (DSK), it was decided to make some simulations with the software MATLAB, which is used in many subjects of the degree.

Firstly, it was decided to start working on the effect simulations with the frequency spectrum of the input signal. It was believed that working directly with the frequencies would make the modification of the signal easier.

Therefore, the input signal (which is based on amplitude values over samples) must be transformed into the frequency spectrum. The frequency spectrum shows the amplitude values of the different frequencies which form the signal.

In fact, some effects such as the Octaver (which generates an output signal with its frequency an octave above the input frequency) can only be generated when working inside the spectrum of the signal. For this simple reason, it was determined that this would be the ideal way to work.

Although it seemed the most optimal way, it was also known that it required additional work, since it is necessary to reach its frequency spectrum first. Initially, this point was not important and the complexion of the matter was unknown.

To get the frequency spectrum, the fastest way is using the **Fast Fourier Transform** (FFT).

The FFT is an efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. In exchange, the DFT transforms a function in the time domain in a representation of the frequency domain.
The DFT requires that the input function is a discrete and finite duration sequence.
Therefore, this algorithm is perfect because the input signal is sampled and stored in an array using a Digital Analog Converter (DAC), before it can use its samples.

Due to the fact that the final application gathers all the information in real time, it has to simulate equally.
Therefore it cannot take a full data array and make the FFT. It must be done every certain period of time, depending on the quantity of samples which are being entered.

To perform the FFT correctly, it must understand first the concepts of **window length** and **window step**.

The window length is the dimension of data that it must be taken to perform the FFT. This means it has to wait until all the necessary information is saved.

The window step is the time (in samples) which it has to wait between each FFT to ensure that each new result provides new information.

Both elements are calculated depending on the sampling frequency according the following equations:

$$\text{Windows lentgh (in samples)} = \frac{1}{\text{min. freq. (in Hz)}} \cdot \text{sampling freq. (in Hz)}$$

$$\text{Window step (in samples)} = \frac{1}{\text{max. freq. (in Hz)}} \cdot \text{sampling freq. (in Hz)}$$

These formulas are correct, but it can set other different values of these parameters to adapt it in each specific situation.

Again, the application is processed in real time, therefore the DSP app must perform this operation continuously.
For this reason, with a correct value of the previous parameters, the FFT is performed the minimum number of times, subtracting processing work to the DSP.

To sum up, the FFT is performed after a specific number of samples (window step) with a specific number of samples (window length), returning a single value of frequency.

After numerous FFTs, the frequency spectrum of the input signal is formed, which has the ratio between the amplitude and the frequency of the input signal.

Finally, when it has the spectrum of frequencies, if it searches for the maximum amplitude, the main note of the signal is determined.
And with this value, the output signal can be generated.

This system seemed feasible at first, but after several tests and simulations, it was determined that, in fact, it has several important problems:

- It can only recognize one note at a time. If it sounds more than one note at a time, it is impossible to find them all with the amount of noise and harmonics which the input signal has... at least by this way of working.

- The sound generated at the output is a pure tone. Therefore there is not similarity with the characteristic sound that a musical instrument delivers (due to the force to generate the note, the wood of the instrument, the body resonance, etc.).

- To fix the last point, it is necessary look for the formula which defines the sound of the musical instrument to apply the found frequency. This process is called characterization. But each instrument has its own formula (even among different models of the same instrument). Therefore, it is a lot of absurd work.

- In addition, if it is decided to introduce a complex sound (like an audio track from a music player), the output would be unrecognizable.

With those problems which greatly complicated the work, after weeks of research, it reached the following conclusion:
Not only this was of working is more complicated, but it also requires more processing work to the DSP microcontroller.

As a result, the first form of work is discarded. The best solution is work directly with the amplitude values over samples.

A lot of time was devoted to reaching this conclusion, therefore it starts to work directly on the DSK now, taking the input samples with the audio codec and treating them consequently.

But fortunately, it was not all a waste of time. The algorithm designed in MATLAB is used to design the implementation of the tuner in the DSP.

Part of this algorithm is used in the DSP app to detect the lowest note using the FFT and their other functions.

However, for a proper performance of the tuner, it can only play one note at a time. If more than one note is reproduced, the algorithm only takes the lowest one.

Followed it shows part of the code in MATLAB, which is used for the final algorithm of the tuner in the DSP app. The rest of the code is shown in the annexes.

```
for N=1:length(ArrayNote)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    READING THE INPUT SIGNAL IN REAL TIME                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic
    for j=1:(WinLength-1)
        inputSamples(j,1) = inputSamples(j+1);
    end
    inputSamples(WinLength) = ArrayNote(N);
timeInputSignalReading=toc;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              WINDOW STEP                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic
    samplesCount = samplesCount+1;
    if samplesCount>=WinStep && N>=WinLength
        samplesCount = 0;
timeWinStep=toc;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              MAIN FREQUENCY                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic
        if inputSamples<=0.05 & inputSamples>=-0.05
%             mainFrec = 0;
%             noteFreq = 0;
        else
            % The window type is applied to the input samples
            for j=1:WinLength
                WinInput(j) = coefWindow(j)*inputSamples(j);
            end
            % The FFT is performed
tic;
            windowFFT = abs(fft(WinInput, fs));
timeFFT = toc;

            SpecAmpMax = 0;
            for j=1:2000  % The array is crossed until 2000 Hz
                if windowFFT(j)>SpecAmpMax
                    SpecAmpMax = windowFFT(j);
                    mainFreq = j;
                end
            end
        end
timeMainFrec=toc;
```

All the code is explained in the annexes.

Texas Instruments TMS320C6713

Differences between DSP and General Purpose Processor

A microprocessor incorporates the functions of a the central processing unit of a computer (CPU) on a single or few integrated circuits. The purpose of a microprocessor is to accept digital data as input, process it as per the instructions, and then provide the output. This is known as sequential digital logic. The microprocessor has internal memory and operates basically on the binary system.

A general purpose microprocessor is a processor that is not tied to or integrated with a particular language or piece of software. Most of the general purpose microprocessors are present in personal computers. They are often used for computation, text editing, multimedia display, and communication over a network. Other microprocessors are part of embedded systems.
These ones provide digital control over practically any technology, such as appliances, automobiles, cell phones, industrial process control, etc.

The DSP processor, on the other hand, is a particular type of microprocessor. DSP stands for digital signal processing. It is basically any signal processing that is done on a digital signal or information signal. A DSP processor is a specialized microprocessor that has an architecture optimized for the operational needs of digital signal processing.

DSP aims to modify or improve the signal. It is characterized by the representation of discrete units, such as discrete time, discrete frequency, or discrete domain signals. DSP includes subfields like communication signals processing, radar signal processing, sensor array processing, digital image processing, etc.

The main goal of a DSP processor is to measure, filter and compress digital or analog signals.
It does this by converting the signal from a real-world analog signal to a digital form. To convert the signal it uses a digital-to-analog converter (DAC). However, the required output signal is often another real-world analog signal. This transformation also requires a digital-to-analog converter.

These algorithms can run on various platforms. Such as general purpose microprocessors and standard computers. Specialized processors called digital signal processors (DSPs). Purpose-built hardware such as application-specific integrated circuit (ASICs) and field-programmable gate arrays (FPGAs). Digital Signal Controllers and stream processing for traditional DSP, or graphics processing applications, such as image or video.

The main difference between a DSP and a microprocessor is that the DSP processor has features designed to support high-performance, repetitive, numerically intensive tasks. DSP processors are designed specifically to perform large numbers of complex arithmetic calculations and as quickly as possible. They are often used in applications such as image processing, speech recognition and telecommunications. In comparison with general microprocessors, DSP processors are the more efficient at performing basic arithmetic operations, especially multiplication.

Most general-purpose microprocessors and operating systems can execute DSP algorithms successfully. However, they are not suitable for use in portable devices such as mobile phones.
Hence, specialized digital signal processors are used.

Digital Signal Processors have approximately the same level of integration and the same clock frequencies as general purpose microprocessors. But they tend to have better performance, lower latency, and no requirements for specialized cooling or large batteries. This allows them to be a lower-cost alternative to general-purpose microprocessors.

DSPs also tend to be from two to three times as fast as general-purpose microprocessors.
This is due to architectural differences. DSPs tend to have a different arithmetic unit architecture. Specialized units, such as multipliers, etc. Regular instruction cycle, a RISC-like architecture. Parallel processing. A Harvard Bus architecture. An internal memory organization. Multiprocessing organization, local links and memory banks interconnection.

For these reasons, the present project is based on a DSP, using its processing capacity for math operations and audio treatment.
Due to this, the EET provides the TMS320C6713 DSP Starter Kit, which is used in the audiovisual system degree.

Description of the TMS320C6713

The TMS320C6713 DSP Starter Kit (DSK) is a low-cost development platform designed to speed up the development of high precision applications based on TI´s TMS320C6000 floating point DSP generation. The kit uses USB communications for true plug-and-play functionality.
Both experienced and novice designers can get started immediately with innovative product designs with the DSK´s full featured Code Composer Studio IDE (Integrated Development Environment) and eXpressDSP Software which includes DSP/BIOS and Reference Frameworks.

This kit is based for a high performance. It has an advanced architecture Very Long Instruction Word (VLIW) developed by Texas Instruments (TI), whose DSP offers a great choice for multichannel and multifunction applications.

The C6713 DSK tools include the latest fast simulators from TI and access to the Analysis Toolkit via Update Advisor, which features the Cache Analysis tool and Multi-Event Profiler.
Using Cache Analysis, developers improves the performance of their application by optimizing cache usage. By providing a graphical view of the on-chip cache activity over time the user can quickly determine if their code is using the on-chip cache to get peak performance.

The C6713 DSK allows to download and step through code quickly and uses Real Time Data Exchange (RTDX) for improved Host and Target communications.
The DSK includes the Fast Run Time Support libraries and utilities such as Flashburn to program flash and Update Advisor (to download tools). It also includes utilities, software and a power on self-test and diagnostic utility to ensure that the DSK is operating correctly.

The full content of the kit includes:

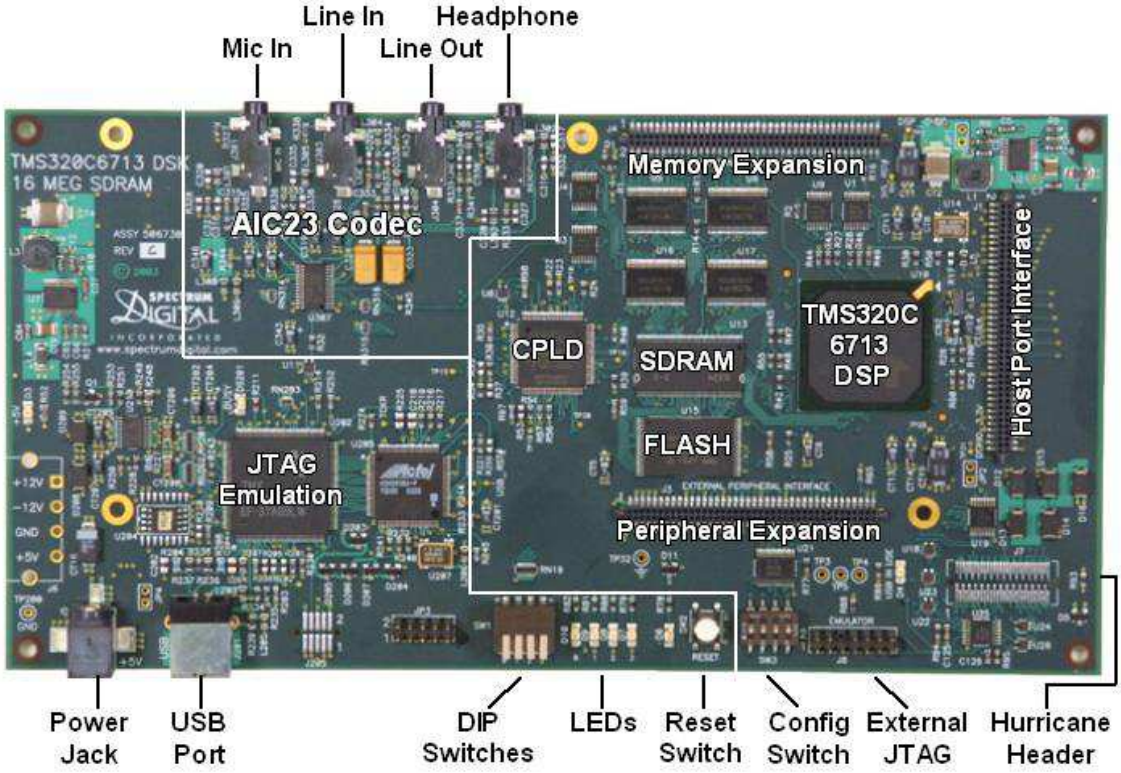
- C6713 DSP Development Board
- C6713 DSK Code Composer Studio IDE including the Fast Simulators and access to Analysis Toolkit on Update Advisor
- Quick Start Guide
- Technical Reference
- Customer Support Guide
- USB Cable
- Universal Power Supply
- AC Power Cord(s)
- MATLAB from The Mathworks 30 day free evaluation

Features

The DSK features the TMS320C6713 DSP. A 225 MHz device which delivers up to 1350 million of floating-point operations per second (MFLOPS). 1800 million of instructions per second (MIPS) with double multiplications fixed-/floating-point, and up to 450 millions of multiplications per second (MMACS).
This DSP generation is designed for applications that require high precision accuracy.
The C6713 is based on the TMS320C6000 DSP platform designed for needs of high-performing high-precision applications such as pro-audio, medical and diagnostic.

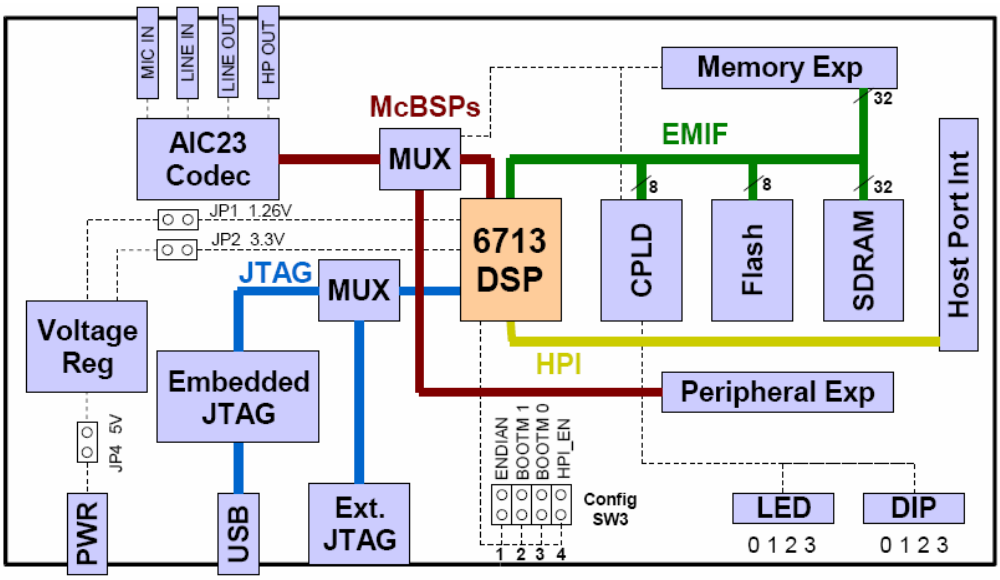
Other hardware features of the TMS320C6713 DSK board include:

- Texas Instrument's TMS320C6713 DSP operating at 225 Mhz.
- Embedded USB JTAG (Joint Test Action Group) controller with plug and play drivers, USB cable included
- TLV320AIC codec
- 2M x 32 on board SDRAM
- 512K bytes of on board Flash ROM
- 3 expansion connectors (Memory Interface, Peripheral Interface, and Host Port Interface)
- On board IEEE 1149.1 JTAG connection for optional emulator debug
- Four 3.5 mm. audio jacks (microphone, line-in, speaker, and line out)
- 4 user definable LEDs
- 4 position dip switch, user definable
- +5 Volt operation only, power supply included
- Size: 8.25" x 4.5" (210 x 115 mm), 0.062" thick, 6 layers
- Compatible with Spectrum Digital's DSK Wire Wrap Prototype Card
- RoHS Compliant



Peripherals

As shown in the following picture, this DSK has a good amount of peripherals that help to the proper functioning of DSP. They are enabled/disabled to interact with the user.



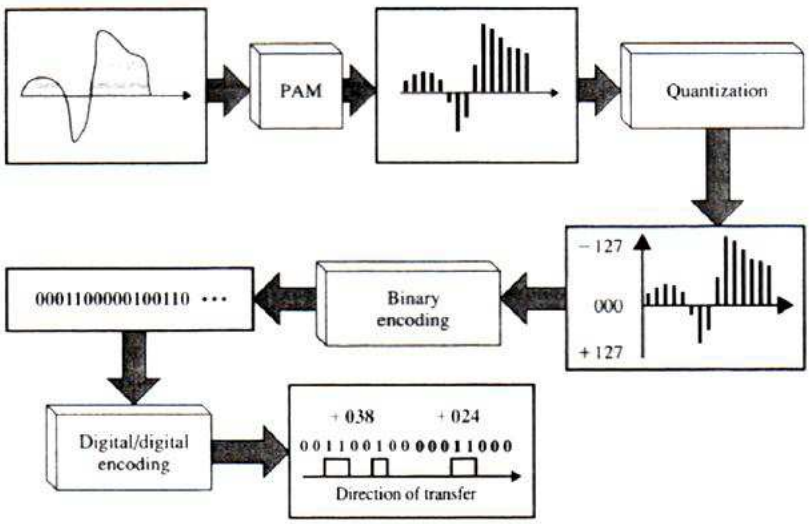
The following describes the most important peripherals which are used to perform this project:

- **Analog-Digital Converter (ADC)**

The analog-digital converter consists basically in performing periodically measurements of the input signal amplitude and translating them into a numeric language.

The digital analog conversion process basically consists of four stages:

- Sampling
- Quantification
- Coding
- Digital-Digital Recoding transmission



Sampling

Sampling consists in taking periodic samples of the wave amplitude. The speed with which the sample is taken, namely, the number of samples per second is what is known as sampling frequency (FS). It depends on the Nyquist theorem. The theorem establishes that the sampling frequency must be the double of maximum frequency (FM) signal to be sampled.

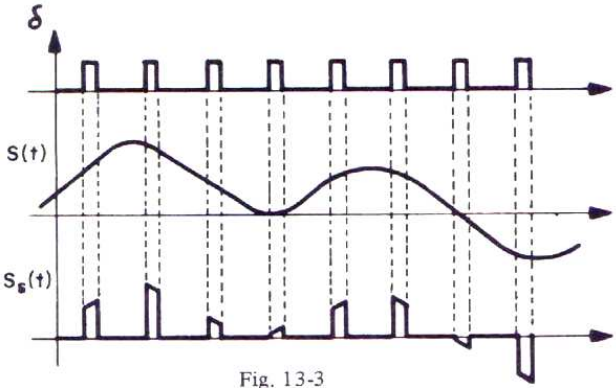
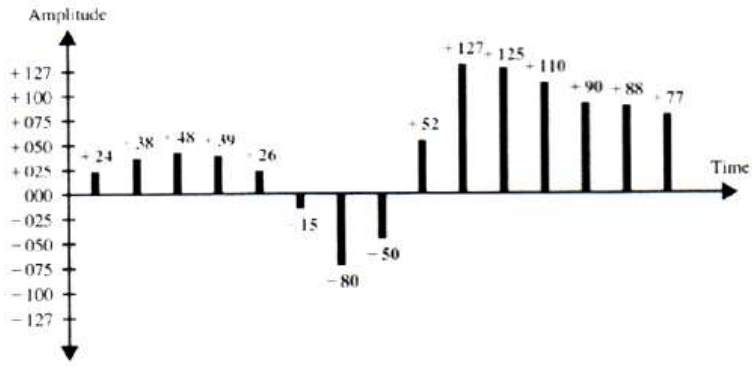


Fig. 13-3

Quantification

The quantification process converts a sequence of amplitude samples into a predetermined sequence of discrete values according to the code used.
During the quantification process, the voltage level of each sample is measured, obtained from the sampling process. They are saved into a finite (discrete) amplitude value selected by approximation within a previously set level range.



The preset values to adjust the quantization are chosen depending on the very resolution itself using the code while encoding. If the level obtained is not identical with any other, it is taken as the next lower value.
Then, the analog signal (which it can take any value) is converted into a digital signal, because the preset values are finite.
However, it is not translated into binary code yet. The signal has been represented by a finite value while encoding, becoming it in a succession of zeros and ones.

Thus, the digital signal resulting after quantification is substantially different from the analog electrical signal. Therefore, there is always some difference between them. The difference is known as the quantification error. This error occurs when the actual sample value is not equivalent to any of the steps available for its approach. The distance between the actual value and its approximation is also very large. A quantification error becomes a noise signal when playing back after digital decoding process.

Coding

Coding involves the translating of the analog voltage values which have been quantified to binary by preset codes. The analog signal will be converted into a digital pulse train.

+024	00011000	−015	10001111	+125	01111101
+038	00100110	−080	11010000	+110	01101110
+048	00110000	−050	10110010	+090	01011010
+039	00100111	+052	00110110	+088	01011000
+026	00011010	+127	01111111	+077	01001101

Sign bit
+ is 0 − is 1

This is the specific code used for encoding / decoding of data. Indeed, the word codec is an abbreviation for Coder-Decoder.

Parameters defining the codec

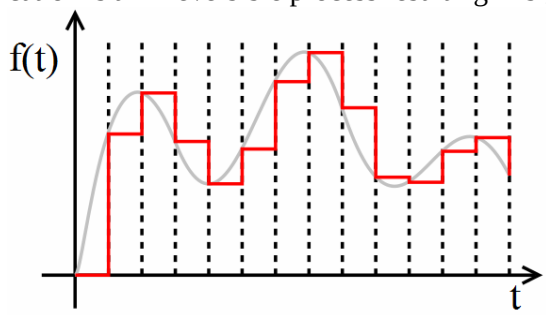
- Number of channels: It indicates the signal type to address: monaural, binaural or multichannel.
- Sampling: The sampling frequency refers to the amount of amplitude samples taken per unit of time in the sampling process. According to the theorem Nyquist-Shannon, sampling rate determines the bandwidth based on the sampled signal. That is to limit the maximum frequency of the sinusoidal components which form a periodic waveform.
- Bit rate: The bit rate is the speed or data transfer rate. Its unit is the bit per second (bps).
- Resolution: It determines the accuracy with the original signal is reproduced. It typically uses 8, 10, 16 or 24 bps. High precision means more number bits.
- Loss: Some codecs removes certain amount of information to do the compression, hence the resulting signal is not equal to the original (loss compression).

• Digital-Analog Converter (DAC)

The reverse process is much less complex. It consists in putting the values of the samples in the order they have been processed according to the used algorithm.
The filters of output recomposition DAC are responsible for converting the resultant signal of discrete values (digital) into an analog signal.

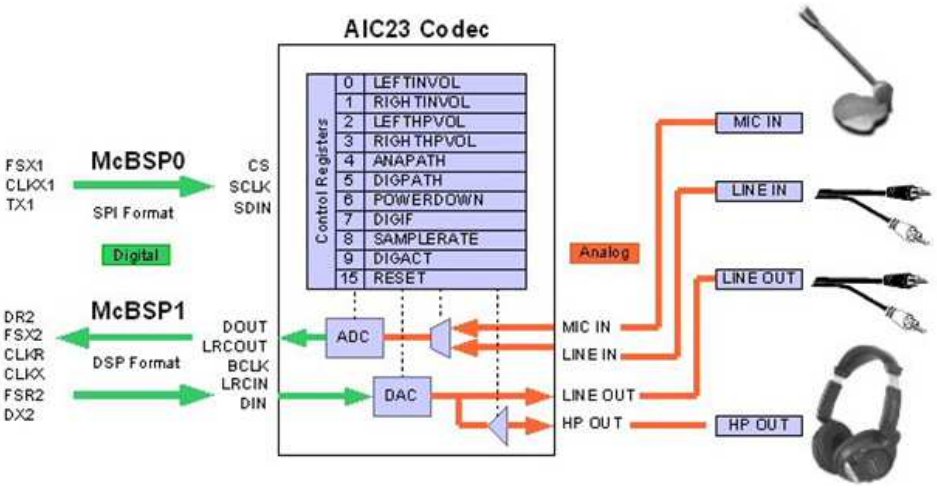
The analog signal can be reconstructed from its samples. The only condition is that the sampling rate is high enough to avoid the problem referred to as aliasing (the signal becomes indistinguishable when it is sampled).

According to the Nyquist-Shannon theorem, higher sampling rate for a signal should not be interpreted as a higher fidelity in the signal reconstruction. The sampling process is reversible, which means that the reconstruction can be performed in an accurate way.
Furthermore, the quantification is an irreversible process resulting in signal distortion.



- **AIC23 codec**

The evaluation kit DSK6713 has a stereo audio codec TLV320AIC23 (AIC23) based on the delta-sigma technology. The AIC23 allows conversion frecuencies of 8, 16, 24, 32, 44.1, 48 and 96 kHz. These sample frequencies are generated from a clock signal of 12 MHz. The same clock signal that it is used in the USB interface.



The comunication with the AIC23 codec is performed with the McBSP0 and McBSP1 (Multichannel Buffered Serial Port). The McBSP0 serial port is used like a unidirectional channel for the sending and receiving of data from the codec or to the codec.

The AIC23 has 10 control registers that allow to manage the volume, data format, sampling frecuency, selection of input signals, etc.
For the access to the audio codec, it is necessary the Board Support Library.

Features:

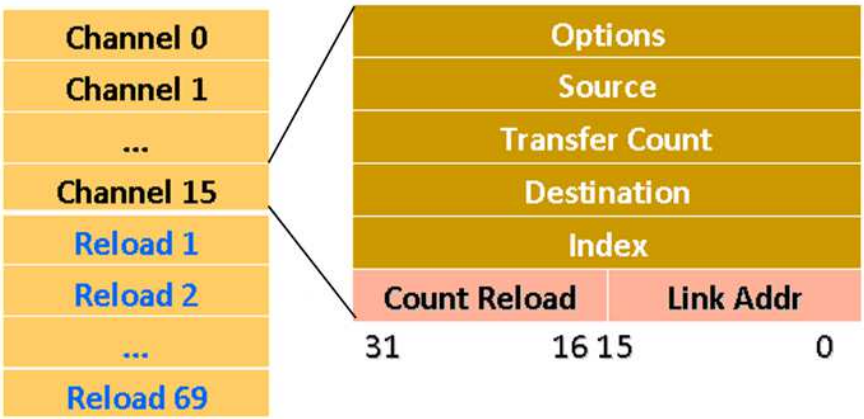
- High-Performance Stereo Codec
  90-dB SNR Multibit Sigma-Delta ADC (A-weighted at 48 kHz)
  100-dB SNR Multibit Sigma-Delta DAC (A-weighted at 48 kHz)
  1.42V– 3.6V Core Digital Supply: Compatible with TI C54x DSP Core Voltages
  2.7V– 3.6V Buffer and Analog Supply: Compatible Both TI C54x DSP Buffer Voltages
  8-kHz – 96-kHz Sampling-Frequency Support
- Software Control Via TI McBSP-Compatible Multiprotocol Serial Port
  2-wire-Compatible and SPI-Compatible Serial-Port Protocols
  Glueless Interface to TI McBSPs
- Audio-Data Input/Output Via TI McBSP-Compatible Programmable Audio Interface
  I2S-Compatible Interface Requiring Only One McBSP for both ADC and DAC
  Standard I2S, MSB, or LSB Justified-Data Transfers
  16/20/24/32-Bit Word Lengths
  Audio Master/Slave Timing Capability Optimized for TI DSPs (250/272 fs), USB mode
  Industry-Standard Master/Slave Support Provided Also (256/384 fs), Normal mode
  Glueless Interface to TI McBSPs
- Integrated Total Electret-Microphone Biasing and Buffering Solution
  Low-Noise MICBIAS pin at 3/4 AVDD for Biasing of Electret Capsules
  Integrated Buffer Amplifier with Tunable Fixed Gain of 1 to 5
  Additional Control-Register Selectable Buffer Gain of 0 dB or 20 dB
- Ideally Suitable for Portable Solid-State Audio Players and Recorders

- Stereo-Line Inputs
  − Integrated Programmable Gain Amplifier
  − Analog Bypass Path of Codec
- ADC Multiplexed Input for Stereo-Line Inputs and Microphone
- Stereo-Line Outputs
  − Analog Stereo Mixer for DAC and Analog Bypass Path
- Volume Control With Mute on Input and Output
- Highly Efficient Linear Headphone Amplifier
  − 30 mW into 32 Ω From a 3.3-V Analog Supply Voltage
- Flexible Power Management Under Total Software Control
  − 23-mW Power Consumption During Playback Mode
  − Standby Power Consumption <150 µW
  − Power-Down Power Consumption <15 µW
- Industry's Smallest Package: 32-Pin TI Proprietary MicroStar Junior
  − 25 mm2
- Total Board Area
  − 28-Pin TSSOP Also Is Available (62 mm2 Total Board Area)

- **EDMA**

The Enhanced Direct Memory Access (EDMA) is a peripheral that it can configures to copy data from one place to another one without the CPU's intervention. It can be set up to copy data or program from a source (external/ internal memory, or a serial port) to a destination (e.g. internal memory). After the transfer is completed, the EDMA can autoinitialize itself and perform the same transfer again, or it can be reprogrammed with another configuration.

There are 16 memory direct access channels which can be configured independently for the data transmision, and 69 reload channels to set a new configuration to previous channels.
These charging channels allow updating the different access channels to perform a new data sending when the previous one has finished.

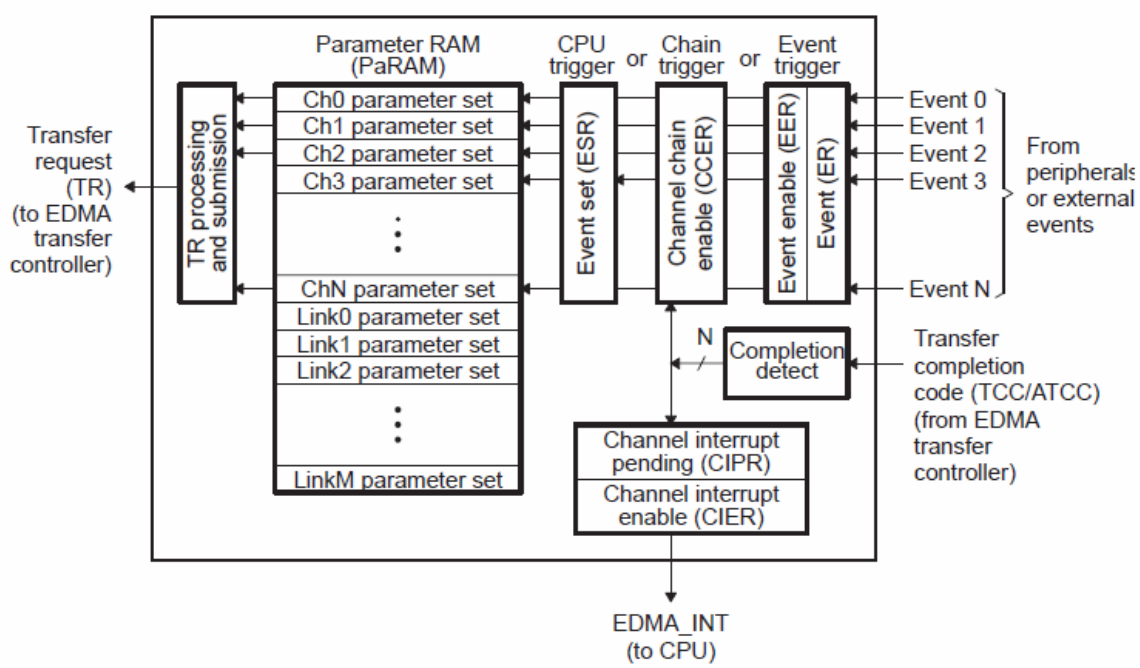


Registers:

- Options (OPT): It allows the configuration of the different avaiable options for perform the data sending, the priority, the number of bits to send, etc.
- Source (SRC): The memory direction where the data is to be sent.
- Transfer Count (CNT): The number of data to send.
- Destination: The memory direction where the data is copied.
- Count Reload/ Link Addr (RLD): It specifies the charging channel associated with the channel which is using. This allows to reload automaticaly this channel with the data in the charging channel, after the previous task has been completed.

Features:

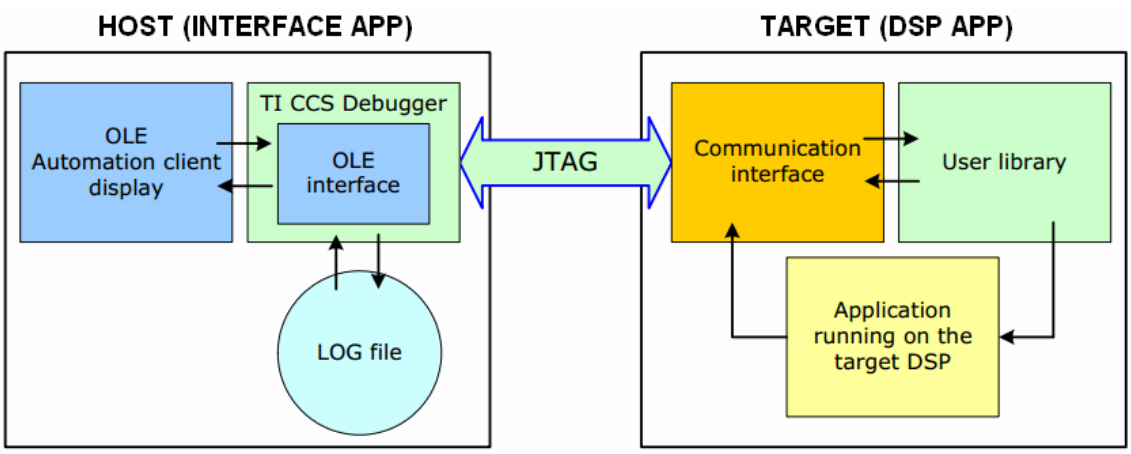
- Fully orthogonal transfer description
  - 3 transfer dimensions
  - A-synchronized transfers: 1 dimension serviced per event
  - AB- synchronized transfers: 2 dimensions serviced per event
  - Independent indexes on source and destination
  - Chaining feature allows 3-D transfer based on single event
- Flexible transfer definition
  - Increment or constant transfer addressing modes
  - Linking mechanism allows automatic PaRAM set update
  - Chaining allows multiple transfers to execute with one event
- Interrupt generation for:
  - Transfer completion
  - Error conditions
- Debug visibility
  - Queue watermarking/threshold
  - Error and status recording to facilitate debug
- 64 DMA channels
  - Event synchronization
  - Manual synchronization (CPU(s) write to event set register)
  - Chain synchronization (completion of one transfer triggers another transfer)
- 8 QDMA channels
  - QDMA channels are triggered automatically upon writing to a PaRAM set entry
  - Support for programmable QDMA channel to PaRAM mapping
- 128 PaRAM sets
  - Each PaRAM set can be used for a DMA channel, QDMA channel, or link set (remaining)
- 2 transfer controllers/event queues. The system-level priority of these queues is user programmable.
- 16 event entries per event queue



- **Communication protocol RTDX**

RTDX is a protocol that TI developed to send data via a debugging interface from a client processor to a host processor and vice versa.
For the DSK-boards this is in the JTAG interface generally.

Many applications require that a host controls the operation of a DSP-system which starts actions (like reading the output values from DSP operations). Mostly this is done via a graphical user-interface (GUI).
The DSK development boards of TI are connected to a host-PC either via a printer port or via an USB-connection.



The same link is also available for the user to transfer data from the host to the DSK-board and vice versa. TI implemented for this purpose a communication channel architecture called RTDX. The basic functionality is similar to I/O-channels found in major operating systems.

To work properly, it needs a "Code Composer Studio" program on a PC, which has to be connected by hardware and software to the DSK-board.
The RTDX-link is only for development purposes. Therefore it has to be replaced by other link-implementations, such as a standard serial or USB link, in final applications.

The RTDX provides data types and functions for:

- To send data from the Target application to the Host application.
- To send data from the Host application to the Target application.
- To send event data from the Target application to the Host application.

Code Composer Studio

Introduction to Code Composer Studio

Designers can readily target the TMS32C6713 DSP through TI´s robust and comprehensive Code Composer Studio DSK development platform. The tools, which run on Windows 98, Windows 2000 and Windows XP, allow developers to seamlessly manage projects of any complexity.

Code Composer Studio (CCStudio) is an Integrated Development Environment (IDE) for Texas Instruments (TI) embedded processor families.

CCStudio comprises a suite of tools used to develop and debug embedded applications.
It includes compilers for each of TI's device families, source code editor, project build environment, debugger, profiler, simulators, real-time operating system and many other features. The intuitive IDE provides a single user interface which it shows each step of the application development flow. Familiar tools and interfaces allow users to get started faster than ever before. They also add functionality to their application thanks to sophisticated productivity tools.

CCStudio is based on the Eclipse open source software framework.
The Eclipse software framework was originally developed as an open framework for creating development tools. Eclipse offers an excellent software framework for building software development environments and it is becoming a standard framework used by many embedded software vendors.
CCStudio combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded developers.

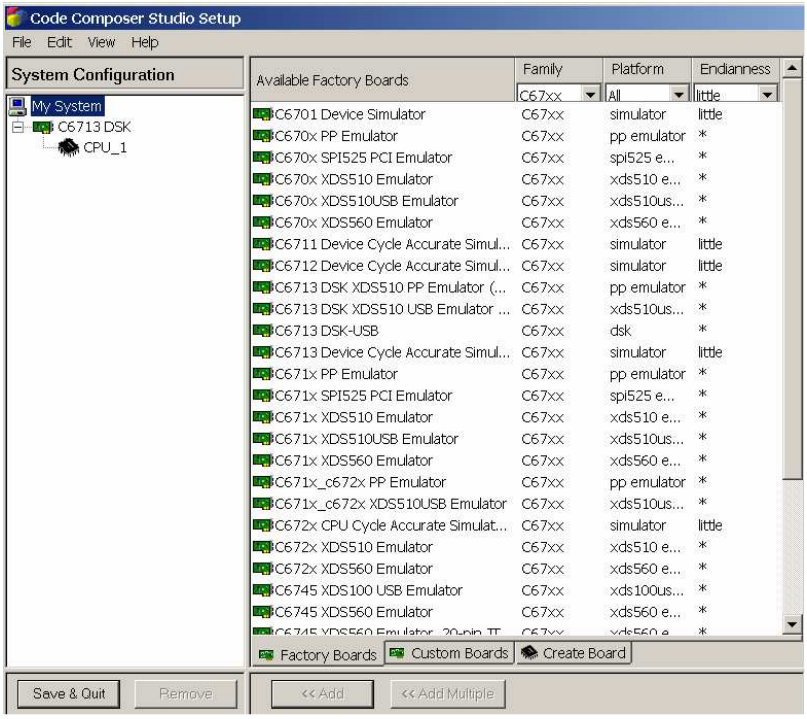
CCStudio features for the TMS320C6713 DSK includes:

- A complete IDE, an efficient optimizing C/C++ compiler assembler, linker, debugger, an a advanced editor with Code Maestro technology for faster code creation, data visualization, a profiler and a flexible project manager.
- DSP/BIOS real-time kernel.
- Target error recovery software.
- DSK diagnostic tool.
- "Plug-in" ability for third-party software for additional functionality.
- Test/sample code provided to reduce coding time.
- Compatible with National Instruments LabView Embedded 2.0.
- Compatible with JTAG emulators from Spectrum Digital.

Creating a Code Composer Studio project

First of all it executes the “Setup CCStudio” program to assign specifically the hardware environment which will be used to work.

In this case, it selects the device "C6713 DSK-USB" (not simultor). It adds it to the project and finally it saves the configuration selecting “Save & Quit” button.

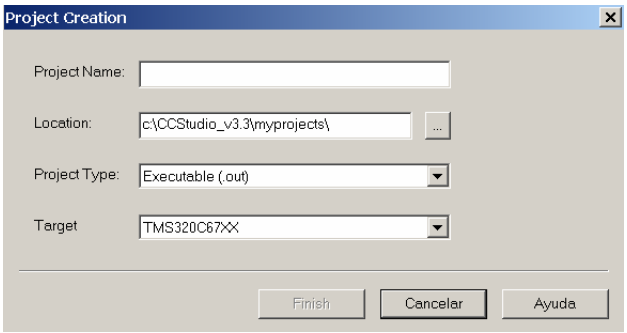


It confirms the upper configuration by clicking “Yes” in the “Start Code Composer Studio on exit?” window. After that, the CCStudio is opened automatically.

Once the CCStudio is opened, it has to check some configuration options in the new window (which appears when it clicks in “Option – Customize”) for a optimized use:

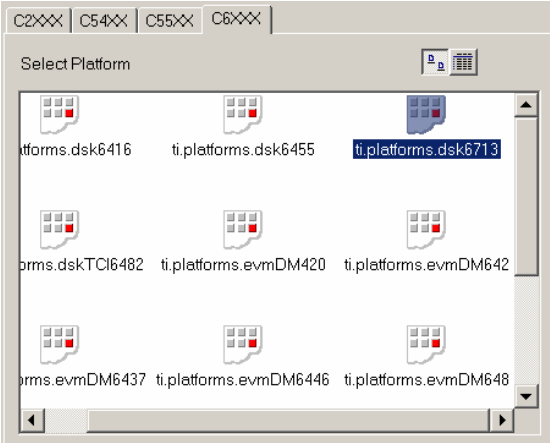
- In the “Debug properties” label, it selects the “Perform Go Main automatically” option.
- In the “Program/Project/CIO” label, it selects the “Disable All Breakpoints When Loading New Programs” and “Auto-save Projects Before Build” options.
- In the “Control Window Display” label, it selects the “Current Project”, “Display full path” and “Close all windows on Project Close” options and it deselects the “Product Name” option.

With the CCStudio configured, it selects "Project - New" to create a new project and it opens a new window to specify the project name, location, project type and its DSP family.



It clicks in the “finish” button to accept the upper features and create the project.

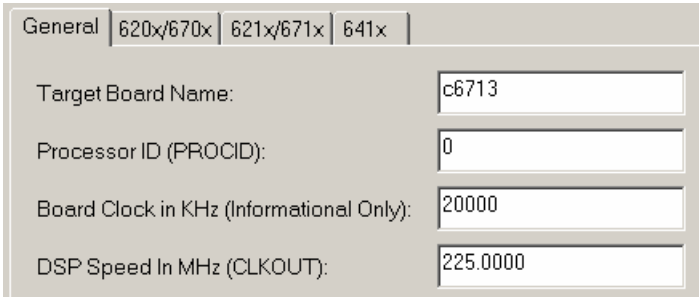
Then it creates the configuration file (.tcf) in "File - New - DSP/BIOS Configuration".
It selects the label "C6XXX" and after the “ti.platforms.dsk6713” template to work with the TMS320C6713 and finally it selects "Ok".



Afterwards it is configured the board in the configuration window:

It goes to “System – Global Settings” and it selects “Properties” with the right button.
It specifies in the popup window "Target Board Name": 6713 and "DSP Speed in MHz (CCKOUT)": 225MHz (These are the features of the used hardware).

It applies and accepts the changes.



Then it configures the memory space:

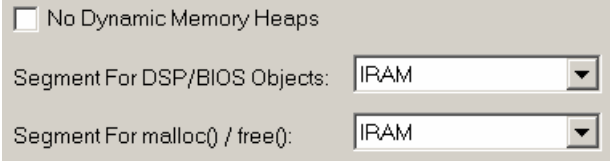
It goes to “MEM – Memory Section Manager – IRAM” and it selects “Properties” with the right button of the mouse. It marks the options “Create a heap in this memory” and “Enter a user defined heap identifier label”.

It also writes MEM_HEAP in the label “heap identifier label”.

It applies and accepts the changes.

Still in the memory space configuration, it clicks in "MEM – Memory Section Manager" and with the right boton select "Properties".

It selects the IRAM (already configured) for the "Segment For DSP/BIOS Objects" and "Segment For malloc() / free()" labels.



It applies and accepts the changes.

At this point, it saves the configuration file in the same location when the project is created.
Then it adds to this project clicking with the right button in "Add files to project" on the XX.pjt file, which can be seen in the left window (the window of the project hierarchy).

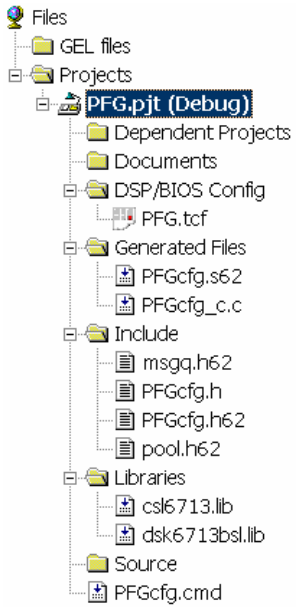
Consequently, it adds automaticaly the cfg.s62 y cfg_c.c files (the ASM and C codes created respectively by the tool of graphic configuration).

There is another file (automaticaly created when the configuration file was saved) which needs to be added manually. This file is the command file of linker called cfg.cmd.

Now it adds to the project the Board Support Library (BSL) and Chip Support Library (CSL) libraries, called csl6713.lib and dsk6713.lib respectively.

With all the necessary files inside the project, it clicks in "Scan All File Dependances" to introduce automaticaly the rest of associated files.
These new files are introduced for a correct operation to the project.

The following picture shows the project hierarchy with all the files inside of it:



From this moment, it is possible to create codes in C.

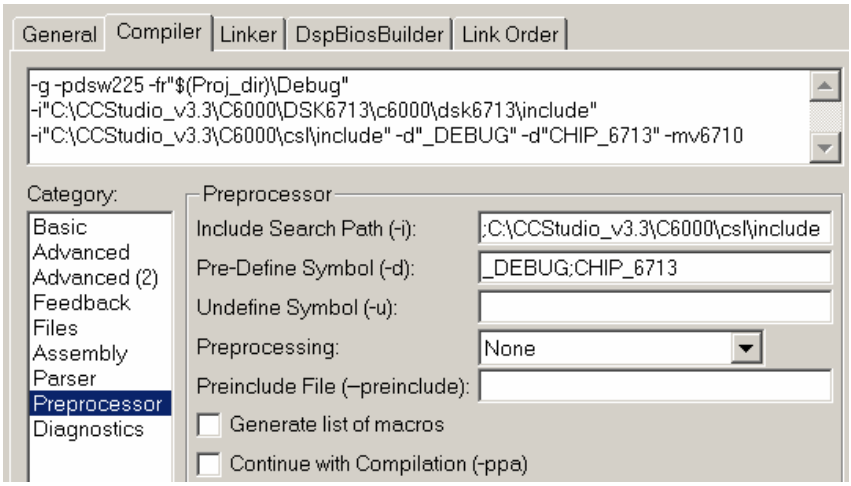
To create a new .c file, it clicks in “File – New File” and then this file can be added to the project with the “Add files to project” option with the right button from the XX.pjt.

The last configurations are the compile and link options.

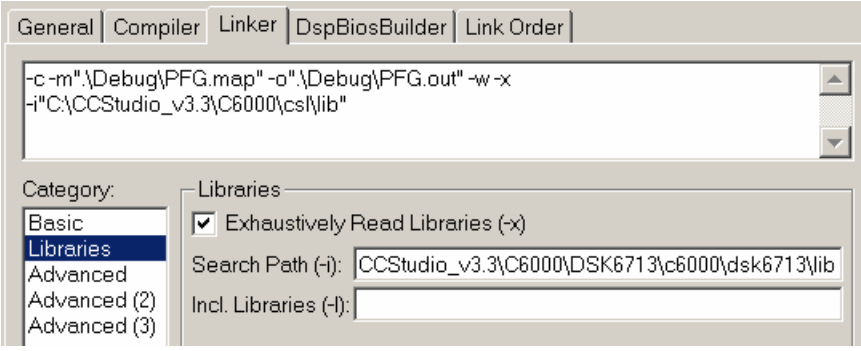
It clicks in “Project – Build Options” and, inside the new window, it goes to “Compiler – Category: Basic” and it selects the processor that it is used (“C671x”).
Here it can also select the use of the debug mode and the optimized options.

In the same new window, it goes to the “Compiler – Category: Preprocessor” label to add the directory (if it exists more than one directory, they must be separated using “;”). This directory contains the header files (extension .h).
These files have the implementation of the board and DSP functions.

If it works with the CSL library, it has to write “CHIP_6713” (it puts “;” previously to separate this name with the previous one) in the “Pre-Define Symbols” label too.



Finally it goes to “Linker – Category: Libraries” to add the CSL and DSK6713 directories which contain the necessary library files (extension .lib).

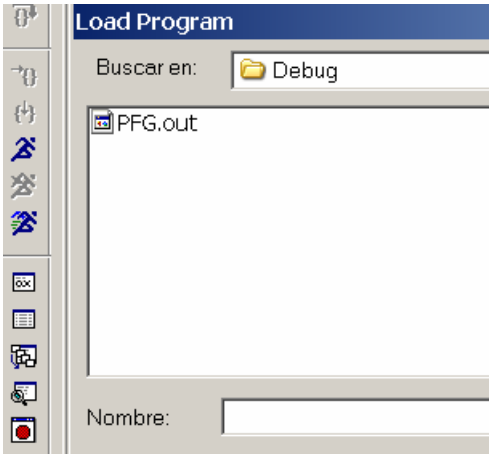


It applies and accepts the changes.

The configuration is finished. If it is already done the ANSI C program, it would be necessary to click in “Rebuild all” button to compile all the project.

If it appears some warnings or errors, they have to be solved.

With the code compiled without errors or warnings, it opens the executable file (.out) in “File – Load Program” option (inside Debug folder) to prepare to run the ANSI C code.



Finally it clicks the “Run” button to run the code.

DSK program structure

Here it shows the different parts of the performed program in the DSK. It has been scheduled to receive the audio signal from the codec, process the received signal to generate the sound effects and take out the results to reproduce in speakers.

The implementation of the effects is described in a separate section for a better explanation.

The present code is divided into the following parts according to the functionality of each one:

Prototypes

The ANSI C language needs to define first the methods to be used (except for the main method). Thus, the compiler can identify them when it sees them.

It shows some of them:

```
void edma_init (void);  //It defines the prototypes of
void EDMA_HWI (void);   //the functions which are used
void ByPass (void);
void Overdrive (void);
void Compressor (void);
void NoiseGate (void);
void AutoWah (void);
void DelayEcho (void);
void Tremolo (void);
void Tuner (void);
void DSP_bitrev_cplx (int *x, short *index, int nx);
```

edma_init → Initialization of the EDMA peripheral.
EDMA_HWI → Interruption produced by the EDMA.
ByPass → It takes out the input signal.
Overdrive → It generates the Overdrive effect.
Distortion → It generates the Distortion effect.
Fuzz → It generates the Fuzz effect.
Compressor → It generates the Compressor effect.
Expansor → It generates the Expansor effect.
Noise Gate → It generates the Noise Gate effect.
AutoWah → It generates the auto Wah effect.
Panning → It generates the Panning effect.
Chorus → It generates the Chorus effect.
Flanger → It generates the Flanger effect.
Tremolo → It generates the Tremolo effect.
Delay/Echo → It generates the Delay/Echo effect.
Reverb → It generates the Reverb effect.
DSP_radix2 → It calculates the FFT radix 2 (The result is disordered).
DSP_bitrev_cplx → It orders the previously calculated FFT.

Header files

To control the DSK and to use of all peripherals, it must define first a set of header files (.h). These header files contain the declaration of the functions to use when the compiler finds them. Anyway, these functions are implemented in other files.

To define them, it must put the word "# include" and then the name of the specific file between brackets. However, it is not necessary put ";" at the end of the line.

Some of the used header files are:

```
#include <csl.h>
#include <csl_irq.h>
#include <dsk6713_aic23.h>
#include <math.h>
#include <rtdx.h>
#include <tw radix2.h>
```

csl.h → To manage the internal functions of the DSP.
csl_irq.h → To use the interruptions.
csl_mcbsp.h → To use the serial communication.
csl_edma.h → To use the EDMA peripheral.
dsk6713.h → To use all the intern peripherals.
dsk6713_aic23.h → To use and configure the audio codec
math.h → To implement the mathematical operations.
rtdx.h → To implement the functions for RTDX communication between the PC and DSP.
tw_radix2.h → To implement the methods which are used to calculate the coefficients of the FFT radix2.
bitReverse.h → To implement the methods which are used to calculate the coefficients for ordering the FFT result.

Constant statement

In this section, the constants used in the main program are defined.
These constants are values which cannot be modified. Therefore, they are usually used to configure all functions automatically.
To define these values, it puts "#define" at the beginning, then a reference to this element and ultimately its value. It is not necessary put ";" at the end of line.

Because there are many constants, they are shown directly in the annexes.
It only gets one of them to see the definition structure: `#define PI 3.141592653589793`

Global variables

This section defines all the global variables/arrays which are used throughout the main program. These variables/arrays are constantly updated to perform the various functions in the main code. To define a variable, it puts the type first and then its name to use it. Later all these variables are initialized before the infinite loop.

Due to the large number of variables, it puts directly in the annexes.
It only gets a couple of them to see the definition structure:

```
short flagInterrupt;
short inputSamples[PASS];
```

Definition and configuration of audio codec and RTDX channels

- **Audio codec**

To activate and use correctly the A/D and D/A converters from the audio codec, which are controlled by the DSP, it has to configure properly the specific registers in the DSP.

First of all a handle is created to control the audio codec:

```
DSK6713_AIC23_CodecHandle hCodec; //Handle for the AIC23 códec
```

When the handle is created, it has to configure its working mode by creating a configuration variable. This variable defines the volume, samplig frequency, etc.

The following picture shows the different registers to configure this configuration variable:

ADDRESS	REGISTER
0000000	Left line input channel volume control
0000001	Right line input channel volume control
0000010	Left channel headphone volume control
0000011	Right channel headphone volume control
0000100	Analog audio path control
0000101	Digital audio path control
0000110	Power down control
0000111	Digital audio interface format
0001000	Sample rate control
0001001	Digital interface activation
0001111	Reset register

The manufacturer (Texas Instruments) gives all the information about this topic in the **TLV320AIC23B datasheet**, which is defined in the bibliography section.
The datasheet explains the meaning of each register and all the possible values to set in them.

For this project, according the last document, the optimal configuration for the audio codec is:

```
DSK6713_AIC23_Config config = {
    0x0017,
    0x0017,
    0x01F9,
    0x01F9,
    0x0011,
    0x0000,
    0x0000,
    0x0043,
    0x0001,
    0x0001
};
```

Left line input channel volume control: The input volume of the left channel is set to 0dB without mute.

Right line input channel volume control: The input volume of the right channel is set to 0dB without mute.

Left channel headphone volume control: The headphone volume of the left channel is set to 0dB.

Right channel headphone volume control: The headphone volume of the right channel is set to 0dB.

Analog audio path control: Microphone boost is set to 20dB and the DAC is selected.

Digital audio path control: It doe so't use the digital control.

Power down control: Enable all the peripherals and I/O to be used.

Digital audio interface format: Master mode and frame sync followed by two data words (DSP format).

Sample rate control: Frequency Sample I/O to 48Hz.

Digital interface activation: The digital interface is activated.

Reset register: It does not use (in fact, it is not necessary to put it in the configuration variable).

- **RTDX channels**

This library provides the data types and functions for:
- Sending data from the target to the host
- Sending data from the host to the target

The following data types and functions are defined in the header file rtdx.h. They are available via DSP/BIOS or standalone.

Declaration Macros
— RTDX_CreateInputChannel
— RTDX_CreateOutputChannel

Functions
— RTDX_channelBusy
— RTDX_disableInput
— RTDX_disableOutput
— RTDX_enableOutput
— RTDX_enableInput
— RTDX_read
— RTDX_readNB
— RTDX_sizeofInput
— RTDX_write

Macros
— RTDX_isInputEnabled
— RTDX_isOutputEnabled

With the upper functions, it is possible configure the CCStudio to enable the RTDX communication between the DSP app and an external device (in this case, the user interface).

Therefore, 2 RTDX channels are created for the communication between the PC and VC++:

```
//Create the channels for the communication between DSP and PC
RTDX_CreateInputChannel(RTDXinput);
RTDX_CreateOutputChannel(RTDXoutput);
```

RTDX_CreateInputChannel(RTDXinput): A RTDX channel called RTDXinput is created to receive the data from a external device (in this case, from the VC++ interface).

RTDX_CreateOutputChannel(RTDXoutput): A RTDX channel called RTDXoutput is created to send the treated data from a external device (in this case, from the VC++ interface).

With these channels and the RTDX protocol configured (the configuration is performed in the user interface), the communication is enabled to request or receive data from one platform to another.
The sending and receiving actions are performed in the infinite loop, inside the main method.

Definition and configuration of EDMA channels

To read the input data from the codec, update the input array and take out the treated data to the speakers, it uses the EDMA peripheral to move all that data continuously. Hence it subtracts a lot of work to the DSP app.

The channels performed for these tasks are:

```
EDMA_Handle hEdmaRead;
EDMA_Handle hEdmaWrite;
EDMA_Handle hEdmaUpdateInput;
```

hEdmaRead: EDMA channel used for move the input data from the codec to the buffer.
hEdmaWrite: EDMA channel used for move the output data from the buffer to the codec.
hEdmaUpdateInput: EDMA channel used for move the input data from the buffer to an array.

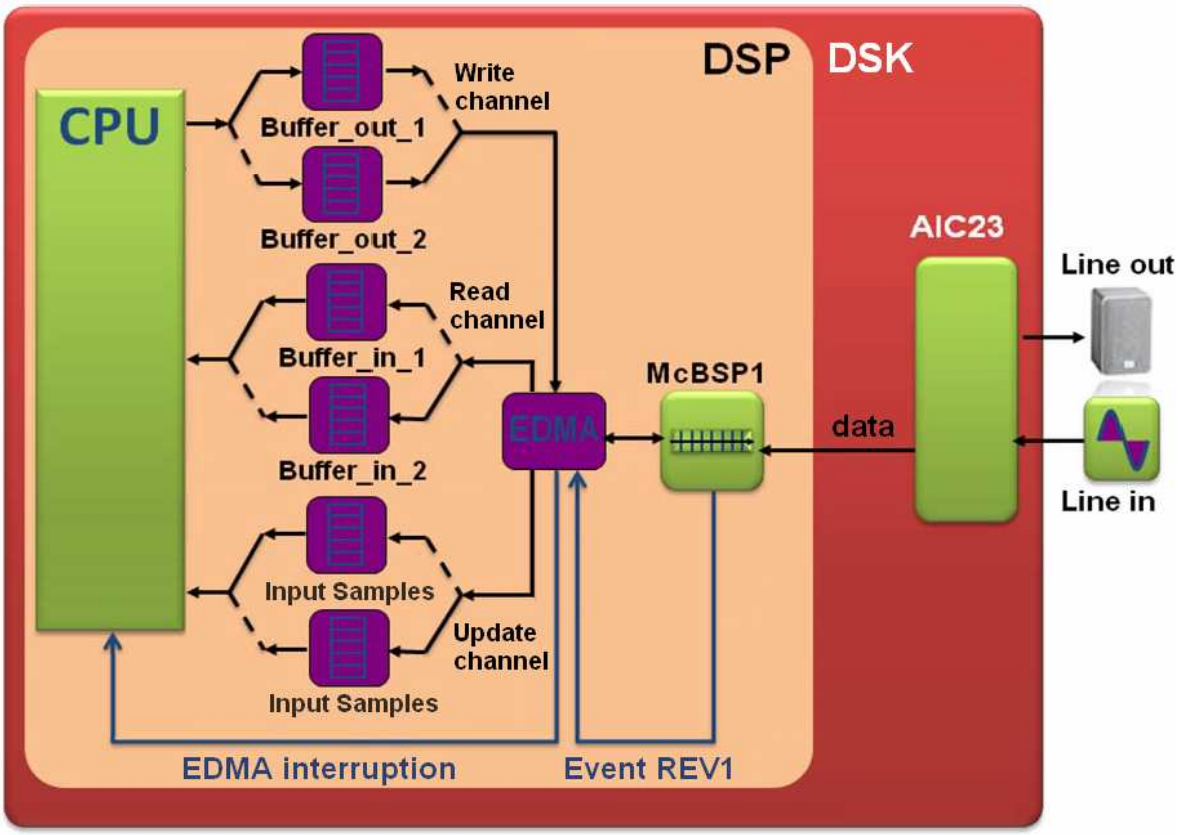
In addition of those channels, 6 reload channels are created to change the configuration (i.e. the performance) of the previous EDMA channels.

```
EDMA_Handle hEdmaLINKread1, hEdmaLINKread2;
EDMA_Handle hEdmaLINKwrite1, hEdmaLINKwrite2;
EDMA_Handle hEdmaLINKupdateInput1, hEdmaLINKupdateInput2;
```

hEdmaLINKread1: EDMA reload channel used for change the hEdmaRead configuration.
hEdmaLINKread2: EDMA reload channel used for change the hEdmaRead configuration.
hEdmaLINKwrite1: EDMA reload channel used for change the hEdmaWrite configuration.
hEdmaLINKwrite2: EDMA reload channel used for change the hEdmaWrite configuration.
hEdmaLINKupdateInput1: EDMA reload channel used for change the hEdmaUpdateInput configuration.
hEdmaLINKupdateInput2: EDMA reload channel used for change the hEdmaUpdateInput configuration.

The EDMA is configured to take the input data from the codec firstly (hEdmaRead). Its 2 reload channels (for stereo input) configure the input channel which it is in use.
Consequently, it updates the input array with these data to treat them (hEdmaUpdateInput).
Finally, it takes out the treat data to the codec (hEdmaWrite). Its 2 reload channels (for stereo output) configure the output channel which it is using.

The following picture shows the configured data transfer between the audio codec and DSP by the EDMA:



The following picture shows the different registers to configure these reload channels:

Acronym	Parameter Name
OPT	EDMA channel options parameter
SRC	EDMA channel source address parameter
CNT	EDMA channel transfer count parameter
DST	EDMA channel destination address parameter
IDX	EDMA channel index parameter
RLD	EDMA channel count reload/link address parameter

The manufacturer (Texas Instruments) gives all the information about this topic in the **EDMA controller datasheet**, which is defined in the bibliography section.
The datasheet explains the meaning of each register and all the possible values to set in them.

The configuration of the reload channels are very long, therefore it only shows one of them.
For this project, according the last document, the optimal configuration of this reload channel is:

```
EDMA_Config edma_Config_Read1={
  EDMA_OPT_RMK(
    EDMA_OPT_PRI_HIGH,
    EDMA_OPT_ESIZE_16BIT,
    EDMA_OPT_2DS_NO,
    EDMA_OPT_SUM_NONE,
    EDMA_OPT_2DD_NO,
    EDMA_OPT_DUM_INC,
    EDMA_OPT_TCINT_YES,
    EDMA_OPT_TCC_OF(9),
    EDMA_OPT_LINK_YES,
    EDMA_OPT_FS_NO
   ),
  EDMA_SRC_OF(McBSP1_DRR),
  EDMA_CNT_OF(2*WIN_STEP),
  EDMA_DST_OF(buffer_in_1),
  EDMA_IDX_OF(0),
  EDMA_RLD_OF(0)
 };
```

EDMA_OPT_RMK: OPTIONS
- **EDMA_OPT_PRI_HIGH:** High priority
- **EDMA_OPT_ESIZE_16BIT:** Data length to read
- **EDMA_OPT_2DS_NO:** One dimension origin (the element is inside of a frame)
- **EDMA_OPT_SUM_NONE:** Static direction. It always reads from the same place
- **EDMA_OPT_2DD_NO:** One dimension destiny (the element is inside of a frame)
- **EDMA_OPT_DUM_INC:** It increases one memory position in buffer when data is read.
- **EDMA_OPT_TCINT_YES:** It enables the chain to connect with the next channel.
- **EDMA_OPT_TCC_OF (9):** When buffer is full, it pass to the channel 9.
- **EDMA_OPT_LINK_YES:** It allows the linker. It links channel read1 with channel read2.
- **EDMA_OPT_FS_NO:** The channel is synchronized by element.

EDMA_SRC_OF (McBSP1_DRR): SOURCE: DRR MCBSP1
EDMA_CNT_OF (2*WIN_STEP): LENGHT: 64 positions.
EDMA_DST_OF (buffer_in_1): DESTINATION: buffer_in_1.
EDMA_IDX_OF (0): INDEX: It doesn't use. It leaves to 0.
EDMA_RLD_OF (0): RELOAD: link. It puts to 0 because later it will configure the reload.

According the upper configuration variable, the reload channel has high priority. It only moves a byte from a static direction to a dinamic direction.
The pointer of the dinamic direction increases its position one unity in each movement of data until 64 posistions (the double of WIN_STEP).

When this read channel finishes its work, it pass to the input update channel.

The rest of reload channels are shown and explained in the annexes.

Main method

The main method is divided into two parts: the initialization of variables and the infinite loop.

Initializing variables

The first part is to initialize all the variables, arrays and handles to be used later. This section is only executed once after the code runs.

It also fills the arrays with the necessary coefficients for the FFT calculation. The interruptions are defined too.

Due to it exists a lot of variables, arrays and handles to initialize, here it is only shows some of them. The rest it can see in the annexes.

```
delayedSampleCF = 0;
pointerFreqCFT = 0;

for (i=0; i<WIN_STEP; i++) {inputSamples[i]=0;}
for (i=0; i<WIN_STEP; i++) {inputEffect[i]=0;}

for (i=0; i<delayMax; i++) {delayEchoReverb[i]=0;}
for (i=0; i<WIN_STEP; i++) {reflections[i]=0;}

for (i=0; i<15; i++) {ArrayRTDX[i]=0;}
readFreq = 0;

for (i=0; i<WIN_LENGTH; i++) {coefRadix2[i]=0;}
for (i=0; i<longCoefBitRev; i++) {coefBitRev[i]=0;}
gen_twiddle_r2(coefRadix2, WIN_LENGTH, scale);
bitrev_index(coefBitRev, WIN_LENGTH);

CSL_init();
DSK6713_init();
hCodec = DSK6713_AIC23_openCodec(0, &config);
edma_init();
```

Infinite loop

The infinite loop treats continuously the received data in function of the selected options from the graphical interface.
It also receives the variable values from the VC++ interface.

For a optimal configuration of the tuner implementation, when it uses this one, the sampling frequency is changed from 48Hz to 8Hz to ensure a good resolution of the calculated frequency.

The audio codec also incorporates a Nyquist filter to avoid that frequencies bigger than the half of the sampling frequency (which it is 8000Hz when the tuner is enabled) are taken to ensure input samples with a good quality.
Therefore, the maximum value which the tuner can detect is 4000Hz (8000Hz / 2).

Here it can see a part of this section. The rest can be seen in the annexes.

```
if(!RTDX_channelBusy(&RTDXinput)) {
    RTDX_readNB(&RTDXinput, &ArrayRTDX, sizeof(ArrayRTDX));}

if (ArrayRTDX[13]==1) {DSK6713_AIC23_rset(hCodec, 8, 0x0009);}
if (ArrayRTDX[13]==0) {DSK6713_AIC23_rset(hCodec, 8, 0x0001);}

ByPass();
if (ArrayRTDX[10] == 0) {
    //Gain effects
    if (ArrayRTDX[9] == 1) {Overdrive();}
    if (ArrayRTDX[9] == 2) {Distortion();}
    if (ArrayRTDX[9] == 3) {Fuzz();}
    //Dyanmic range effects
    if (ArrayRTDX[3] == 1) {Compressor();}
    if (ArrayRTDX[3] == 2) {Expansor();}
    if (ArrayRTDX[3] == 3) {NoiseGate();}
    if (ArrayRTDX[3] == 4) {AutoWah();}
    if (ArrayRTDX[3] == 5) {Panning();}
} else {Tuner();}

buffer_out_1[2*i] = outputEffect1[i];
buffer_out_1[2*i+1] = outputEffect2[i];
```

Routine to initialize the EDMA

To use the EDMA peripheral, firstly it has to initialize the channels.
In this method, the channels are configured and it loads in them the corresponding reload channels in each cycle. Therefore, this method is only executed once before enter to the infinite loop.

This routine can be seen directly in the annexes.

Service routine for the EDMA interrupt

With the EDMA interrupt configured, it enters in this method every time that the audio codec gets any input data, executing its code inside.

Inside the method, it resets the interruption to prepare it for the next time. Then it activates a flag which serves to enter to the infinite loop.
If the infinite loop is only executed when the input data is received, it subtracts a lot of work to the DSP.

```
void EDMA_HWI(void) {
    EDMA_intClear(9);     //Bring down the interrupt petition
    flagInterrupt = 1;    //Flag which indicates when it has
}                         //entered to the EDMA interrupt
```

Effects programming

This section contains the algorithms which simulate the previously described effects used in this project.

Some algorithms are more robust/complex than others, but anyway they have been thought to obtain a result as close as possible to the original analog effect in the simplest possible way.

In turn, these effects have been designed with an external variable (from the interface in Visual C++) which changes significantly the final result. It changes an essential parameter for each group of effects.

ByPass

This effect (in fact it is not a real effect, but it has scheduled like it was one more) is the first one in the effect chain and it is always executed. No matter the previously selected configuration from the VC++ interface.

Its only function is to pass the information from the input array to the output arrays to use them later with other effects if they are selected, or directly take them out to the codec.

The implementation code is:

```
for (i=0; i<WIN_STEP; i++) {
    outputEffect1[i] = inputSamples[i];
    if (outputEffect1[i]>30000) {outputEffect1[i] = 30000;}
    if (outputEffect1[i]<-30000) {outputEffect1[i] = -30000;}
    outputEffect2[i] = outputEffect1[i];
}
```

At each position of the received input array (inputSamples[i]) is processed as follows:

- It updates the value of the output arrays with the input array value.
- The output values are limited for security.

Gain effects

The gain effects (Overdrive, Distortion and Fuzz) are achieved playing with the output gain, determined by a serie of predefined equations.

Overdrive

The overdrive effect produces a low distortion.
For overdrive simulations a soft clipping of the input values has to be performed.

This effect is simulated according the following equation:

$$f(x) = \begin{cases} 2x & \text{for} \quad 0 \leq x \leq 1/3 \\ \frac{3-(2-3x)^2}{3} & \text{for} \quad 1/3 \leq x \leq 2/3 \\ 1 & \text{for} \quad 2/3 \leq x \leq 1 \end{cases}$$

Taking the previous equation, the implementation code is:

```
float thresholdOver;
float saturOverdrive = -100 * ArrayRTDX[7] + 20000;
float absInput = 0;
thresholdOver = saturOverdrive/320;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i] / 32000;
    absInput = fabs(inputEffect[i])*100;
    if (absInput<(thresholdOver/3))
        {outputEffect1[i] = 2*inputEffect[i];}
    if ((absInput>=(thresholdOver/3))
        && (absInput<(thresholdOver*2/3))) {
            if (inputEffect[i]>0) {
                outputEffect1[i] =
                    (3-(pow((2-(3*inputEffect[i])),2)))/3;
            }
            if (inputEffect[i]<0) {
                outputEffect1[i] =
                    -(3-(pow((2-(3*fabs(inputEffect[i]))),2)))/3;
            }
    }
    if (absInput>=(thresholdOver*2/3)) {
        if (inputEffect[i]>0) {outputEffect1[i] = 1;}
        if (inputEffect[i]<0) {outputEffect1[i] = -1;}
    }
    outputEffect1[i] = outputEffect1[i] * 32000;
    if (outputEffect1[i]>25000) {outputEffect1[i] = 25000;}
    if (outputEffect1[i]<-25000) {outputEffect1[i] = -25000;}
    outputEffect2[i] = outputEffect1[i];
}
```

It takes the threshold value according the external value of the gain effect from the slider in the VC++ interface (saved in ArrayRTDX [7]).
The auxiliar variable is defined.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- If the absolute value of the input is less than 1/3 the threshold, the output is the double of the input.
- If the absolute value of the input is between 1/3 and 2/3 threshold, the output is obtained in function the above described equation.
- If the absolute value of the input is greater than 2/3 of the threshold, it shows directly the threshold (saturation) value.
- The oputput value is limited for security.
- Finally the output arrays are updated.

Distortion

The distortion effect produces a medium ditortion, perfect for the rock/metal music.

A nonlinearity suitable for the simulation of distortion is given by:

$$f(x) = \mathrm{sgn}(x)\left(1 - e^{-|x|}\right)$$

Taking the previous equation, the implementation code is:

```
short gainDist = 0.15 * ArrayRTDX[7] + 5;
float distValue = 0;
float distortedSignal = 0;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    distValue = inputEffect[i] * gainDist / 32768;
    distortedSignal = (distValue*32768)/fabs(distValue*32768)
                        * (1-exp(-fabs(distValue)));
    outputEffect1[i] = 0.6*distortedSignal*32768+0.6*inputEffect[i];
    if (outputEffect1[i]>25000) {outputEffect1[i] = 25000;}
    if (outputEffect1[i]<-25000) {outputEffect1[i] = -25000;}
    outputEffect2[i] = outputEffect1[i];
}
```

It takes the output gain value according the external value of the gain effects from the slider in the VC++ interface (saved in ArrayRTDX [7]).
The auxiliar variables are defined.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The input sample is multiplied by the received output gain and it leaves in percentage 1%.
- It calculates the distorted sample according the previous equation.
- The output value is limited for security.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.

Fuzz

The fuzz effect clips the sound wave until it is nearly a square wave, resulting in a heavily distorted or "fuzzy" sound.

A non-linear function commonly used to simulate distortion/fuzz is given by:

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

Taking the previous equation, the implementation code is:

```
short gainFuzz = 0.10 * ArrayRTDX[7] + 5;
float fuzzValue = 0;
float fuzzedSignal = 0;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    fuzzValue = inputEffect[i] * gainFuzz / 32768;
    fuzzedSignal = (fuzzValue*32768)/fabs(fuzzValue*32768) *
                (1-exp((0.7*fuzzValue*fuzzValue)/fabs(fuzzValue)));
    outputEffect1[i] = 0.5*fuzzedSignal*32768+0.6*inputEffect[i];
    if (outputEffect1[i]>25000) {outputEffect1[i] = 25000;}
    if (outputEffect1[i]<-25000) {outputEffect1[i] = -25000;}
    outputEffect2[i] = outputEffect1[i];
}
```

It takes the output gain value according the external value of the gain effects from the slider in the VC++ interface (saved in ArrayRTDX [7]).
The auxiliar variables are defined.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The input sample is multiplied by the received output gain and it leaves in percentage 1%.
- It calculates the super distorted sample according the previous equation.
- The output value is limited for security.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.

Dynamic range effects

The dynamic range effects (Compressor, Expander, Noise Gate, AutoWah and Panning) are achieved by altering parameters in them, very different to each other.
The aim is increases/decreases/alters the dynamic range in which they work.

Compressor

The compression effect is achieved by elevating the input with a exponent lower than 1 to reduce its dynamic range and compress the signal.

The compressor behaviour follows this equation:

$$f(x) = x^{\text{comp}} \qquad 0.6 < \text{comp} < 1$$

Taking the previous equation, the implementation code is:

```
float compLevel = -0.004*ArrayRTDX[2] + 1;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    auxVarCompExp = pow (inputEffect[i], compLevel);
    outputEffect1[i] = auxVarCompExp;
    if (outputEffect1[i]>30000) {outputEffect1[i]=30000;}
    if (outputEffect1[i]<-30000) {outputEffect1[i]=-30000;}
    outputEffect2[i] = outputEffect1[i];
}
```

It takes the elevated value according the external value of the dynamic range effects from the slider in the VC++ interface (saved in ArrayRTDX [2]).

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The input sample is elevated to the received value (which is between 0.6 and 1) to compress the signal.
- The output value is limited for security.
- Finally the output arrays are updated.

Expander

The expansor effect is achieved by elevating the input with a exponent higher than 1 in order to increase its dynamic range and expand the signal.

The compressor behaviour follows this equation:

$$f(x) = x^{\text{exp}} \qquad 1 < \text{exp} < 1.1$$

Taking the previous equation, the implementation code is:

```
float expLevel = 0.001*ArrayRTDX[2] + 1;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    auxVarCompExp = pow (inputEffect[i], expLevel);
    outputEffect1[i] = auxVarCompExp;
    if (outputEffect1[i]>30000) {outputEffect1[i]=30000;}
    if (outputEffect1[i]<-30000) {outputEffect1[i]=-30000;}
    outputEffect2[i] = outputEffect1[i];
}
```

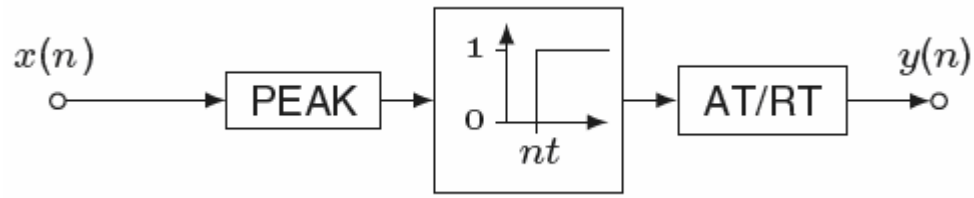
It takes the elevated value according the external value of the dynamic range effects from the slider in the VC++ interface (saved in ArrayRTDX [2]).

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The input sample is elevated to the received value (which is between 1 and 1.1) to expand the signal.
- The output value is limited for security.
- Finally the output arrays are updated.

Noise Gate

The Noise Gate effect deleted the information under a predifined threshold.
Its behaviour is described with the following diagram:



Taking the previous diagram, the implementation code is:

```
int auxNoiseGate = 0;
if (ArrayRTDX[9]==0)
    {thresholdNoiseGate = 150*ArrayRTDX[2] + 100;}
else {thresholdNoiseGate = 330*ArrayRTDX[2] + 500;}

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    if (fabs(inputEffect[i])<thresholdNoiseGate)
        auxNoiseGate++;
}
for (i=0; i<WIN_STEP; i++) {
    if (auxNoiseGate>=(WIN_STEP/2)) {outputEffect1[i]=0;}
    else {outputEffect1[i] = inputEffect[i];}
    outputEffect2[i] = outputEffect1[i];
}
```

It takes the threshold value according the external value of the dynamic range effects from the slider in the VC++ interface (saved in ArrayRTDX [2]).

The range of the threshold changes if any gain effect is activated.

The auxiliar variable is defined.

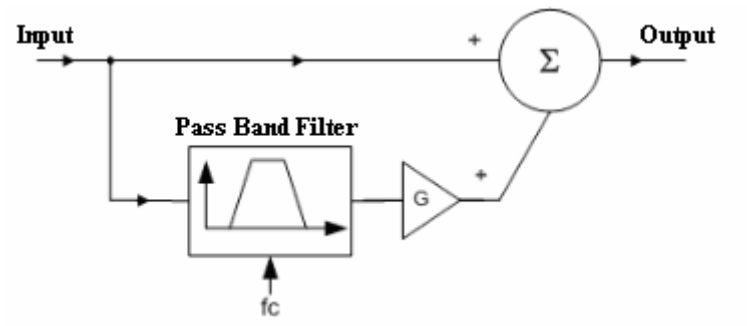
At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- It counts how many samples of the input array which are under the threshold value.
- If more than half of the array values are under the threshold, all the output array is 0.
- Finally the output arrays are updated.

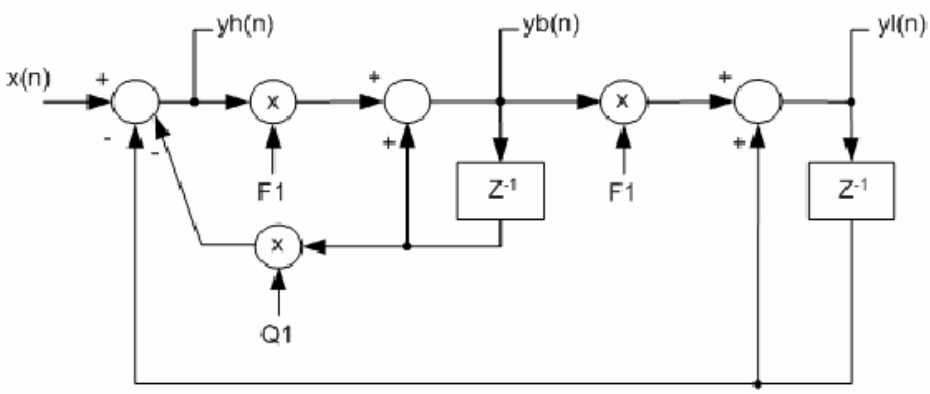
Auto Wah

The Auto Wah effect is generated with a variable band pass filter automatically in time with a LFO (Low Frequency Oscillator).

This effect consist in add to the input signal the same filtered signal with a band pass filter, which the cut frequency is changed temporally, increasing it or decreasing it.



For its implementation, the most used algorithm is the use of state variable filters:



$$y_l(n) = F_1 \cdot y_b(n) + y_l(n-1)$$
$$y_b(n) = F_1 \cdot y_h(n) + y_b(n-1)$$
$$y_h(n) = x(n) - y_l(n-1) - Q1 \cdot y_b(n-1)$$
$$Q1 = 0{,}1$$
$$F1 = 2\sin\left(\frac{\pi f_c}{f_s}\right)$$

Taking the previous diagram and equations, the implementation code is:

```
short auxWah;
int oscillationWah = -320*ArrayRTDX[2] + 64000;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    FCvariableWah = 850*cos(2*PI*FCpointerWah/oscillationWah)+950;
    F1wah = 2*sin(PI*FCvariableWah/FS);
    auxWah = i-1;   if (auxWah<0) {auxWah = auxWah + WIN_STEP;}
    highPassFilter[i] = inputEffect[i] - lowPassFilter[auxWah]
                        - 0.1*bandPassFilter[auxWah];
    bandPassFilter[i] = F1wah*highPassFilter[i]
                        + bandPassFilter[auxWah];
    lowPassFilter[i] = F1wah*bandPassFilter[i]
                        + lowPassFilter[auxWah];
    if (highPassFilter[i]>=25000)   {highPassFilter[i]=0;}
    if (bandPassFilter[i]>=25000)   {bandPassFilter[i]=0;}
    if (lowPassFilter[i]>=25000)    {lowPassFilter[i]=0;}
    outputEffect1[i] = 1*bandPassFilter[i] + 0.3*inputEffect[i];
    outputEffect2[i] = outputEffect1[i];
    FCpointerWah++;
    if (FCpointerWah>=oscillationWah) {FCpointerWah=0;}
}
```

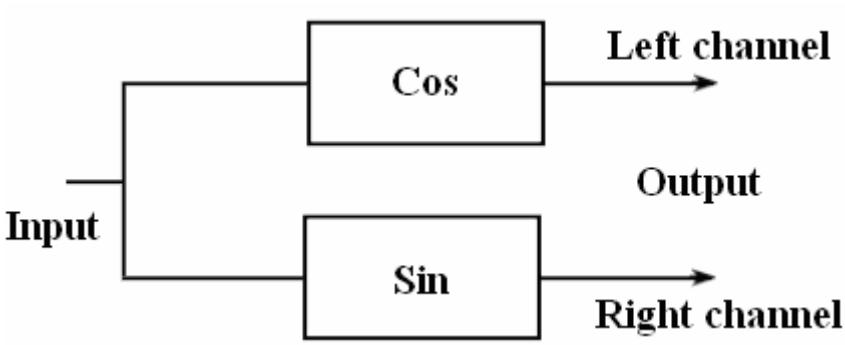
It takes the frequency value according the external value of the dynamic range effects from the slider in the VC++ interface (saved in ArrayRTDX [2]).
The auxiliar variable is defined.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The cut frequency is updated (between a predefined range) with a cosinus signal (LFO) which frequency is the received value from outside.
- With the previous cut frequency, one parameter for the state variable filter is updated.
- It calculates the value of the 3 filters (low pass filter, high pass filter and the band pass filter), which depend on each other, its previous values and input samples.
- The auxiliar variable is used to implement a circular buffer for the next step.
- If the filter values are higher than the maximum value, they are resetted.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.
- It increments the used pointer and resetted if it achieves the maximum value.

Panning

The Panning effect moves the sound from one output channel to the another one with a LFO (Low Frequency Oscillator).
For this effect, the only thing which has to do is introduce the horizontal component of the input signal in one output channel (Left or Right) and the vertical component in the another channel.



The implementation code is:

```
int freqPanning = -1440*ArrayRTDX[2] + 192000;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    outputEffect1[i] =
        inputEffect[i]*cos(2*PI*pointerPanning/freqPanning);
    outputEffect2[i] =
        inputEffect[i]*sin(2*PI*pointerPanning/freqPanning);
    pointerPanning++;
    if (pointerPanning>=freqPanning) {pointerPanning=0;}
}
```

It takes the frequency value according the external value of the dynamic range effects from the slider in the VC++ interface (saved in ArrayRTDX [2]).

At each position of the received input array (inputEffect[i]) is processed as follows:

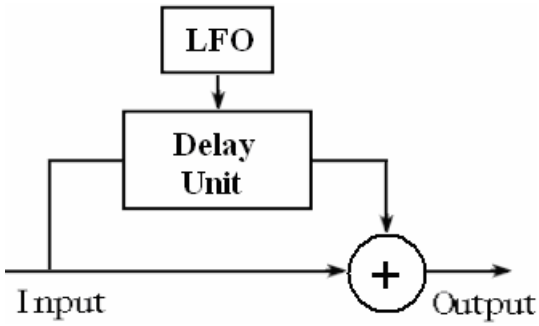
- It updates the input effect array with the actual sample.
- One output channel (left or right) is updated by multiplying the input array with a cosinus signal (LFO opposite to the sinus signal). The used frequency is defined with the external value. It also determines the output gain.
- The another output channel (left or right) is updated by multiplying the input array with a sinus signal (LFO opposite to the cosinus signal). The used frequency is defined with the external value. It also determines the output gain.
- It increments the used pointer and resetted if it achieves the maximum value.

Modulation effects

The modulation effects (Chorus, Flanger and Tremolo) are achieved playing with previous samples of a variable delay by a LFO. This LFO modulates the signal under specific circumstances.

Chorus

The chorus effect plays with delayed samples (using a LFO) to simulate the mix of 2 inputs with the same information, but without perfect (but constant) synchronization between them.



Taking the previous diagram, the implementation code is:

```
int oscillationChorus = -1600*ArrayRTDX[0] + 192000;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    delayedSampleCF = delay5ms *
            cos(2*PI*pointerFreqCFT/oscillationChorus)+delay25ms;
    indexChorus = pointerChorus - delayedSampleCF;
    if (indexChorus<0) {indexChorus = indexChorus + delay30ms;}
    outputEffect1[i] = 0.8*chorusSamples[indexChorus]
                       + 0.5*inputEffect[i];
    outputEffect2[i] = outputEffect1[i];
    chorusSamples[pointerChorus] = inputEffect[i];
    pointerChorus++;
    pointerFreqCFT++;
    if (pointerChorus>=delay30ms) {pointerChorus=0;}
    if (pointerFreqCFT>=oscillationChorus) {pointerFreqCFT=0;}
}
```

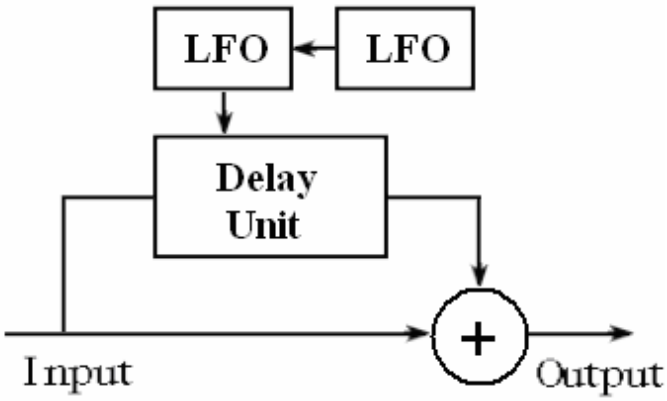
It takes the frequency value according the external value of the modulation effects from the slider in the VC++ interface (saved in ArrayRTDX [0]).

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- It takes a variable delayed value (with a LFO) between 10ms and 30ms.
- It subtracts the previous value to the actual pointer to achieve the desired array index.
- If this array index is negative, it adds the maximum value to simulate a circular buffer.
- With the final value of the array index, it takes the delayed sample from the chorus array.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.
- The actual input sample is saved in the chorus array to update it.
- It increments the used pointers and they are resetted if they achieve the maximum value.

Flanger

Similar to the chorus effect, the flanger effect plays with delayed samples (using a LFO which changes its frequency periodically with another LFO) to simulate the mix of 2 inputs with the same information but with variable imperfect synchronization between them.



Taking the previous diagram, the implementation code is:

```
int oscillationFlanger = -360*ArrayRTDX[0] + 48000;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    freqFlanger = (48000-12000)/2 *
        cos(2*PI*pointerFreqCFT/oscillationFlanger)+(48000+12000)/2;
    delayedSampleCF = delay5ms *
        cos (2*PI*(freqFlanger-12000)/(48000-12000)) + delay5ms;
    indexFlanger = pointerFlanger - delayedSampleCF;
    if (indexFlanger<0) {indexFlanger = indexFlanger + delay10ms;}
    outputEffect1[i] = 0.8*flangerSamples[indexFlanger]
                       + 0.5*inputEffect[i];
    outputEffect2[i] = outputEffect1[i];
    flangerSamples[pointerFlanger] = inputEffect[i];
    pointerFlanger++;
    pointerFreqCFT++;
    if (pointerFlanger>=delay10ms) {pointerFlanger=0;}
    if (pointerFreqCFT>=oscillationFlanger) {pointerFreqCFT=0;}
}
```

It takes the frequency value according the external value of the modulation effects from the slider in the VC++ interface (saved in ArrayRTDX [0]).

At each position of the received input array (inputEffect[i]) is processed as follows:

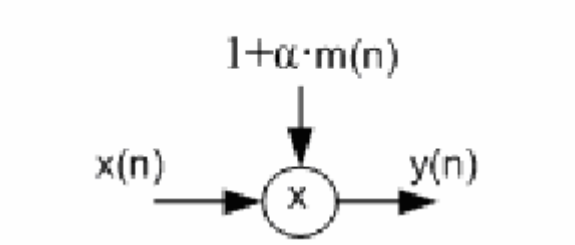
- It updates the input effect array with the actual sample.
- It takes a variable frequency value (with a LFO) between 1Hz and 4Hz.
- It takes a variable delayed value (with another LFO) between 0ms and 10ms.
- It subtracts the previous value to the actual pointer to achieve the desired array index.
- If this array index is negative, it adds the maximum value to simulate a circular buffer.
- With the final value of the array index, it takes the delayed sample from the flanger array.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.
- The actual input sample is saved in the chorus array to update it.
- It increments the used pointers and they are resetted if they achieve the maximum value.

Tremolo

The Tremolo effect simulates that the output gain change continuosly (with a LFO) all the time.

Its behaviour is described with the next diagram:

$$y(n) = (1 + \alpha m(n)) \cdot x(n)$$



Taking the previous diagram, the implementation code is:

```
float gainTremolo = 0;
int oscillationTremolo = -400*ArrayRTDX[0] + 64000;

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    gainTremolo =
        cos (2*PI*pointerFreqCFT/oscillationTremolo);
    outputEffect1[i] = (1+gainTremolo)/2*inputEffect[i];
    outputEffect2[i] = outputEffect1[i];
    pointerFreqCFT++;
    if (pointerFreqCFT>=oscillationTremolo)
        {pointerFreqCFT=0;}
}
```

It takes the frequency value according the external value of the modulation effects from the slider in the VC++ interface (saved in ArrayRTDX [0]).

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- It takes a variable value (with a LFO) of output gain.
- The previous oscillatory gain is multiplied with the actual input sample according the upper equation.
- Finally the output arrays are updated.
- It increments the used pointer and it is resetted if it achieves the maximum value.

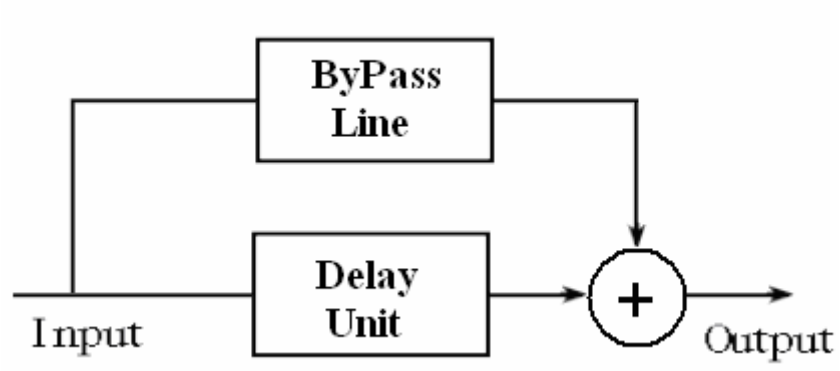
Repetition effects

The modulation effects (Delay/Echo and Reverb) are achieved using delayed samples (sometimes many of them at the same time with different distance from each other) and digital filters to simulate the acoustics of the rooms.

Delay/Echo

The Delay or Echo effect simulates the returned (and delayed) data which was bounced off in a wall.

To realice the Delay or Echo effects, it has to follow this diagram:



Taking the previous diagram, the implementation code is:

```
int fixDelayEcho = 60 * ArrayRTDX[4];
if (fixDelayEcho > delayMax) {fixDelayEcho = delayMax;}

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    if (fixDelayEcho==0) {outputEffect1[i] = inputEffect[i];}
    else {outputEffect1[i] = 0.7*delayEchoReverb[indexDelayEcho]
                            + 0.6*inputEffect[i];}
    outputEffect2[i] = outputEffect1[i];
    delayEchoReverb[indexDelayEcho] = inputEffect[i];
    indexDelayEcho++;
    if (indexDelayEcho>=fixDelayEcho) {indexDelayEcho=0;}
}
```

It takes the delayed time value (in samples) according the external value of the repetition effects from the slider in the VC++ interface (saved in ArrayRTDX [4]).
If this extern value is higher than the maximum possible value, this value is reduced to the maximum possible value.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- If the delayed time is 0, the output information is the same than the input infomation.
- If the delayed time is not 0, it takes the delayed sample according the received delay value.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.
- The actual input sample is saved in the Delay/Echo/Reverb array to update it.
- It increments the used pointer and it is resetted if it achieves the maximum value.

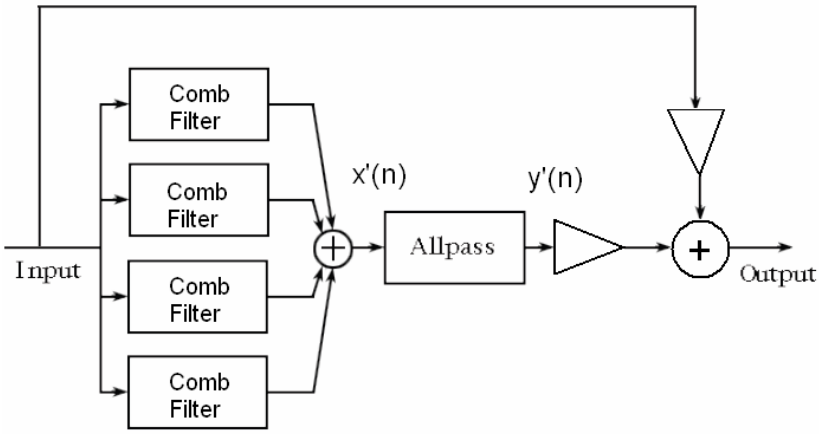
Reverb

The Reverb effect simulate the acoustics of a room with many delayed samples.

To facilitate this algorithm, it uses a fixed/static equation which simulates the acoustic of a room with nothing which would muffle the sound (without curtains, blankets...):

$$y'[n] = 0.4\, y'[n-1] - 0.2499\, y'[n-2] + 0.0441\, y'[n-3] + 0.5814\, x'[n-1] + 0.2142\, x'[n-2]$$

With this equation, it uses the Schroeder model (which uses the mix of many delayed samples with a IIR filter) to simulate the effect. Its diagram is:



Taking the previous diagram and equation, the implementation code is:

```
int fixReverb = 60 * ArrayRTDX[4];
short auxRev1, auxRev2, auxRev3;
if (fixReverb > delayMax) {fixReverb = delayMax;}

for (i=0; i<WIN_STEP; i++) {
    inputEffect[i] = outputEffect1[i];
    auxRev1=i-1;    if (auxRev1<0){auxRev1=auxRev1+WIN_STEP;}
    auxRev2=i-2;    if (auxRev2<0){auxRev2=auxRev2+WIN_STEP;}
    auxRev3=i-3;    if (auxRev3<0){auxRev3=auxRev3+WIN_STEP;}
    if (fixReverb==0) {outputEffect1[i] = inputEffect[i];}
    else {
        reflections[i] = 0.8*delayEchoReverb[indexReverb]
                        + 0.4*delayEchoReverb[indexReverb/2]
                        + 0.2*delayEchoReverb[indexReverb/3]
                        + 0.1*delayEchoReverb[indexReverb/4];
        if (reflections[i]>=25000) {reflections[i]=0;}
        IIRoutput[i] = 0.4*IIRoutput[auxRev1]
                        - 0.2499*IIRoutput[auxRev2]
                        + 0.0441*IIRoutput[auxRev3]
                        + 0.5814*reflections[auxRev1]
                        + 0.2142*reflections[auxRev2];
        outputEffect1[i]=0.7*IIRoutput[i]+0.7*inputEffect[i];
    }
    outputEffect2[i] = outputEffect1[i];
    delayEchoReverb[indexReverb] = inputEffect[i];
    indexReverb++;
    if (indexReverb>=fixReverb) {indexReverb=0;}
}
```

It takes the delayed time value (in samples) according the external value of the repetition effects from the slider in the VC++ interface (saved in ArrayRTDX [4]).
The auxiliar variables are defined.
If this extern value is higher than the maximum possible value, this value is reduced to the maximum possible value.

At each position of the received input array (inputEffect[i]) is processed as follows:

- It updates the input effect array with the actual sample.
- The auxiliar variables are used to implement circular buffers for the next steps.
- If the delayed time is 0, the output information is the same than the input infomation.
- If the delayed time is not 0:
  - It gets a sum of delayed samples in different moments (which simulate all the reflections). Each one has its specific gain (the most recent samples have the higher gain because these ones take less time to return to the origin).
  - The previous value is set to 0 if exceeds the maximum possible value.
  - The upper final value is processed inside a IIR filter, which uses the value of the reflections and their own previous values to generate the filtered output value.
- Finally the output arrays are updated by mixing a percentage of the modified signal and another percentage from the original signal.
- The actual input sample is saved in the Reverb array to update it.
- It increments the used pointer and it is resetted if it achieves the maximum value.

Tuner

This one is not a real effect, but it has been designed like one of them because it is activated or deactivated according to the user interface, like the other effects.

The tuner (on the DSP app) takes the input information and it performs the FFT to get the frequency spectrum of the input signal and to determine its main frequency.
Then the data is sent to the interface by RTDX and there it calculates the name of the note and its octave.

It is important to indicate that the detection range of this implementation is between 32 Hz and 4000 Hz.

The minimum frequency (32Hz) is due to the configuration of the window length and the window step, which determine the resolution according the sampling frequency.
32 Hz is a Do/C note in the first octave.

The maximum frequency (4000Hz) is due to the Nyquist filter which the codec incorporates to ensure a good quality of the input samples.
This means that the sampling frequency here is 8000 Hz.

According to the work in MATLAB, the algorithm is:

```
int specAmpMax = 0;
float position = 0;

for (i=0; i<WIN_STEP; i++) {
    outputEffect1[i] = 0;
    outputEffect2[i] = 0;
}
if (ArrayRTDX[14]==1) {
    for (readFreq=0; readFreq<5; readFreq++) {
        for (i=0; i<(WIN_LENGTH-WIN_STEP); i++)
            {inputSamples2[i] = inputSamples2[WIN_STEP+i];}
        for (i=0; i<WIN_STEP; i++)
            {inputSamples2[i+WIN_LENGTH-WIN_STEP]=inputSamples[i];}
        for (i=0; i<WIN_LENGTH; i++)
            {FFTsamples[2*i] = inputSamples2[i]/WIN_LENGTH;}

        DSP_radix2 (WIN_LENGTH, FFTsamples, coefRadix2);
        DSP_bitrev_cplx ((int*)FFTsamples, coefBitRev, WIN_LENGTH);

        for (i=0; i<WIN_LENGTH; i++) {
            absFFT[i] = FFTsamples[2*i]*FFTsamples[2*i] +
                        FFTsamples[2*i+1]*FFTsamples[2*i+1];
            FFTsamples[2*i+1]=0;
        }
        specAmpMax = 0;
        for (i=0; i<WIN_LENGTH/2; i++) {
            if (absFFT[i]>specAmpMax) {
                specAmpMax = absFFT[i];
                position = i;
            }
        }
        frequency = position * 8000/WIN_LENGTH;
    }
}
if (readFreq == 5) {
    RTDX_write(&RTDXoutput, &frequency, sizeof(frequency));
    ArrayRTDX[14] = 0;
    readFreq = 0;
}
```

The auxiliar variables are defined.
The output signal is muted.

If the a frequency value is solicited, during a predefined number of times:

- The data is prepared to use it for the FFT function.
- The FFT is performed (the result is disordered).
- The FFT result is ordered (these values have real and imaginary part).
- To ease the process, it takes the module of each sample.
- It takes the frequency value with the higher amplitude.

The frequency is calculated a predefined number of times (to ensure a real value).
This final value is sent to the interface to show it.

Microsoft Visual C++

There are two forms to schedule an user interface: the programming to graphic interface and the programming oriented to a text interface (console mode).

In the second one, the scheduler organizes in a sequential form the calculation instructions and the interaction with the user. But in the programming in a graphic interface, it is not defined the order which the user interacts with the program (selecting any option, changing text…).
Due to this, the form to organize a program to both environments is different.

In the console mode, the scheduler can intercalate calculation and user interaction sentences.
However, in the graphic environment it executes an infinite loop. It waits for an event (from user or system), it executes the code associated to that event, and then it returns to wait to the next one. The events can be everything: pressing a button from the mouse, pressing a key, selecting a menu option, creating a window, changing the window size, etc.
Furthermore, it cannot define the order of these events. It only depends on the user.

In the different events in graphic mode, it is important to understand which events are associated with the window. On the other hand, when it writes something in the console mode, it is permanent. In the graphic mode it is necessary to redraw the window completely when it is required. This requirement can come from the program or the system (for example, a new window inhibits the older one). This requirement is indicated with an event. Due to this, in the graphic mode is necessary that the program stores all the data to redraw the window content in any moment when it receives the event.

The graphic interface elements can be described and used easily like objects. It is normal to use the programming oriented to objects for the scheduling of these interfaces.
For this reason, it is decided to program a graphic interface using the Microsoft Visual C++ (VC++).

Introduction to Microsoft Visual C++

The development environment of VC++ 6.0 provides many possibilities to the programmer from creating applications for various formats and features, plus through the creation of DLL libraries, icons, bitmaps, cursors, windows, etc.

All this Integrated Development Environment (IDE) is a great tool to develop the most versatile applications for both, Windows and DOS, environments.

To realize the scheduling for windows and its events, the operating system provides many functions in libraries. That set of functions is called Application Programming Interface (API), and in Windows is called Software Development Kit (SDK). These functions serve to manage windows (create, resize, close, etc.) of different types (normal, menu, button, dialog, text, selection list, etc.), obtain events, perform draw actions, etc.

An API is a set of functions very extensive. It is necessary a lot of functions and very varied to manage the windows environment.
Furthermore, a lot of functionalities are very repetitive along the programs (create a main window with menu options, etc.), hence it requires a set of complex and heavy API, even for the easiest program.

To facilitate this job, Visual C++ provide another API, including a class library that encapsulates the most part of the complexity. It only leaves the task of the specific part of the program to the scheduler. In Visual C++, this class library is called Microsoft Foundation Classes (MFC).

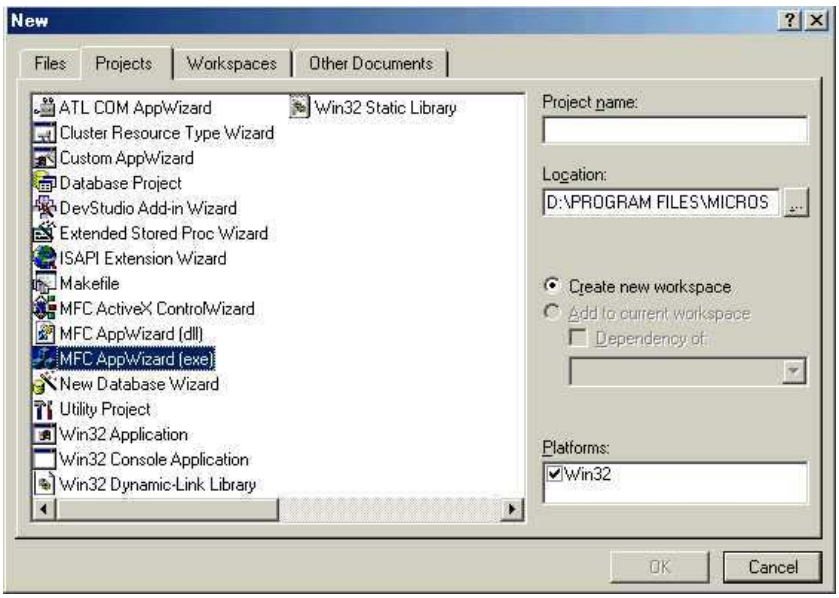
These classes have become a standard of development for Windows applications. They have facilitated a lot the programming in C++, for example to include a CString type data to declare strings. That does not exist in C.

In this way, for example, the creation of the main window is encapsulated in series of objects that create the program framework without adding any additional code line.

To help even more, the development environment sometimes disposes of utilities which allow to put in a graphic and intuitive way the interface elements (menus, buttons, text squares, etc.) and even links with the service functions of its events.
Of this manner, the scheduler can write less code (it saves time and errors). Furthermore, the part of the program which it can see is more clear and concise.

Creating a Visual C++ project

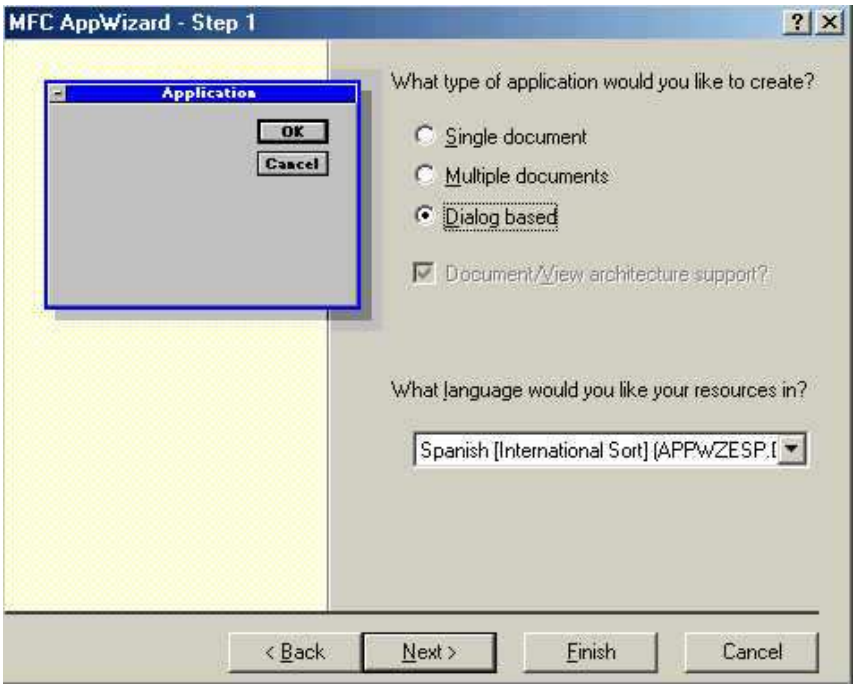
To create the interface, firstly it enters in the Visual C++ 6.0 program. Then it selects “NEW” from the “FILE” menu and the following screen appears:



This screen shows the different projects which can be developed with VC++. It selects “MFC AppWizard (exe)” to do a MFC app. The VC++ works with PROJECTS, and these ones are grouped into WORKSPACES.
Therefore it selects the name and the directory of this project.

After entering the name of the project, it presses the “OK” button. Then it starts the MFC Application Wizard which creates a default window.

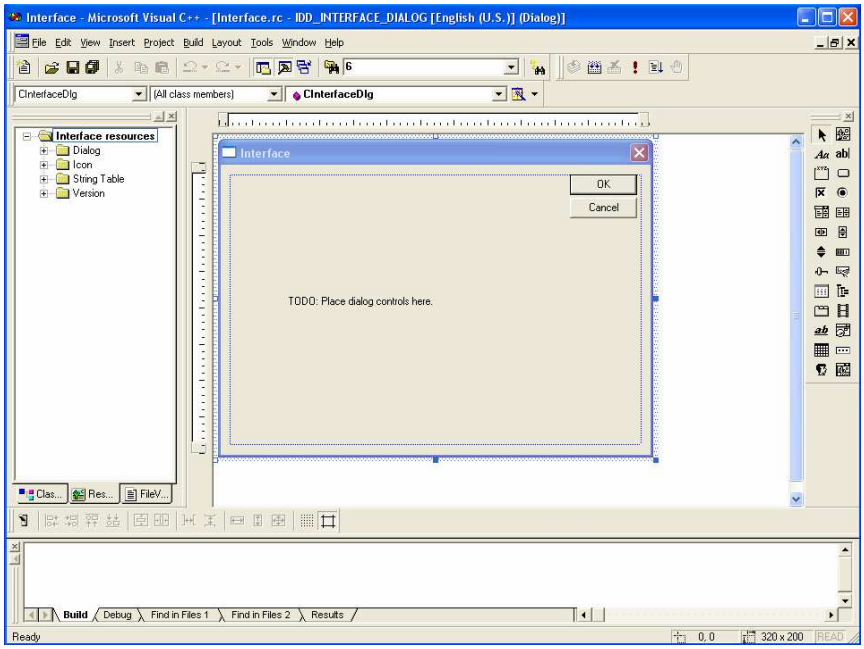
This process is shown in the following window:



This application can be Single Document Interface (SDI), Multiple Document Interface (MDI) or based on dialogues (Dialog Based). SDI is very similar to Dialog Based, except that by default VC++ adds menus. MDI is an application such as Word, with a main window that can contain many other inside (as a container).

This user interface is scheduled in a graphical way (with windows as classes). So it selects the Dialog Based option and after that it selects the correct language (in this case English).

Later, it presses “NEXT” to move the rest of screens (and “FINISH” for last one) until the following window is shown, and the app is created:



Now it can schedule a new program, in this case with the graphical user interface.

When a new program is scheduled, it must check if it has errors or warnings.
If it has any problem, it has to fix it. But if it has none, the code is compiled automatically.

To verify and compile the code, it clicks on the "Build" button in the top toolbar.

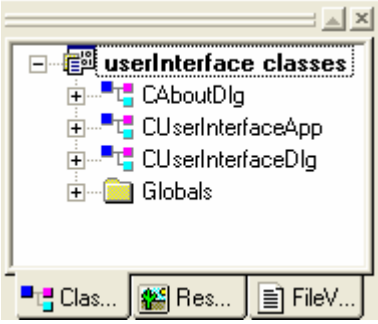
When the code has been verified, it proceeds to execute it with the "Execute program" button (which it is a red exclamation sign), next to previous button.

In fact, it can do all the above process directly by pressing the button "Execute program".

The main interface classes

When the VC++ project has been performed, the graphic interface is generated with 3 main classes. They are the minimum required classes (which are generated automatically by the Visual C++) to the correct functionality of the application.

They can be seen in the “Classes” label in the left window.



CAboutDlg: This class contains the "About message" dialog, as in most window-based applications.

CxxxxxxApp: This class is a standard class included in most projects to handle the application start-up, since there is no main function as in a typical C console application. It represents the application, and it is necessary for the compilation of the program.
Inside this one has to be declared the use of the OLE objects for the data exchange between the interface app and the DSP app. Last one has to be performed by executing the function: AfxOleInit();

CxxxxxxDlg: This class is the main class of the program. It represents the dialog window where appears the controls, buttons, lists and other app functionalities.
This class contains all the weight of the application, since this one is the main class. If it exists other windows, the main class allows the data exchange between all of them.

It also contains all the specific methods which are created to this project.

In addition to these classes, it is necessary to define one more class to enable the RTDX communication between the Visual C++ app and the ANSI C app, which it is explained later.

The graphical user interface

Here it exposes the different parts from the user interface, as well as their classes and methods.

Only when the user clicks in any check box or slider, the interface sends a data frame (using the RTDX communication) with the selected options to enable or disable and configure the audio effects previously shown.

Parallely when the tuner option is enabled, the interface receives the frequency value of the input signal and it shows on the screen.

The IrtdxExp class

As previously it has told, it must add manually an additional class (IrtdxExp) to manage the communication between the user interface and the DSP application. The communication is performed through the RTDX libraries.

This class is created from importing the Dll "Rtdxint.dll" supplied by Texas Instruments in the CCStudio folders.

Thus, it is possible from the interface to enable or disable the effects which the DSP processes. The DSK works like an external card.

The complexity is encapsulated in the class, from the dynamic library (dll).
Therefore, the RTDX has two interfaces: the first part corresponds to the Host (the user interface), and the second part to the Target (DSP).

The export RTDX interface allows the access to the functionalities from the Client to the Host. Using these functions, the Client application can get data from the RTDX libraries of the Host or it can send data to the RTDX libraries of the Host.

The functions in this interface can be used with Host clients written in Visual Basic, C++ or Labview. Below they are shown some of them. In the bibliography it can see many documents which explain these functions and its possible values.

Processor Activation:
SetProcessor

Configuration functions:
ConfigureRTDX
ConfigureLogFile
EnableRtdx, DisableRtdx
EnableChannel, DisableChannel
(The RTDX configuration can be done either from the user interface or from the CCStudio).

Functions to open/close channels:
Open
Close

Functions to read channels:
ReadSAI1, ReadSAI2, ReadSAI4, ReadSAF4, ReadSAF8
ReadSAI2V, ReadSAI4V
Read
ReadI1, ReadI2, ReadI4, ReadF4, ReadF8

Functions to write channels:
Write
WriteI1, WriteI2, WriteI4, WriteF4, WriteF8
StatusOfWrite

Functions to search channels:
Seek
SeekData

Flush functions of channels:
Flush

To use the IrtdxExp class in the interface, a couple of pointers are declared. With these pointers, it is possible to use all the previous methods.

To perform the communication, it must distinguish between the two types of channels:

- Input channels: They allow the communication from the Target application to the Host application. These channels are only for input data. Therefore, to access them, it must use the Read() function. But before, it must has been defined the previous channel as a global variable of the class.

- Output channels: They allow the communication from the Host application to the Target application. These channels provide the ability to send the configuration data to the code which is running in the Target application (DSP).
To use these output channels, first they must be declared. Then it accessed to the sending information using the Write() function or its derivatives.

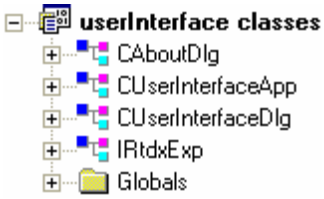
With the previous information, the implementation in the interface to use the RTDX communication is:

```
//The writing channel for the RTDX comm. is defined and set
w_RTDX = new IRtdxExp;
w_RTDX->CreateDispatch(_T("RTDX"));
if(!w_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
     MessageBox("It is impossible initialize the processor","Error");

//The reading channel for the RTDX comm. is defined and set
r_RTDX = new IRtdxExp;
r_RTDX->CreateDispatch(_T("RTDX"));
if(!r_RTDX->SetProcessor(_T("C6713_DSK"),_T("CPU_1")))
     MessageBox("It is impossible initialize the processor","Error");

//The RTDX communication is configured
w_RTDX->DisableRtdx();
w_RTDX->ConfigureRtdx(1,1024,4);  //Continuous mode, buffer of 1024
w_RTDX->EnableRtdx();             //and 4 buffers

//It shows error messages if these channels cannot be opened
if(w_RTDX->Open("RTDXinput","W"))
    MessageBox("Unable to open the writing channel","Error");
if(r_RTDX->Open("RTDXoutput","R"))
    MessageBox("Unable to open the reading channel","Error");
```

All the code can be seen in the annexes.

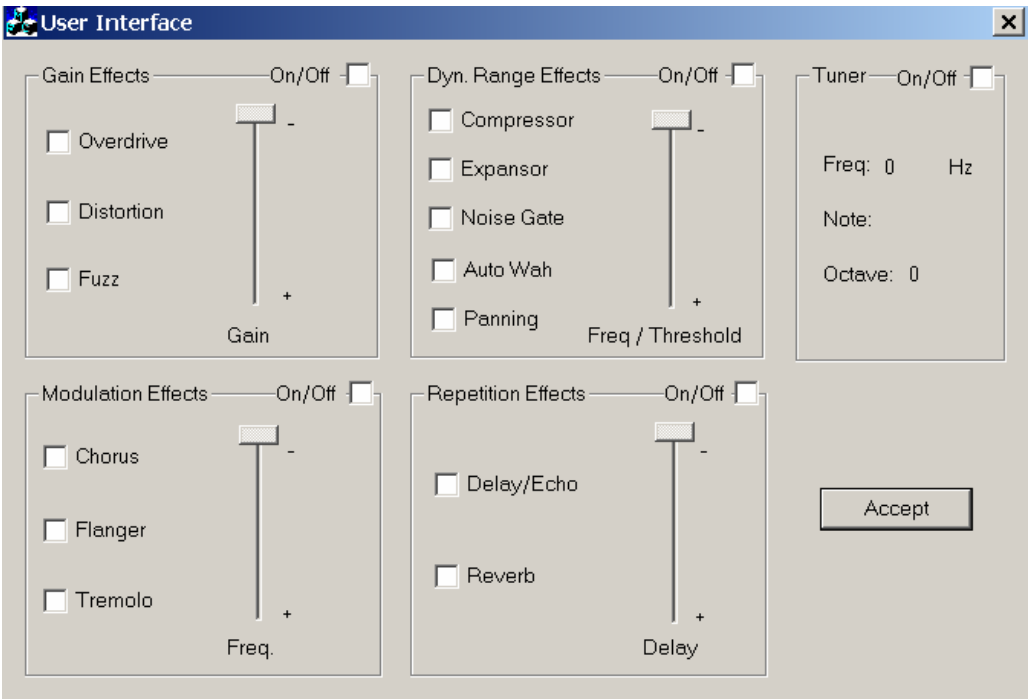
It has to remember that these channels are not bidirectional. Hence, each channel can be an input or output channel exclusively. For this reason, two channels are created.

When the class is added and the pointers are defined to use the RTDX communication, it can see the new class in the class hierarchy, with all its methods to be used inside.



The application parts

The present application is a very easy application which has the following parts:



- Many check boxes to activate or deactivate the flags which enable or disable the different effects. Only the effects inside a group are mutually exclusive. It adds text labels to clarify the function of every check box.

- 4 Check boxes to enable or disable each effect group (gain, dynamic range, modulation and repetition effects). It adds text labels to clarify the function of each check box.

- A slider for each effect group changes a specific parameter to vary the selected effect (quantity of gain, frequency or threshold value, frequency value and quantity of delay respectively) in real time. It adds text labels to clarify the variable paramenter in each slider.

- A check box and 3 text labels to interact with a digital tuner in real time. The tuner gives the name, the frequency and the octave of the received note from the DSP app.

Performed methods

The designed methods of the user interface are explained here. The rest of methods are omitted because they are created automatically by the Visual C++.
Anyway, all the effects can be seen in the annexes.

Gain effects

void CUserInterfaceDlg::OnOverdrive()

It activates or deactivates the corresponding flag for that the DSP app can perform the Overdrive effect when the specific check box is clicked.

Enabling this flag, it disables the Distortion and Fuzz effects if they are selected. The slider corresponding to the gain effects is rebooted.

```
UpdateData(TRUE);
if (m_overdrive == TRUE) {
    m_distortion = FALSE;
    m_fuzz = FALSE;
    GetDlgItem(IDC_SLIDER_GAIN)->EnableWindow(TRUE);
} else {
    GetDlgItem(IDC_SLIDER_GAIN)->EnableWindow(FALSE);
}
m_slider_gain = 0;
UpdateData(FALSE);

sndRTDX[7] = m_slider_gain;
sndRTDX[9] = m_overdrive;
SendArray();
```

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnDistortion()

It activates or deactivates the corresponding flag for that the DSP app can perform the Distortion effect when the specific check box is clicked.

Enabling this flag, it disables the Overdrive and Fuzz effects if they are selected. The slider corresponding to the gain effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnFuzz()

It activates or deactivates the corresponding flag for that the DSP app can perform the Fuzz effect when the specific check box is clicked.

Enabling this flag, it disables the Overdrive and Distortion effects if they are selected. The slider corresponding to the gain effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnSliderGain(NMHDR* pNMHDR, LRESULT* pResult)

When the corresponding slider is moved, it changes the quantity of gain of the selected gain effect under the bounds previously defined. If there is not any effect selected, or the check box for disable the gain effects is activated, the slider remains disabled.

```
UpdateData(TRUE);
sndRTDX[7] = m_slider_gain;
SendArray();

*pResult = 0;
```

The variable parameters are updated and this information is also sent to the DSP app.

void CUserInterfaceDlg::OnGainOnOff()

It deactivates and disables all the check boxes and the slider inside the gain effects group box, when the corresponding check box is clicked.

All the gain effects are not enabled until this flag is deactivated again, leaving those effects in their initial configuration.

```
UpdateData(TRUE);
if (m_gain_on_off == TRUE) {
    GetDlgItem(IDC_OVERDRIVE)->EnableWindow(FALSE);
    GetDlgItem(IDC_DISTORTION)->EnableWindow(FALSE);
    GetDlgItem(IDC_FUZZ)->EnableWindow(FALSE);
    GetDlgItem(IDC_SLIDER_GAIN)->EnableWindow(FALSE);
    m_overdrive = 0;
    m_distortion = 0;
    m_fuzz = 0;
    m_slider_gain = 0;
}
else {
    GetDlgItem(IDC_OVERDRIVE)->EnableWindow(TRUE);
    GetDlgItem(IDC_DISTORTION)->EnableWindow(TRUE);
    GetDlgItem(IDC_FUZZ)->EnableWindow(TRUE);
}
UpdateData(FALSE);

sndRTDX[7] = m_slider_gain;
sndRTDX[9] = 0;
SendArray();
```

The variable parameters are updated and this information is also sent to the DSP app.

Dynamic range effects

void CUserInterfaceDlg::OnCompressor()

It activates or deactivates the corresponding flag for that the DSP app can perform the Compressor effect when the specific check box is clicked.

Enabling this flag, it disables the Expander, Noise Gate, Autowah and Panning effects if they are selected. The slider corresponding to the dynamic range effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnExpansor()

It activates or deactivates the corresponding flag for that the DSP app can perform the Expander effect when the specific check box is clicked.

Enabling this flag, it disables the Compressor, Noise Gate, Autowah and Panning effects if they are selected. The slider corresponding to the dynamic range effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnNoiseGate()

It activates or deactivates the corresponding flag for that the DSP app can perform the Noise Gate effect when the specific check box is clicked.

Enabling this flag, it disables the Compressor, Expander, Autowah and Panning effects if they are selected. The slider corresponding to the dynamic range effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnAutoWah()

It activates or deactivates the corresponding flag for that the DSP app can perform the Autowah effect when the specific check box is clicked.

Enabling this flag, it disables the Compressor, Expander, Noise Gate and Panning effects if they are selected. The slider corresponding to the dynamic range effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnPanning()

It activates or deactivates the corresponding flag for that the DSP app can perform the Panning effect when the specific check box is clicked.

Enabling this flag, it disables the Compressor, Expander, Noise Gate and Autowah effects if they are selected. The slider corresponding to the dynamic range effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnSliderDynRan(NMHDR* pNMHDR, LRESULT* pResult)

When the corresponding slider is moved, it changes the frequency level or the threshold value of the selected dynamic range effect under the bounds previously defined. If there is not any effect selected, or the check box for disable the dynamic range effects is activated, the slider remains disabled.

The variable parameters are updated and this information is also sent to the DSP app.

void CUserInterfaceDlg::OnDynRanOnOff()

It deactivates and disables all the check boxes and the slider inside the dynamic range effects group box, when the corresponding check box is clicked.

All the dynamic range effects are not enabled until this flag is deactivated again, leaving those effects in their initial configuration.

The variable parameters are updated and this information is also sent to the DSP app.

Modulation effects

void CUserInterfaceDlg::OnChorus()

It activates or deactivates the corresponding flag for that the DSP app can perform the Chorus effect when the specific check box is clicked.

Enabling this flag, it disables the Flanger and Tremolo effects if they are selected. The slider corresponding to the modulation effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnFlanger()

It activates or deactivates the corresponding flag for that the DSP app can perform the Flanger effect when the specific check box is clicked.

Enabling this flag, it disables the Chorus and Tremolo effects if they are selected. The slider corresponding to the modulation effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnTremolo()

It activates or deactivates the corresponding flag for that the DSP app can perform the Tremolo effect when the specific check box is clicked.

Enabling this flag, it disables the Chorus and Flanger effects if they are selected. The slider corresponding to the modulation effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnSliderModul(NMHDR* pNMHDR, LRESULT* pResult)

When the corresponding slider is moved, it changes the frequency level of the selected modulation effect under the bounds previously defined. If there is not any effect selected, or the check box for disable the modulation effects is activated, the slider remains disabled.

The variable parameters are updated and this information is also sent to the DSP app.

void CUserInterfaceDlg::OnModuOnOff()

It deactivates and disables all the check boxes and the slider inside the modulation effects group box, when the corresponding check box is clicked.

All the modulation effects are not enabled until this flag is deactivated again, leaving those effects in their initial configuration.

The variable parameters are updated and this information is also sent to the DSP app.

Repetition effects

void CUserInterfaceDlg::OnDelayEcho()

It activates or deactivates the corresponding flag for that the DSP app can perform the Delay/Echo effect when the specific check box is clicked.

Enabling this flag, it disables the Reverb effect if it is selected. The slider corresponding to the repetition effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnReverb()

It activates or deactivates the corresponding flag for that the DSP app can perform the Reverb effect when the specific check box is clicked.

Enabling this flag, it disables the Delay/Echo effect if it is selected. The slider corresponding to the repetition effects is rebooted.

The variable parameters are updated and this data is also sent to the DSP app.

void CUserInterfaceDlg::OnSliderRepet(NMHDR* pNMHDR, LRESULT* pResult)

When the corresponding slider is moved, it changes the quantity of delay of the selected repetition effect under the bounds previously defined. If there is not any effect selected, or the check box for disable the repetition effects is activated, the slider remains disabled.

The variable parameters are updated and this information is also sent to the DSP app.

void CUserInterfaceDlg::OnRepeOnOff()

It deactivates and disables all the check boxes and the slider inside the repetition effects group box, when the corresponding check box is clicked.

All the repetition effects are not enabled until this flag is deactivated again, leaving those effects in their initial configuration.

The variable parameters are updated and this information is also sent to the DSP app.

Tuner implementation

void CUserInterfaceDlg::OnTunerOnOff()

This method allows that the user interface can receive information from the DSP application and it enables the tuner to use it.

A timer is set to execute periodically its method according the defined time.
It also deactivates the rest of the interface, because when the tuner is working, it does not have sense to use the audio effects. Here it shows part of this method:

```
UpdateData(TRUE);
if (m_tuner_on_off==TRUE) {
    sndRTDX[10] = 1;
    sndRTDX[13] = 1;
    SetTimer(1, 50, NULL);
    GetDlgItem(IDC_OVERDRIVE)->EnableWindow(FALSE);
    GetDlgItem(IDC_DISTORTION)->EnableWindow(FALSE);
    GetDlgItem(IDC_FUZZ)->EnableWindow(FALSE);
    GetDlgItem(IDC_SLIDER_GAIN)->EnableWindow(FALSE);
}
if (m_tuner_on_off==FALSE) {
    sndRTDX[10] = 0;
    sndRTDX[13] = 0;
    KillTimer(1);
    GetDlgItem(IDC_OVERDRIVE)->EnableWindow(TRUE);
    GetDlgItem(IDC_DISTORTION)->EnableWindow(TRUE);
    GetDlgItem(IDC_FUZZ)->EnableWindow(TRUE);
    GetDlgItem(IDC_GAIN_ON_OFF)->EnableWindow(TRUE);
}
m_freq = 0;
UpdateData(FALSE);
SendArray();
```

void CUserInterfaceDlg::OnTimer(UINT nIDEvent)

This method is executed periodically according the defined time by a timer.
It is only executed when the tuner flag is enabled.

It reads in real time the data which the DSP sends to the user interface (the frequency of the input signal) to calculate the name of the note and its octave.
Then it shows all the information on the screen.

```
if (m_tuner_on_off==TRUE) {
    float frequency = 0;
    r_RTDX->ReadF4(&frequency);
    r_RTDX->Rewind();
    if (frequency>=frec_min) {m_freq = frequency;}
    else {
        m_freq = 0;
        m_note = " ";
        m_octave = 0;}
    short octave = 0;
    while (frequency>frec_min)  {
        frequency = frequency / 2;
        octave++;}
    frequency = frequency * 2;
    for (short i=0; i<notesNumber; i++) {
        if (frequency<=(notesFrequency[i]*semitone)&&
            frequency>(notesFrequency[i]/semitone)){
                m_note = notesName[i];
                m_octave = octave;
                break;}}
    UpdateData(FALSE);
    sndRTDX[13]=2;
    sndRTDX[14]=1;
    SendArray();}
CDialog::OnTimer(nIDEvent);
```

Common methods

BOOL CUserInterfaceDlg::OnInitDialog()

This method is only executed once when the interface is initialized. It is created automatically by the program.

For this project, it adds the inicialization of the sliders and the variables to use.
Here it shows part of the code:

```
//At the beginning, all the sliders are diseabled
GetDlgItem(IDC_SLIDER_GAIN)->EnableWindow(FALSE);
GetDlgItem(IDC_SLIDER_DYN_RAN)->EnableWindow(FALSE);
GetDlgItem(IDC_SLIDER_MODUL)->EnableWindow(FALSE);

//All the variables/array are initialized
for (int i=0; i<longSND; i++) {sndRTDX[i] = 0;}
m_freq = 0;
m_octave = 0;

//The lowest frequency values are defined
notesFrequency[0] = 32,71;
notesFrequency[1] = 34,65;
notesFrequency[2] = 36,71;

//The note names are defined
notesName[0] = "Do / C";
notesName[1] = "Do# / C#";
notesName[2] = "Re / D";
```

It also includes the definition of the RTDX communication, which is shown above.

void CUserInterfaceDlg::SendArray()

This method is called whenever the array with the audio effects variables is updated. The data is sent from the user interfacing to the DSP using the RTDX communication.

```
VARIANT sa;
SAFEARRAYBOUND rgsabound[1];

::VariantInit(&sa);
sa.vt = VT_ARRAY | VT_I4;
rgsabound[0].lLbound = 0;
rgsabound[0].cElements = longSND;
sa.parray = SafeArrayCreate(VT_I4, 1, rgsabound);

HRESULT hr;
for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++) {
    hr = ::SafeArrayPutElement(sa.parray, &i, (long*)&sndRTDX[i]);}
long bufferstate;
w_RTDX->Write(sa, &bufferstate);
::VariantClear(&sa);
w_RTDX->Flush();
```

void CUserInterfaceDlg::ReceiveArray()

This method is designed to receive a data array from the DSP to the user interface using RTDX communication.

The received information would be used for the tuner implementation and to show the output signal level in real time. However, when the RTDX receives and sends information altogether, it has problems to work correctly. Therefore this method is only defined, but it is not used.

```
VARIANT sa;
HRESULT hr;
SAFEARRAYBOUND rgsabound[1];

::VariantInit(&sa);
sa.vt = VT_ARRAY | VT_R4;
rgsabound[0].lLbound = 0;
rgsabound[0].cElements = longRCV;
sa.parray = SafeArrayCreate(VT_R4, 1, rgsabound);

r_RTDX->ReadSAI2(&sa);
r_RTDX->Rewind();
for (long i=0; i<(signed)sa.parray->rgsabound[0].cElements; i++){
    hr = ::SafeArrayGetElement(sa.parray,&i,(long*)&rcvRTDX[i]);}
::VariantClear(&sa);
```

void CUserInterfaceDlg::OnOK()

This method closes the user interface when the "Accept" button is pressed. It is also created automatically by the program.

In this application, it also closes the timer defined before and it resets all the variables to their initial state to be sent to the DSP via RTDX.

```
KillTimer(1);
for (int i=0; i<longSND; i++) {sndRTDX[i] = 0;}
SendArray();
CDialog::OnOK();
```

sndRTDX

sndRTDX (ArrayRTDX in the DSP app) is the data array sent to the DSP app by the user interface whenever it interact with it (to move any slider, to press any check box ...).

Each position in this array informs to the DSP app about which effects have to be used and the variable value of each group of effects.

The information in each position is:

sndRTDX [14] = It solicits another frequency value from the DSP app to the user interface.

sndRTDX [13] = It changes the sampling frequency I/O between 48Hz and 8Hz. This flag is only used for the tuner algorithm.

sndRTDX [12] = For future uses.

sndRTDX [11] = For future uses.

sndRTDX [10] = It enables the tuner. It also disables the rest of audio effects.

sndRTDX [9] = It indicates which gain effect (Overdrive, Distortion or Fuzz) has to be used.

sndRTDX [8] = For future uses.

sndRTDX [7] = It indicates the quantity of gain for the gain effects.

sndRTDX [6] = For future uses.

sndRTDX [5] = It indicates which repetition effect (Delay/Echo or Reverb) has to be used.

sndRTDX [4] = It indicates the quantity of delay for the repetition effects.

sndRTDX [3] = It indicates which dynamic range effect (Compressor, Expansor, Noise Gate, Auto-Wah or Panning) has to be used.

sndRTDX [2] = It indicates the the frequency level or the threshold value for the dynamic range effects.

sndRTDX [1] = It indicates which modulation effect (Chorus, Flanger or Tremolo) has to be used.

sndRTDX [0] = It indicates the frequency level for the modulation effects.

CONCLUSIONS

Cost analysis

This project was carried out in a total of 8 months (2 semesters), in which it has been invested all the possible time in the documentation, studying, designing and programming of it.

First of all, the time spent to simulate the audio treatment in real time with MATLAB has been:

SIMULATING WITH MATLAB	
Task	**Time**
Studying of audio treatment	20 hours
Studying the acquisition in real time	20 hours
Fast Fourier Transform	20 hours
Programming in MATLAB	80 hours
Others	10 hours
TOTAL	**150 hours**

150 hours of studying were necessary to simulate the function of the project.

The time spent to study each part of the project has been:

STUDYING OF TMS320C6713	
Task	**Time**
Processor DSP	30 hours
Programming environment	30 hours
Peripheral audio codec	10 hours
Peripheral McBSP (comm. serie)	15 hours
Interruptions	15 hours
Peripheral EDMA	40 hours
Peripheral RTDX	30 hours
Rest of peripherals	15 hours
Programming language ANSI C	50 hours
Others	25 hours
TOTAL	**260 hours**

STUDYING OF THE INTERFACE	
Task	**Time**
Searching of a software	10 hours
Using Microsoft Visual	20 hours
Programming language Visual C++	50 hours
Communication RTDX	30 hours
Others	30 hours
TOTAL	**140 hours**

400 hours of studying were necessary to prepare the project.

When all the knowledge has been learnt, the invested time to design and programming this project has been:

PROGRAMMING THE DSP APP	
Task	**Time**
Configuration of the project	10 hours
Initialization of the DSP	20 hours
Configuration of audio codec	15 hours
Configuration of McBSP	25 hours
Configuration of EDMA	40 hours
Managing the interruptions	20 hours
Effects programming	400 hours
Implementation of the FFT	160 hours
Communication RTDX	30 hours
Others	40 hours
TOTAL	**760 hours**

PROGRAMMING THE INTERFACE APP	
Task	**Time**
Configuration of the project	10 hours
Designing the interface	25 hours
Programming the interface	55 hours
Communication RTDX	30 hours
Others	20 hours
TOTAL	**140 hours**

900 hours of programming were necessary to perform the project.

Once the programming has been finished, the time spent writing the memory has been:

WRITING THE MEMORY	
Task	**Time**
Introduction (1st part)	30 hours
Programming (2nd part)	70 hours
Conclusions (3rd part)	30 hours
Others	15 hours
TOTAL	**145 hours**

145 hours of writing were necessary to document the project.

Budget

The sum of all these hours makes a total of 1595 hours, or 200 working days if they are distributed in 8 hours/day (like the working time of a worker).

To do the budget of this project, it can only count the programming hours. The hours spent in the studying or simulating (and writing the memory) cannot count because these ones are not part of the project development.

This results in 900 hours spent in the development (programming) of the project, or 113 working days if they are distributed in 8 hours/day (like the working time of a worker).

The price charged to a customer is **25 €/hour**. This price includes the labor of junior engineer, the used material, internal expenses, etc.

It also has to count the price of the hardware TMS320C6713, which is **304 €** aprox.

Therefore, it multiplies the hours spent in scheduling by the upper price per hour, and it adds the price of the used hardware.

The project has a budget of:

Twenty two thousand eight hundred and four euros (22804 €).

Conclusions

Final results

All the performed tests have resulted correct:

Most of the effects are generated correctly with the designed algorithms, but some of them are a little bit different in respect of the original idea.

Following there is a brief description about these results, effect by effect:

- **ByPass:** This effect (although it is not really a real effect) works perfectly. It is verified that the codec takes out the same information which receives.
- **Overdrive:** This effect works fine. This effect applies a few distortions in the output signal.
- **Distortion:** This effect works very well. The signal is distorted and it has a lot of gain in the output.
- **Fuzz:** This effect works fine. The result can annoy in high levels.
- **Compressor:** This effect works very well. It offers compression, but the result is not very impressive.
- **Expansor:** This effect works fine. The result can annoy in high levels.
- **Noise Gate:** The algorithm is technically good, but when the input signal is around the threshold, the result is strange. In this specific case, it produces some noise due to the speakers, which are not of good quality.
- **Auto Wah:** The algorithm and the result are technically good, but the result is a bit different in respect of the expected one.
- **Panning:** This effect works very well. The sound rotates around the speakers perfectly.
- **Chorus:** This effect works very well. It simulates the two inputs with a little delay between them.
- **Flanger:** This effect works fine, but the result is a bit different in respect of the expected one.
- **Tremolo:** This effect works very well. The result oscillates according to the output gain.
- **Delay/Echo:** This effect works very well. If the delay time is small, the Echo effect (it cannot distinguish the two used inputs) can be heard. And if the delay time is big, the Delay effect (it can distinguish the two used inputs) can be heard.
- **Reverb:** This effect works very well. If the delay time is small, it looks like to the delay effect. But when the delay time is big, the acoustics of the room can be perfectly heard.
- **Tuner:** The tuner only works fine when the input frequency is constant (using a function generator, for example).
  In addition, it can only detect frequencies between 32Hz to 4000Hz approx.

The minimum frequency (32Hz) is due to the configuration of the window length and the window step, which determine the resolution according the sampling frequency. To improve this implementation, the FS has to be 8000Hz.
32 Hz is the Do/C note in the first octave.
The maximum frequency (4000Hz) is due to the Nyquist filter which the codec incorporates to ensure a good quality of the input samples. This means that, in fact, the FS is 8000 Hz.

The effects used together work fine. Until 4 effects simultaneously can be put (one from each group), getting curious results.
The effects of gain have presented problems to combine with others effects because they have big output gain. But those problems were fixed.

In respect of the user interface, the result has been perfect.
All the check boxes, sliders and labels work correctly. The effects can be activated and deactivated using the check boxes. The specific value of the sliders can be changed and the data of the tuner can be seen with its labels without problems.

Although the tuner implementation in the DSP app has limitations, when a frequency is detected and sent to the user interface, this one shows the frequency, the name of the note and its octave on the screen perfectly. Without any errors, as it was scheduled.

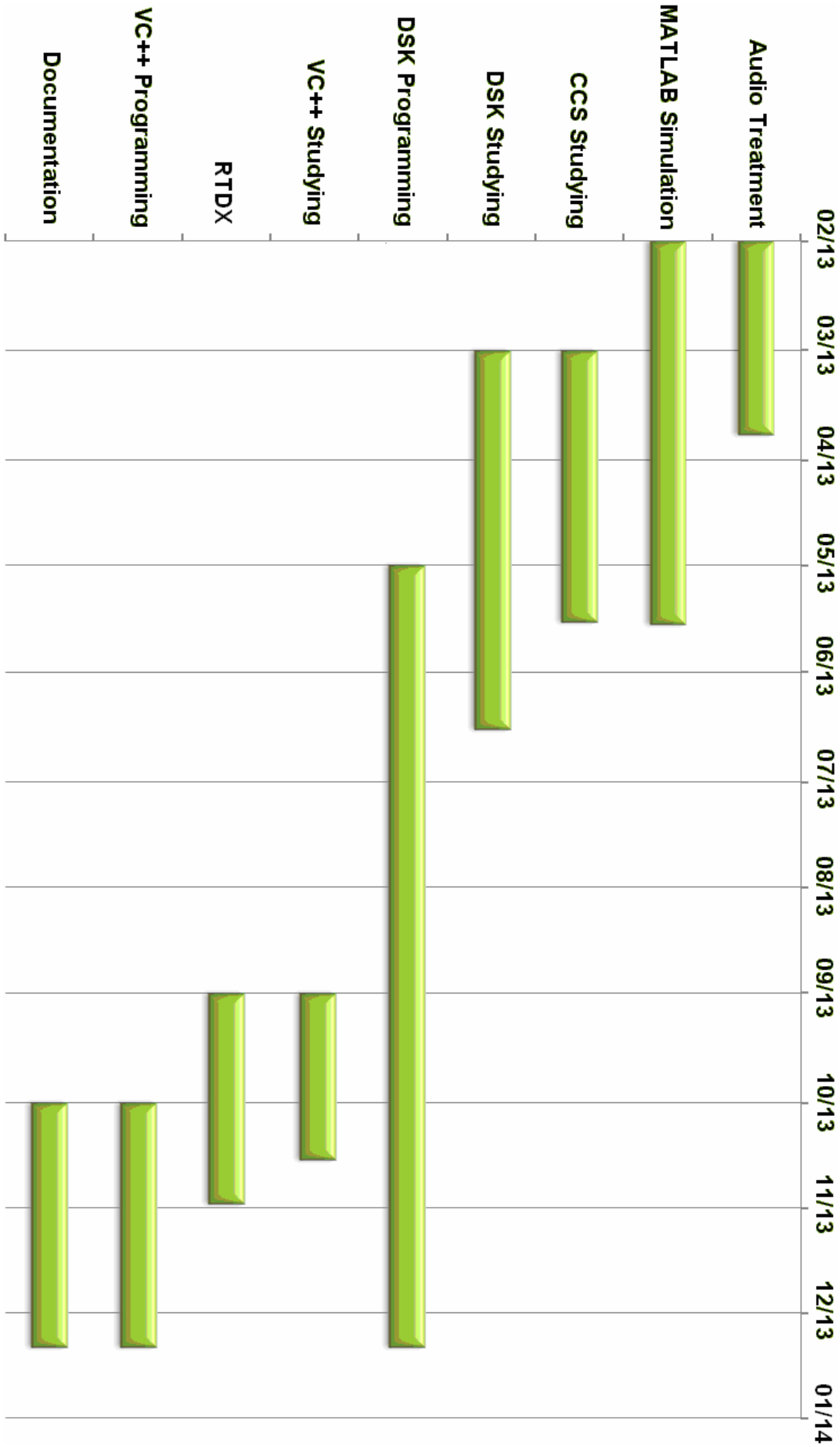
The RTDX communication works fine. But it has problems when it sends and receives data at a time. In this case, it is sometimes blocked.
Therefore, it is not a good idea to send and receive information at a time. For this reason, it was not possible to implement a volume screen.

All the other audio processing parts work fine too.
Therefore, the audio codec gets and samples the input data to use them later. When the data are treated, the codec also takes out the information correctly.
The EDMA moves the data to the different destinations without problems.

When a external device supplies the data (like an iPod or PC), their input samples have a correct amplitude value.
But when a musical instrument is connected, it sends the data with a very small amplitude.
Therefore, a preamplifier is necessary between the DSK and the musical instrument to amplify the input data.
With the preamplifier, the instrument can be used as the source perfectly.

Therefore it can be said that all the objectives defined at the beginning of the project are fulfilled.

The following picture shows the schedule with the different tasks of this project:


02/13
03/13
04/13
05/13
06/13
07/13
08/13
09/13
10/13
11/13
12/13
01/14
Audio Treatment
MATLAB Simulation
CCS Studying
DSK Studying
DSK Programming
VC++ Studying
RTDX
VC++ Programming
Documentation

Future developments

The main improvement that should be made to this project is to make a hardware interface to interact with it with the feet. Namely, to change or set effects while it is playing the musical instrument at a time.
The hardware interface requires more work than the software interface, hence the second one was the most suitable option due to lack of time.

Another extension is the developing of new effects, like creating the phaser or equalizer effects.
It can also modify some of the existing effects which can admit greater complexity in order to achieve more realistic sounds.
For example, the improving of the chorus effect with more voices simultaneously. It can also improve the reverb effect, setting the particular acoustic conditions to reproduce a better background sound, including special characteristics such as attenuation or amplification of certain materials, dimensions and specific geometries, etc.

Moreover, it could use a communication system more robust in respect of the one implemented in this project, since most of transmissions have not verification of receipt or delivery. Therefore it can be interesting to investigate more about this field.
By the way, it is not appreciated any failure in the sending or receiving separately.

Finally, in the VC++ interface part, it would be interesting to implement a system to save the configuration of the effects in a simply file for load and use it again in another time with the same parameters.

It is also interesting to improve the interface with those end small details in the communication interface as to use a Smitter to know the amplitude values of the input sound, to create a menu with support options, etc.

Personal conclusions

I am very satisfied with the realization of this project and its final result.

I could see that the field of the sound effects is a difficult and extensive field. Even if I have only worked the basic concepts, they have helped me to understand many things of this exciting world. They also made me want to continue investigating on my own.

It is necessary to say that the implementation of all the system has been hard, since adequate information was not available. Therefore, many hours have been devoted of consultation and tests with the compiler in order to implement all the features.

Personally, the development of this project has helped me to understand better the functioning of my personal equipment (amplifier, effect pedals and the electronics of the guitar) and learn to use it more efficiently.

Another big plus for me was the need to learn two programming languages very important in the professional world: the ANSI C language (to schedule the DSP microprocessor) and the Visual C++ language (to schedule the graphical interface).

Even I had to attend a special class outside of my degree to learn how to use the hardware from Texas Instruments, giving me to understand the performance of the DSPs and the most important and commonly used peripherals.

With all these knowledge, I think that I will be a more efficient and versatile engineer.

I also have to say that it has only been possible to reach the end of this road with the help of my professors, who answered my questions and explained the steps to complete the project properly.

I recommend to other students for their future PFGs also investigate the sound and its effects, or directly continue this project to make it more complete, especially if they like playing a musical instrument like I do.

Bibliography

Audio treatment

[1] José Mª Grijota Delgado. **Implementación de un procesador digital de efecos mediante DSP e interfaz gráfica sobre plataforma Windows XP.**
Published in 2008.

[2] Cristian Quirante Catalán. **Implementación de algoritmos de efectos de audio en un procesador DSP de Texas Instruments.**
Published in 2008.

[3] Udo Zölzer. **DAFX: Digital Audio Effects, Second Edition**.
Published in 2011 by John Wiley & Sons, Ltd. ISBN: 978-0-470-66599-2

[4] D. Marshall. **MATLAB, DSP, Graphics. Module No: CM0268.**

TMS320C6713

[5] Rulph Chassaing. **Digital signal processing and applications with the C6713 and C6416 DSK**.
Published in 2005 by John Wiley & Sons, Inc., Hoboken, New Jersey.

[6] Texas Instruments. **TMS320C62x DSP Library – Programmer's Referente Guide**
Published in 2003. Literature Number: SPRU402B

[7] Texas Instruments. **TMS320C67x DSP Library – Programmer's Reference Guide**
Published in 2010. Literature Number: SPRU657C

[8] Texas Instruments. **TMS320C6000 – Chip Support Library – API Reference Guide**
Published in 2004. Literature Number SPRU401J

[9] Texas Instruments. **TMS320C6000 DSP – 32Bit Timer - Reference Guide.**
Published in 2003. Literature Number: SPRU582B

[10] Texas Instruments. **TLV320AIC23B – Stereo Audio CODEC, 8 to 96 kHz, With Integrated Headphone Amplifier.**
Published in 2004

[11] Texas Instruments. **TMS320C6000 DSP – Multichannel Buffered Serial Port (McBSP) - Reference Guide.**
Published in 2006. Literature Number: SPRU580G

[12] Texas Instruments. **TMS320C6000 DSP – Enhanced Direct Memory Access (EDMA) Controller - Reference Guide.**
Published in 2006. Literature Number: SPRU234C

[13] Sophocles J. Orfanidis. **DSP Lab Manual**
Published in 2012.

RTDX communication

[14] Horst Rogalla. **RTDX Tutorial Version 1.0**
http://www.tsseshop.com/Developer/Tutorials/RTDX/TutorialRTDX.html

[15] Deborah Keil. **Real-Time Data Exchange. Digital Signal Processing Solutions**
http://www.ti.com/lit/wp/spry012/spry012.pdf

[16] **APPLICATION NOTE – RTDX feature**
http://www.sundancedsp.com/docs/RTDXbyJTAG.pdf

Graphical user interface

[17] **Application Programming Interface (API) of C++**
http://www.cplusplus.com/reference/

[18] Demian C. Panello. **Tutorial 1 of Visual C++**
http://www.dcp.com.ar

[19] **Tutorial 2 of Visual C++**
http://www.programacionfacil.com/visual_cpp/start

[20] **Tutorial 3 of Visual C++**
http://www.tenouk.com/visualcplusmfc/visualcplusmfc25a.html