Università degli
Studi di Padova

Informatica Musicale

Hardware DSP design for audio effects

an approach to the "open source" concept for developing hardware-based digital audio effects

Prof. Giovanni de Poli

Student: Himar Alonso Díaz

Contents

List of Figures

List of Tables

Document status

Version	Changes
1.0	Initial release

1 Introduction

Nowadays the industry of audio processing for professional applications is getting more and more developed, especially in the field of digital effects. All related research is usually kept secret since every company sells the final product, without the –detailed– circuit schematics. The software industry, across the *open source movement* [1], maintains that *not sharing the source code means restricting the sharing of knowledge, and therefore limits intellectual development in society.* They have also strongly proven that sharing this information *freely* helps worldwide developers to create applications without the need to reinvent what has already been invented; there is also the possibility to create development communities; modular programming and open source produces fast and efficient software development; the debugging process is much easier when thousands of developers have open-access to the source code[1]...

The current project proposes an approach to the digital hardware design in line with the "open source" concept, particularly for developing *hardware-based digital audio effects.*

Since the digital audio processors became so widely used –and cheap– there has been a controversy between the quality between analog and digital audio effects. The analog effects are traditionally said to obtain much better results in most cases, however today's A/D and D/A converters and *Digital Signal Processors* (DSP) also give a very good performance.

At the present time the Electronic Industry also offers us new possibilities to develop digital hardware without spending such high quantities of money. For example, the use of *Field-Programmable Gate Arrays* (FPGA) permits the developer to make as many designs as they want just by using a single chip, which can be reprogrammed as if it were a software-based system. In fact, today, most of these designs start with a "computer language" description (these languages for digital hardware design are called *Hardware Description Language* - HDL), which is then synthesised and implemented by using a computer CAD tool, and finally downloaded onto the chip.

1.1 Project proposal

The present conditions set the ideal framework to make a further advancement, and *create* a new "system" where digital effects can be easily designed, tested, and used. The aim of this system is to make programming easy for the programmer. The implementation of FPGA devices will be useful for this purpose. In addition to this, programming audio effects by using HDL gives us all the advantages of software programming, i.e. low cost for production –just a common PC is needed–, quick testing and easy debugging, easy distribution and expansion,...

In order to realise these objectives, I propose the following project, which is initially divided in two stages:

Framework design: Consists of designing a physical support for handling and using the following digital designs.

[1]It would also be more accurate to include part of the hardware industry. Some hardware companies also release some of their product schematics (e.g. *OpenSPARC* processor, by *Sun Microsystems*).

- Before the digital audio becomes processed, analog-to-digital conversion is needed. Also the digital-to-analog module is required after the digital processing.
- We also need to specify the *entity*[2] of every effect block. The design of the entity should be *standard*: all the blocks must use the same entity, and it must be possible to connect n blocks in cascade configuration, according to the designer's choice.
- A physical user interface is needed for handling the parameters of every effect block. This interface should contain buttons, rotary switches, and some kind of display device, which must be the same for all effects (so the effects will need to be adapted to this generic interface). Then, settings should be saved into an EEPROM memory block.

Expansion Having a framework like the one described previously, would make the programming and testing of all kind of effects extremely easy (a simple *preamplifier*, *delay*, *chorus*, *distortion*,...) by simply:

- Using a template design of the *entity* (interface) to program the *architecture* (implementation) of every effect block.
- Connecting in cascade as many effect blocks as desired (with the only restriction of hardware size[3]).
- Synthesising, implementing and downloading the program onto the FPGA.

This proposal sets a *standard framework* for digital effects, making it public (across the Internet, in a dedicated hardware design website, or an existing one like *http://www.opencores.com/*) would permit the creation of an international community for HDL audio effects development, under the open source philosophy.

1.2 Applications

This proposal is initially intended for a *single-channel real-time signal processing instrument*. It is preferable to start from a simple, but reachable proposal.

Once developed, it will be easy to adapt the system for stereo (or generally *multichannel*) applications.

The instrument has a wide scope. Input *gain value* or *filters* should be slightly different, depending on the instrument (furthermore, not all the instruments use the same kind of effects, but this is a programmable parameter). Taking into account these subtle differences, the present proposal could be easily adapted for:

- Acoustic or electric guitar
- Bass guitar
- Keyboard

[2]In hardware design, an *entity* describes the interface, the input and output ports, but not the actual implementation, which is described instead in the *architecture* [1].

[3]In microelectronics, the hardware size measure is the "number of gates", where *gates* refers to the MOS transistor gates.

2 Framework design

A *personal computer* is the most typical physical support for software development, but it is also a physical support for the final user. In the same way, we need to design a physical support, which can act either as an effect developer or a final user. An initial block diagram is shown in Figure 1. The framework design consists on finding the way to "connect" all devices, and the optimal specifications for all these connections.

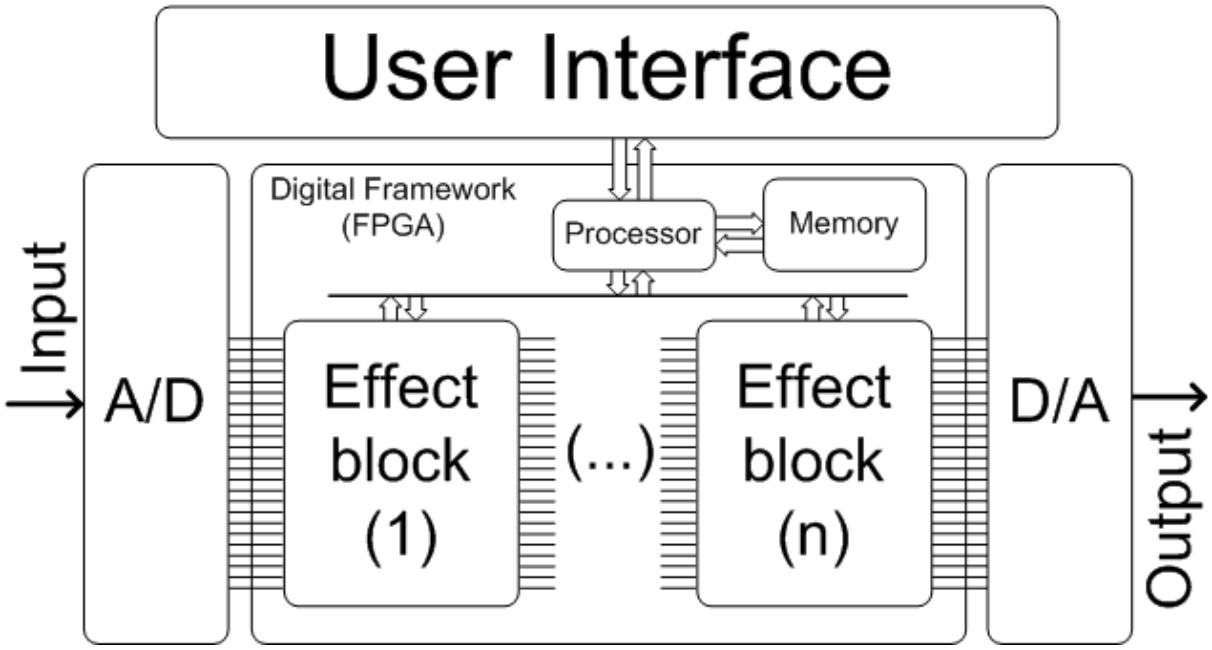

Figure 1: Framework initial block diagram

Some of these parts are analog, others digital, so also all voltage levels, number of bits, communication protocols,... need to be specified.

2.1 Input and output stages

Since all the music signals are initially analog (even the synthesisers, which produce digital sounds, have an analog output), either the input or output framework interfaces must be analog. In this first design release we will try to concentrate all efforts on the digital part. To begin, there must be an input and output circuitry in order to adapt the *impedances* and the *voltage levels*.

2.1.1 A/D and D/A conversion

The analog-to-digital and digital-to-analog converters are the most important –physical– parts, because their quality directly determines the final audio quality. The Burr-Brown division of Texas instruments is traditionally known to produce very high quality audio chips. The ADS-1271 analog-to-digital [5] and DSD-1794A digital-to-analog converters [4] would make an interesting reference point for this project.

When sound becomes digital, an important parameter appears, which is the *system clock frequency*, usually represented by the "`clk`" signal. In all digital systems it is very

important to have *just one* clock signal source. Otherwise any phase differences may cause synchronisation problems. For the A/D and D/A proposed devices, a 27MHz clock is enough. A work frequency of 100MHz will suffice all the digital processing needs, and the lower frequencies can be easily generated by using simple *counters*.

2.1.2 Quantisation

The *quantisation* is the process of *approximating* a continuous range of values by a relatively-small set of discrete symbols or integer values [1]. Depending on the number of discrete values, we will need more or less data bits, which is an important and delicate question. When we quantise the audio signal samples, a *quantisation noise* is added to the original audio signal [2]. The more bits we use for quantisation, the smaller the noise, and the higher remains the quality (see the graphical example in Figure 2). On the other hand, an excessive number of bits would make the system unaffordable (physically and economically).

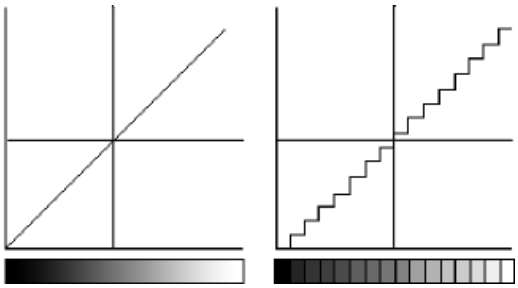

Figure 2: Example of quantisation process

A compact disc (CD) audio is sampled at $44,100$Hz and quantised with 16 bits [1], so for every sample, one of the $2^{16} = 65,536$ discrete values is taken. For studio recordings and other accurate applications, higher precision is needed (in some cases 32 or even 64 bits). For digital audio processing the 16 bits used for audio data storage would not suffice the development of all effects. An average choice of 24 bits ($2^{24} = 16,777,216$ possible discrete values) would give an optimal performance.

2.1.3 Codification

Every quantisation level must correspond to a 24-bit binary code, in order to *identify* the sample level of quantisation. This process is called *codification*, and the codifier is usually integrated into the A/D chip. For digital-to-analog conversion, a *decodifier* is used instead. Most A/D and D/A converters use a "two's complement" based codification [5]. The Table 1 shows the most significant (hexadecimal) code values, versus the input signal voltage.

2.2 DSP entity block

As mentioned previously, we call the *entity* to every effect block *interface*, which includes input and output ports, but not the internal behaviour or structure. The entity needs

Input signal	Output code (hex)
$\geq +V_{\mathrm{REF}}$	7FFFFF
$(\cdots)$	$(\cdots)$
$V_{\mathrm{REF}}\left(\frac{2}{2^{23}-1}\right)$	000002
$V_{\mathrm{REF}}\left(\frac{1}{2^{23}-1}\right)$	000001
0	000000
$-V_{\mathrm{REF}}\left(\frac{1}{2^{23}-1}\right)$	FFFFFF
$-V_{\mathrm{REF}}\left(\frac{2}{2^{23}-1}\right)$	FFFFFE
$(\cdots)$	$(\cdots)$
$\leq -V_{\mathrm{REF}}\left(\frac{2^{23}}{2^{23}-1}\right)$	800000

Table 1: Output code versus input signal

to be as *generic* as possible, in order not to limit the programmer's design possibilities. Every effect block will have the following ports:

Clock: Single-bit input port.

Reset: Single-bit input port. System reset for device initialisation.

Signal input: 24-bit input port for signal input.

Signal output: 24-bit input port for signal output.

Control input: 8-bit input port for the handling of effect parameters.

Control output: 8-bit output port for the controlling of effect parameters and sending of display information.

Aux input: Single-bit auxiliary input. Undetermined specific use; it may be useful if (for example) a *round-robin* architecture is needed to handle the effects one by one.

Aux output: Single-bit auxiliary output. If it was necessary to develop the previous example, we should implement a simple *shift register*: (`Aux_Output`←`Aux_Input`).

The VHDL implementation code of this entity is as follows:

```
entity effect is
    Port ( clk      :  in  std_logic;
           rst      :  in  std_logic;
           input    :  in  std_logic_vector (23 downto 0);
           output   :  out std_logic_vector (23 downto 0);
           ctrl_in  :  in  std_logic_vector (7 downto 0);
           ctrl_out :  out std_logic_vector (7 downto 0);
           aux_in   :  in  std_logic;
           aux_out  :  out std_logic );
end effect;
```

2.3 DSP architecture example

So let us take the DSP entity and give a practical and easy example using VHDL. In fact, in this first design we will use neither the *control* buses nor the auxiliary signals, so these ports will be short-circuited (as shown in the first lines of the architecture code).

One of the principal effects used by guitarists is the *overdrive* distortion. The analog way to reach this effect is rising the gain up to the non-linear work zone of the transistors. A mathematical model is proposed [3] on this function:

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

The input (x) and output ($y = f(x)$) signals are shown in Figure 3. We need to adapt this function to our 24-bit system. In two's complement the first bit stands for the sign, and the rest for the –absolute– value, so the maximum possible value is $2^{23} = 8388607$. In Table 2 there is the equivalence for the most significant points.

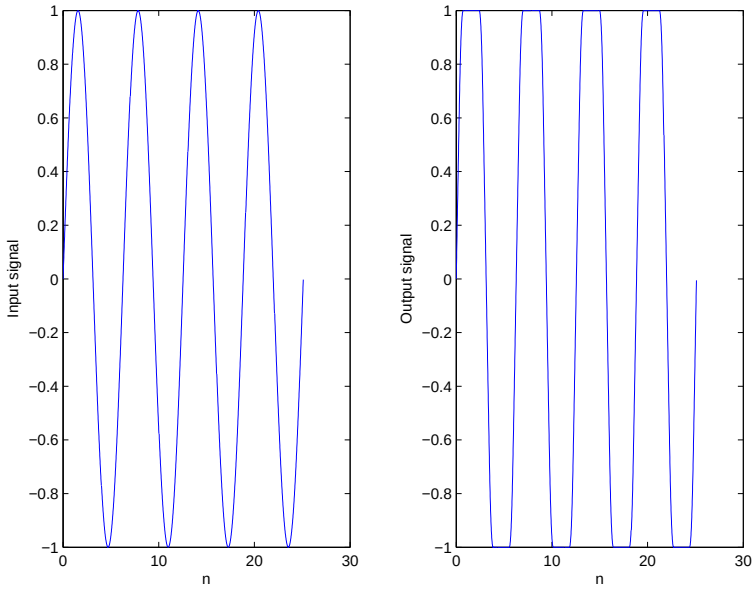

Figure 3: Effect of symmetrical clipping overdrive

Value (0 to 1)	24-bit equivalence	Hexadecimal value
0	0	000000
1/3	2796202	2AAAAA
2/3	5592404	555554
1	8388607	7FFFFF

Table 2: Output code versus input signal

Adapting the function, we have:

$$f(x) = \begin{cases} 2x & \text{for} \quad 0 \leq x \leq 2796202 \\ -(3,576 \times 10^{-7})x^2 + 4x - 2796202 & \text{for} \quad 2796202 \leq x \leq 5592404 \\ 1 & \text{for} \quad 5592404 \leq x \leq 8388607 \end{cases}$$

But it is not so practical to multiply by such small values $(3,576 \times 10^{-7})$, so we can make an approach to this parable function by a straight line (see characteristic curve(s) in Figure 4). In the worst case, just a $8,33\%$ error is made.

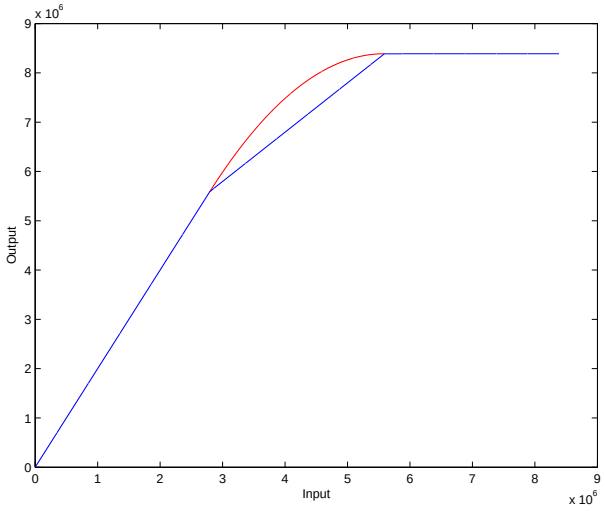

Figure 4: Static characteristic curve of symmetrical clipping overdrive (just positive values are represented). The theoretical curve is in red, the approach for VHDL implementation is the one in blue

So the approach function we will implement in VHDL is definitely:

$$f(x) = \begin{cases} 2x & \text{for} \quad 0 \leq x \leq 2796202 \\ x + 2796201 & \text{for} \quad 2796202 \leq x \leq 5592404 \\ 1 & \text{for} \quad 5592404 \leq x \leq 8388607 \end{cases}$$

The Figure 5 shows a FSM (*Finite State Machine*) diagram of the algorithm for implementing this function, dividing it into different computation *stages*. This division makes it possible to *pipeline* the process, which means that there is no need to wait for one sample to be processed before starting to process the following one [1]. This is because the processing in every stage is designed to be *independent* from the others. These kind of designs are highly recommended, since they are optimised for reducing unwanted signal *delays*.

The VHDL implementation code of this algorithm is as follows:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

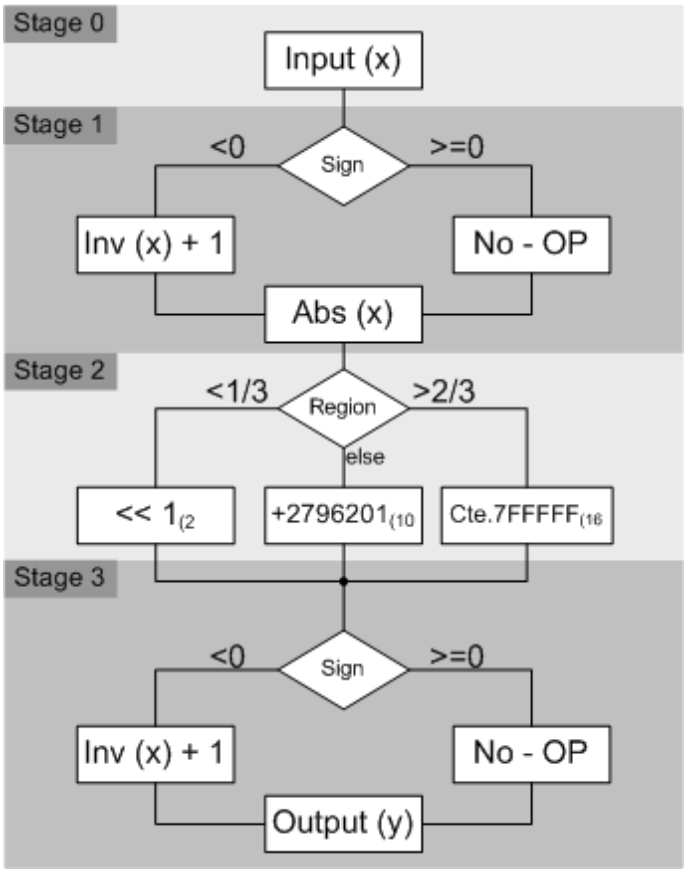

Figure 5: Overdrive FSM diagram

```
entity effect is
    Port ( clk      :  in  std_logic;
           rst      :  in  std_logic;
           input    :  in  std_logic_vector (23 downto 0);
           output   :  out std_logic_vector (23 downto 0);
           ctrl_in  :  in  std_logic_vector (7 downto 0);
           ctrl_out :  out std_logic_vector (7 downto 0);
           aux_in   :  in  std_logic;
           aux_out  :  out std_logic );
end effect;

architecture symclip of effect is

   signal clean :  integer range 0 to 3 := 3;
   signal sign :  std_logic_vector (2 downto 0);
   type t_reg is array (3 downto 0) of std_logic_vector (23 downto 0);
   signal reg :  t_reg;

begin

   aux_out <= aux_in;
```

```
ctrl_out <= ctrl_in;

process (clk,rst)
begin
   if rst = '1'then
      clean <= 3;
      reg <= (others => (others => '0'));
      sign <= (others => '0');
      output <= (others => '0');
   elsif clk'event and clk = '1'then
      --
      -- Zero output during the dirty cycles:
      -- (3 first cycles after system reset)
      --
      if clean > 0 then
         clean <= clean - 1;
      end if;
      --
      -- Sign bit shift register
      --
      sign(2 downto 1) <= sign(1 downto 0);
      --
      -- Stage 0: Input capture
      --
      sign(0) <= input(23);
      reg(0) <= input;
      --
      -- Stage 1: reg(1) <= abs(reg(0))
      --
      if sign(0) = '1'then
         reg(1) <= not(reg(0)) + 1;
      else
         reg(1) <= reg(0);
      end if;
      --
      -- Stage 2: Signal distortion
      --
      if reg(1) <= X"2AAAAA" then
         reg(2)(0) <= '0';
         reg(2)(22 downto 1) <= reg(1)(21 downto 0);
         reg(2)(23) <= '0';
      elsif reg(1) >= X"555554" then
         reg(2) <= X"7FFFFF";
      else
         reg(2) <= reg(1) + X"2AAAA9";
      end if;
      --
```

```
        -- Stage 3: Sign recover & output
        --
        if sign(2) = '1'then
            if clean = 0 then
                output <= not(reg(2)) + 1;
            else
                output <= (others => '0');
            end if;
        else
            if clean = 0 then
                output <= reg(2);
            else
                output <= (others => '0');
            end if;
        end if;
    end if;
  end process;

end symclip;
```

2.4 Digital framework

In the present design the *digital framework* refers to the part which is contained in the FPGA. This fact allows the designer to customise even further the behaviour of the effects processor. In any case, one of the most important purposes of this project is letting the designer concentrate on the effect blocks, so we should provide a *standard base design* (even if in this first release just some general ideas will be given).

The best way to manage all the effect blocks and the interaction with the user is by using a small *microprocessor*. There are many HDL processor designs like the PicoBlaze™ [6], by *Xilinx*, ready to be included into our design. The PicoBlaze™ has a simple structure –see Figure 6–, is very well-documented, and comes with assambler programming tools. It is a worthwhile choice.

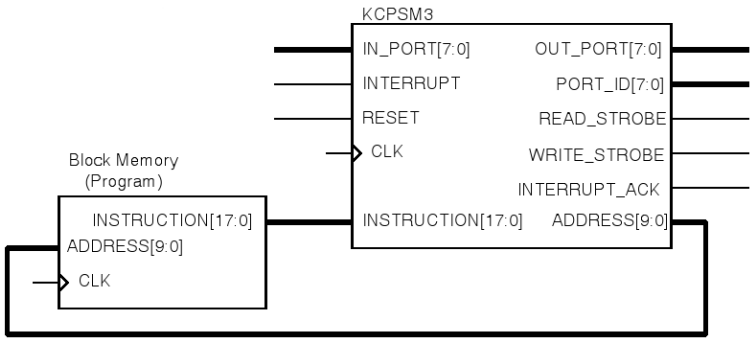

Figure 6: PicoBlaze™structure

A suggested block diagram for the digital framework is proposed in Figure 7. More important than embedding the processor in our framework is *designing the software routines* to control all of the other blocks. Notice that the processor we are refering to now has nothing to do with signal processing; this mission corresponds to the effect blocks, as said before.

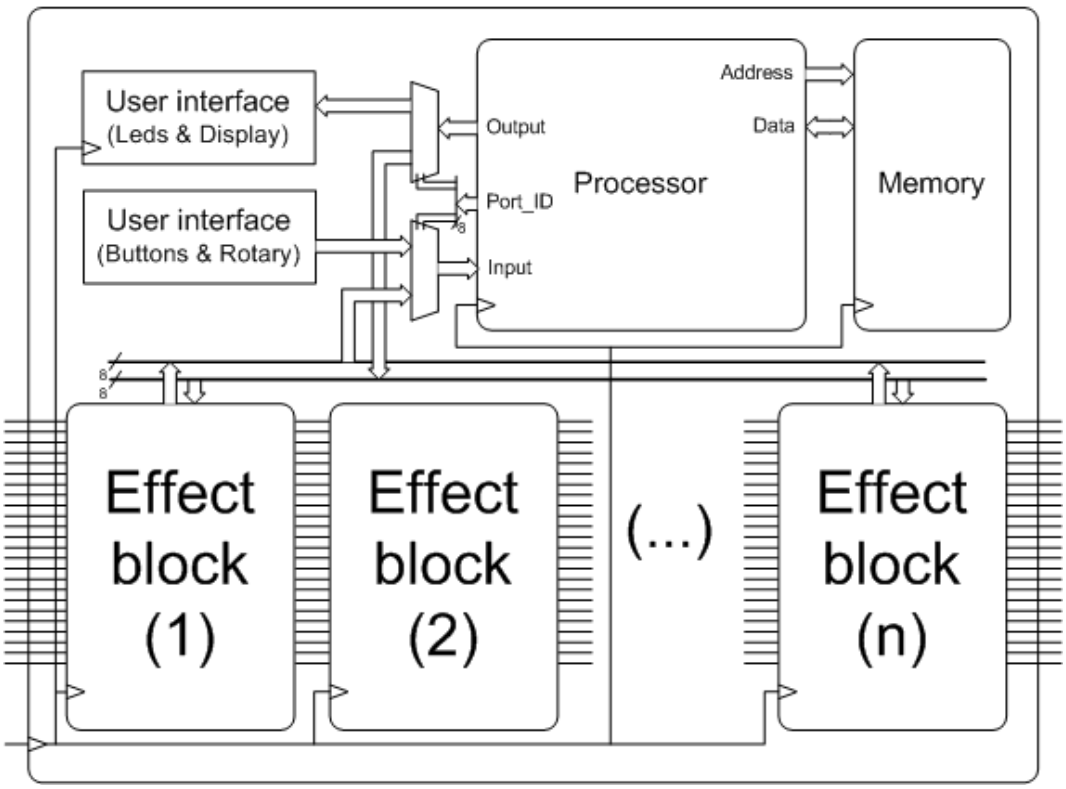

Figure 7: Digital framework block diagram

The processor routines should take into account the following main tasks:

- The capturing of user interface events.
- The management of effect parameters.
- Display control.
- The loading and saving of effect parameters.

All these tasks are indirectly associated with the first, because the user interface events are the reason why an effect parameter gets changed, and therefore something must be shown in the display device...

The digital framework (which includes the software routines), as was said before for the entity design, needs to be as generic and flexible as possible, in such a way that the number of effect blocks be indifferent to the general digital framework structure.

2.5 User interface

The user interface is one of the parts in which it is more difficult to obtain a "generic" design, since it is made of physical buttons, rotary switches, display devices, etc. which are not simple "programmable parameters". The proposal shown in Figure 8 is inspired in the KORG©'s AX1500G Toneworks [7] guitar multieffect.

Even if it is not possible to have a generic interface, some kind of *protocol* may be reached. For example, many effect pedals include a "*gain*" control. In order to help the user we could locate the gain control always in the first rotary switch (either the second, or the third, but always the same one).

These "agreements" or "rules" would be solely a recommendation –never an obligation– for the programmer, to make the interface the most intuitive possible.

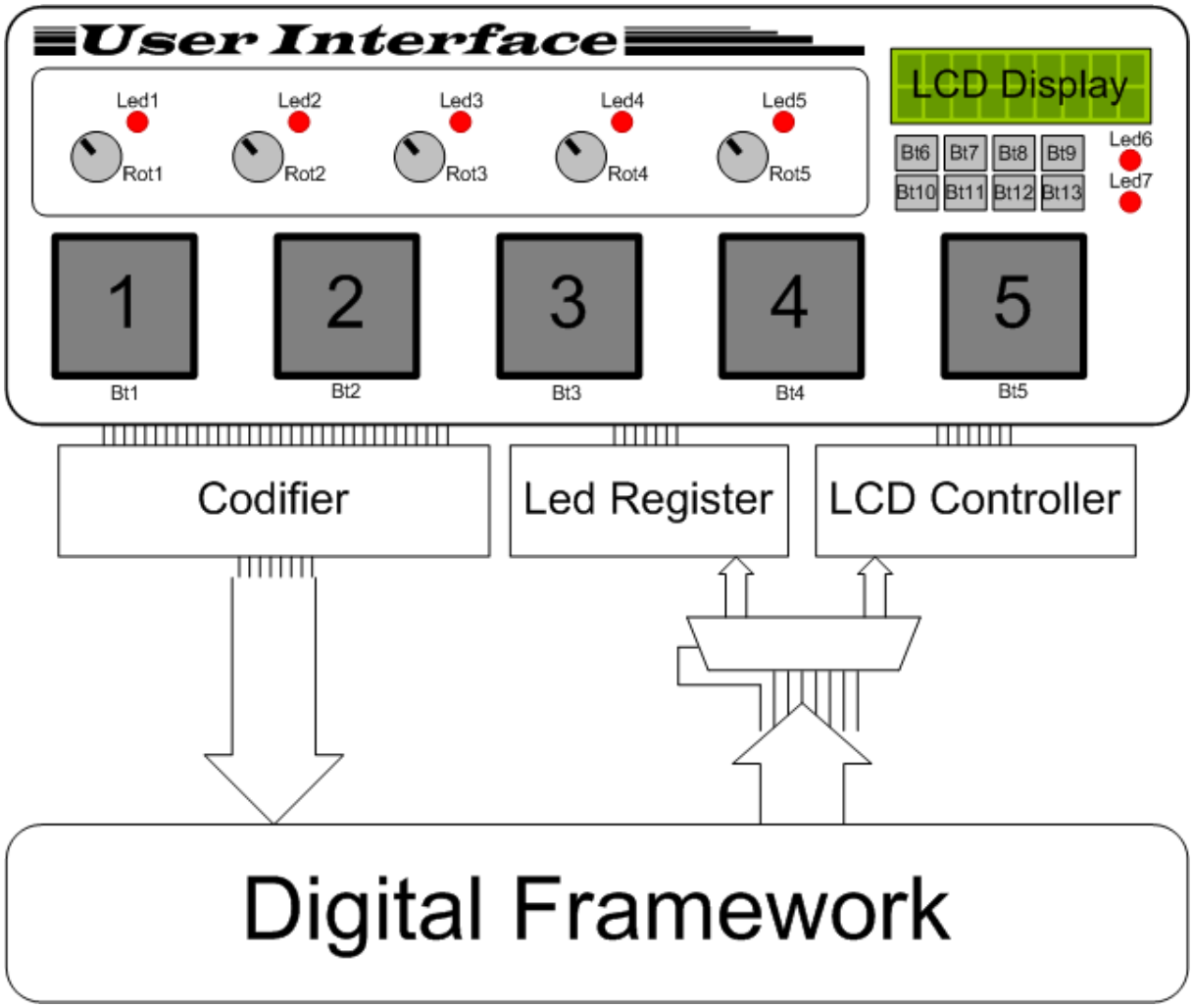

Figure 8: User interface diagram

The user interface design includes the *codes* and *protocols* needed to transmit the user interface events, and also the interface information. The choice of codes depends on the final design of the interface, and also the digital framework (overall the software routines), which is the responsable unit of managing the events.

3 Expansion and prototyping

The expansive purpose of this project is not to create a framework for digital effects. The framework is just a "tool" for developing, and a "physical support" for using the effects. The expansive idea comes with the effects themeselves, for three main reasons:

- The traditional secret technology becomes *public* and *free*.
- The hardware becomes software, by using HDL languages. Software distributing is more than just cheap, it is free.
- The separate effect blocks are independent, so they can be combined in many ways, and used in other physical supports.

Prototyping process will require an initial harder effort, overall for analog circuitry optimal design, and the processor program to manage all devices. If a robust framework is achieved, the research lines of this project may be optimally focused towards the design of new effect modules.

4 Conclusion

The analog audio effects are perhaps the best devices to obtain the highest audio quality, but the progress made in A/D and D/A converters has made the digital effects processor extremely successful. Using an open-source software-based idea to produce hardware (for this purpose) is the best way to create a wide community of developers and users.

Designing and establishing some *standards* in order to have compatible devices is an easy task. The design of a physical support for its testing and use is much more difficult as more time and prototyping stages are needed.

In any case, this document is an initial step towards that objective. Although nothing has been concretely accomplished as of yet, some powerful yet relatively easy to develop ideas have been put forward.

References

[1] **Wikipedia**
The free encyclopedia
`http://en.wikipedia.org/`

[2] **"Fundamentos de comunicaciones analógicas y digitales: Teoría"**
R. Pérez Jiménez, J.R. Velázquez Monzón, S.T. Pérez Suárez and S.I. Martín González
Universidad de Las Palmas de Gran Canaria

[3] **"Digital Audio Effects"**
(Edited by) Udo Zölzer
Wiley

[4] **ADS-1271 Analog to digital converter datasheet**
Texas Instrument datasheets
`http://www.ti.com/`

[5] **DSD-1794A Digital to analog converter datasheet**
Texas Instrument datasheets
`http://www.ti.com/`

[6] **PicoBlaze™ User Guide**
Xilinx
`http://www.xilinx.com/picoblaze`

[7] **KORG© Toneworks website**
KORG©
`http://www.korg.com/gear/default.asp?categoryID=6`