A Beginner's Practical Guide to Programming Digital Audio Effects on the E-MU10kx family of DSPs in the kX Project Environment

By Martin "Tiger" Borisov
and tips by Max Mikhailov

Preface

DSP programming is not hard, just needs a little time to get used to it!

This guide is for the complete beginner. You don't have to be a programmer to understand what all this is about. I decided to write it due to the fact that there aren't many resources at all on this topic not to mention for beginners. When I was a beginner myself (and I don't consider myself to be very advanced in the time of writing this, although advanced enough so you can trust the information in this guide) I had a hard time finding information and had to try to figure out stuff myself. Of course there is the As10k1 manual, but it is just too advanced for a total beginner, and there is the help file on Dane (a very convenient assembler), which is rather scarce and doesn't provide enough stimulus for a beginner. This is the main reason why there are just a few kX effect programmers, the regular guy, who hasn't any experience in digital signal processing is simply discouraged.

In this guide I'll try to keep things simple and give examples, which are the essence of learning. For convenience we'll use Dane (created by Max Mikhailov) which is very simple (meaning easy), although it doesn't give the flexibility of C++ programming, it will be enough for the basics and will give you a jump start on DSP programming.

The structure of the guide is more practicaly oriented. I'll give you examples coupled with explanations of the 16 instructions used in programming the E-mu10kx chips. You can copy them and test them in Dane.

In the end we'll "disect" several of the commonly used kX Project effects.

The Guide

What is digital audio and how the DSP works?

I won't go into detail how the dsp (digital signal processor, but it also stands for digital signal processing, I'll refer to both) works. When a signal (in our case analog audio signal) enters the input of the soundcard it goes into an ADC(analog to digital converter), which transforms the voltage of the sound wave in regular intervals of time (in our case it does this 48000 times a second, that is 48KHz) to a number, thus making it digital(discrete). So, 48000 numbers form 1 second of audio. Then every number of this constant array of numbers passes through the digital signal processor where it gets transformed by the effects. After that it goes into the DAC (digital to analog converter) where it gets analog again and goes to the souncard outputs.

So, we transform the signal with audio effects when it is in the form of numbers

and this means we have to deal with math and nothing else! So, dsp effects are just simple(well, sometimes a bit more advanced, I have to admit) mathematical functions and expressions, which most of us study in school. Dsp instructions (most of them) are just mathematical expressions like result = 1 + 0.1*0.5 or result =(1-0.1)*0.3 + 0.1*(-0.2). A combination of these forms a system, e.g. the effect. Simple, no? This system of mathematical expressions is performed on each number of the transformed signal - 48000 times per second. The order of instruction execution is from top to bottom.

Let's begin

Let's start by opening the kX editor. Rigt-click the kX icon in the taskbar and open the kX editor. This is Dane (Disasembler, Assembler 'n' Editor as Max M. put it, but there are some other "theories" about the origin of this name; it doesn't matter). Now you might want to take a look at the help provided for Dane if you want.

In the editor window you'll see lines starting with ; - these are just comments, in the upcomming examples there will be many comments. You'll see lines like "name", "copyright" and so on, which are only for providing information for the plugin and are not related to dsp. You can write your own stuff there. The "guid" line is important, because it guarantees the uniqueness of the plugin. You'll also see an "end" word, which says where the code ends. It always should be there.

NOTE: In Dane you can use both the decimal and hexadecimal(aka machine) number systems. Hexadecimal is default, so decimal numbers are automatically transformed to hexadecimal.

If you write 1 it will be transformed to 0x7fffffff. You don't have to know hex numbers!!!

Numbers processed by the E-mu10kx DSPs are fixed-point 32 bit fractional between -1 and 1 or integer values (whole numbers).

Registers

Data is stored in registers. You have seveal types of registers.

1. **Input** and **output** registers. Incomming data is stored in the **input** register, so it can be processed by the effect and processed data is stored in the **output** register, so it can be rooted to physical outputs or other effects.

Declaration:

input in
output out *; in and out are just names, you can assign them any name*

NOTE: the value of the output register can be reused, before the instruction cicle is over, when it gets overwritten with the next output value.

2.**Static** and **temp** registers. They are used for storing intermediate data during instruction execution. The value of a **static** register is preserved untill it is overwritten

(next sample cicle), or the microcode is reinitialized (reloading or resetting the plugin). **Temp** registers are used for the present sample (instruction cicle) only. The truth is that in kX they are treated as equal, so using **temp** registers won't give you any benefits or save resources. It is recommended that you use **static** most of the time. Despite that we'll use **temp** registers in the examples for learning purposes only.

Declaration:

static st
temp t ; *st and t are again just names*

Static and **temp** registers don't have to be initialized with a value, but if an initial value is needed you can assign such to a **static** register:

static st=0x2

NOTE: when you use certain numbers (constants) directly in the microcode which are not present in hardware (in a special read-only memory), they are automatically transferred to **static** registers when compiled.

3.**Control** register. This is a read-only (it can't get overwritten) register and has to be initialized with a value. When the code is compiled, a fader is automatically created for that register, so it can be user controlled. It can have values between 0 and 1 (although you can assign values greater than that, when you move the coresponding slider the value is automaticaly transformed between 0 and 1).

Declaration:

control volume=0.2

The assigned value is the default value.

4. **Constants**. Although this is not very important, there are certain constant values, which are defined in hardware in read-only memory and can be used directly in the code (not with a **static** register) for the purpose of saving some resources. Such are 0, 1, 2, 3, 4, 8, 10, 0x7fffffff ("fractional" 1), 10, 20, 100, etc. If not hardware-defined constants are used, they will be automatically transformed in **static** registers, as allready mentioned.

5. **Accum** register. In this register is stored the result of each instruction. When an instruction is executed, its result is **automatically** stored in it overwriting the previous value and then copied to the result register of that particular instruction. You can access the accumulator with the **accum** keyword. We'll take a look at it later.

6.**CCR (Condition Code Register)**. This one is used in the **skip** instruction. Its value is set after each instruction, based on its result. You can acces it with the **ccr** keyword. We won't pay much attention to it, because it's not widely used and this is a somewhat advanced topic.

7.**TRAM Access Data Register**. That's for delay lines. We'll deal with delays

later.

Instructions

All instructions have the following syntax, remember it:

instruction result **R** ,operand **A**, operand **X**, operand **Y**

Operands are basically registers.

1.**MACS and MACSN (multiply-accumulate with saturation on overflow)** That means that you multiply two numbers, then add them to a third number and store the value in the result register. These instructions operate with fractional numbers only (they perform fractional multiplication)! If the result exeeds -1 or 1 it is truncated to -1 or 1. This means that you can't use whole (integer) numbers with these two instructions.

Formulae:

MACS R = A + X*Y
MACSN R = A - X*Y

Example:

This is a simple volume control program.

```
name "Volume control";
copyright "Copyright (c) 2004.";
created "06/27/2004";
engine "kX";
guid  "...It will be automatically generated!!!..."; Don't copy this

input in
output out                ;we define the registers
control volume=1

macs out, 0, in, volume   ;out = 0 + in * volume
end      ;don't forget this
```

You can try it in Dane. Click on "Save Dane Source" (on the right of the window) not "Export to C++"! Save it, then right click on the DSP window and select "Register Plugin". Open the file and it should now be with the other effects - you know where they are.

2.**MACW and MACWN (MAC with wraparound on overflow)** Same as **MACS** and **MACSN**, but when the value exeeds -1 or 1 it wraps around. This is presumably to minimize noise when saturation occures with **MACS** and **MACSN**. If you have 0.5 + 0.7, the result instead of 1 on saturation, will be -0.8 with warparound.

3.**MACINTS (saturation)** and **MACINTW (wraparound)**. Same as **MACS** and **MACW**, but they perform integer multiplication. That means that you can multiply a fractional value with an integer value as well as integer with an integer. These two instructions always assume that the **Y** operand is an integer.

Formulae:

MACINTS R = A + X*Y
MACINTW R = A + X*Y

Example 1:

NOTE: I won't write the info part (name, copyright etc.) anymore. You can do that yourselves.

A simple gain. We multiply the input by 4 (that's a mono version of the **x4** effect).

```
input in
output out

macints out, 0, in, 0x4  ; out = 0 + in * 4
end
```

Example 2 :

If we want to control the amount of gain:

```
input in
output out
control gain=0.25
temp t

macints t, 0, in, 0x4     ; t = 0 +  in * 4
macs  out, 0, t ,gain     ; out = 0 + t * gain - its actually a volume control
end
```

4. ACC3. This instruction just sums three numbers. It saturates on overflow. The values of the operands can be all fractional (we treat the result as fractional) or all integer (we treat the result as integer).

Formula:

ACC3 R = A +X +Y

Example:

Mix of three mono sources plus a volume control.

```
input in1, in2, in3
output out
temp t
control volume = 0.33

acc3 t, in1, in2, in3  ;t=in1 + in2 + in3
macs out, 0 , t , volume  ; out = 0 + t * volume
end
```

5. **MACMV (MAC plus a parallel move).** The result of X*Y is added to the previous value of the accumulator. This can be useful for filters.

Formula:

R = A, accum += X * Y

Example:

See the **EQ Lowpass** explanation.

6.**ANDXOR** used for generating standart logical instructions. I haven't had any experience with this instruction and there doesn't seem to be much use of it. If you want more details take a look in the As10k1 manual. If anyone wants to add to this section, please contact me (info in the end of the document).

7. **TSTNEG, LIMIT, LIMITN** give the possibility of using something close to "if... then..." statements in the microcode.

Formulae:

TSTNEG R = (A >= Y) ? X : ~X

If A>=Y, the result will be X, else (if A<Y) X is complemented.

LIMIT R = (A >= Y) ? X : Y

If A>=Y the result will be X, else (if A<Y) the result will be Y.

LIMITN R = (A < Y) ? X : Y

If A<Y the result will be X, else (if A>Y) the result will be Y

Example:

A simple hard clipping fuzz. It cuts the wave over 0.05 and under -0.05, thus producing harmonics.

```
input in
```

```
output out
static negclip=-0.05, posclip=0.05  ;negative and positive clip limits
control volume=0.8
temp t

limitn t, posclip, posclip, in  ;if in>posclip, t=posclip; else t=in
limit t,negclip, negclip, t   ;if t<negclip, t=negclip; else t=t
macs out, 0, volume, t   ;volume control
end
```

8. **LOG** and **EXP**. **LOG** converts linear data into sign-exponent-mantissa (scientiffic) and **EXP** does the oposite. I won't go into more detail, because it is simply too advanced for beginners (and this is a total beginner's guide). If you want more info, take a look at the As10k1 manual.

Formula:

LOG R, Lin_data, Max_exponent, Y

Example:

One great use of this is for one instruction soft clipping (for guitar distrortion for instance). It is used in my **Chrome Amp** plugin and in **APS Fuzz**.

```
input in
output out
static exponent=0x2ccccccc
control volume=0.2
temp t

log       t,  in,  exponent,  0x0
macs     out, 0, t, volume
end
```

9. **INTERP -** this instruction performs linear interpolation between two points.

Formula:

INTERP   R = (1 - X) * A + X * Y

Example 1:

This is very useful for one instruction one-pole low-pass filter, which has the following formula:

out = coef * in + (1 - coef) * previous out

where coef is the filter coefficient.

```
input in
output out
control filter=0.5

interp out, out, filter, in ;the value of out is preserved for the next sampe cicle
end                        ;thus forming a one sample delay line
```

Example 2:

We can combine this with the soft clipping log instruction.

```
input in
output out
static exponent=0x2ccccccc
control volume=0.2, filter=0.5
temp t

log      t,  in,  exponent,  0x0  ;log soft clipping
macs     t, 0, t, volume   ;volume control
interp   out, out, filter, t   ;low-pass filter
end
```

Voila, we get a simple, but somewhat useful soft clipping fuzz effect!

10. **SKIP** this provides flow control in the dsp code. It skips a given number of instructions under certain conditions. This instruction is too advanced for beginners and we won't pay attention to it. There aren't many uses of it anyway. It has been used it the **Dynamics Processor** plugin (by eYagos). It makes use of the already mentioned **CCR**. Again, take a look at the As10k1 manual for more information.

```
SKIP   R, CCR, CC_TEST, number of instr to skip
```

Delay lines

For complete information on delay lines look in the As10k1 manual and Dane help. Here I'll discuss the practical side only.

In Dane we declare delay lines by specifying the number of samples in internal or external memory. Remember that 48000 samples form 1 second of audio data, so a delay line of 48000 will give us 1 second of delay.

We declare delay lines like that:

```
itramsize 6000 ;(0.125sec delay)
or
xtramsize 12000 ;(0.25 sec of delay)
```

We declare delay data registers like that:

idelay write *wr* at 0 *;wr is just a name;the write register is generally at 0*
idelay read *rd* at 12000 *;rd is just a name*
or
xdelay write... etc.

Example:

Simplest static 0.25 sec delay.

input in
output out

xtramsize 12000 *;we declare the number of samples(the delay line)*

xdelay write wr at 0 *;write data register at 0*
xdelay read rd at 12000 *;read data register at 12000*

macs wr, in, 0, 0 *;we write the value of in to wr*
macs out, rd, in, 1 *;we mix the input with the 0.25 sec delayed input in rd*
end

What are the benefits of C++ programing of effect plugins?

Full control of the register values (you can do anything you can think of), many MFC based GUI controls (buttons, checkboxes, etc.), use of a timer, skins if you have the nerves.

Although it is not absolutely necessary to program in C++, it gives you full control of all the aspects of the plugin. Nevertheless, all dsp calculations are left completely in the hands of the E-mu DSP chip, so real-time operation can be retained full-time.

Dane is very convenient for programming the plugins in C++, because it can generate a ready to use cpp file with the dsp code. You write the dsp part of the plugin in Dane and export it to C++ with the "Export to C++" command (right in the editor window). You can also test your work very fast with Dane, before you start messing up things in C++. That's always the best way, because writing the code directly in C++ can be a tyring task.

In the end, if you already have some knowledge on C and C++, you won't have any problems, because things are generally quite simple. If you don't, you'll perhaps need one month (if you are very enthusiastic) to learn the basics of C++, which will be enough.

You have to use Microsoft Visual C++ 6.0 or above.

Some simple effects "disection" and explanation

Let's start simple.

1.Stereo Mix

```
; Registers
        input in1L, in1R, in2L;
        input in2R;
        output outL, outR;
        control In1 Level=0x0, In2 Level=0x0;
        temp tmp

; Code
         macs   tmp,  0x0,  in1L,  In1 Level;
         ;tmp = 0 + in1L *  In1 Level
         ;this is a level control for the left input of  channel 1

         macs   outL,  tmp,  in2L,  In2 Level;
         ;oulL = tmp + in2L * In2 Level
        ;here we mix the left input of channel 1 to the left input of channel 2
        ; + level control for the left input of channel 2

         macs   tmp,  0x0,  in1R,  In1 Level;
         ;level control for the right input of channel 1

         macs   outR,  tmp,  in2R,  In2 Level;
          ;here we mix the right input of channel 1 to the right input of channel 2
          ;+ level control for the right input of channel 2

end
```

It is as simple as that.

2.Delay Old

```
; Registers
        input in;
        output out;
        control level=0x7fffffff, feedback=0x40000000, delay=0x1c20000;
```

```
        xtramsize 14400   ;we declare external tram size (number of samples)
; External TRAM delay line (14400 samples; ~0.300000 msec)

        xdelay write wrt at 0x0;   ;write data register
        xdelay read rd at 0x0;     ;read data register


; Code
        acc3   &rd, delay, &wrt, 0x0;
        ;here we produce the delay time by offsetting the rd addres register
        ;by the value of "delay" register, which is user controllable.
        ;We acces the adress register with "&rd"

        macs   out, in, level, rd;
        ;out = in + level * rd
        ;here we mix the input with the delayed rd with level control for rd

        macs   wrt, in, rd, feedback;
        ;wrt = in + rd * feedback
        ;here we create the feedback amount

end
```

3. EQ Lowpass, Bandpass, Highpass, Notch

The example is Lowpass, but they are basically the same, olny the coefficients are different.

NOTE: The values of the registers containing the filter coefficients are C++ controlled. You can't control them with the fader created by Dane only by copying the code. For our example we'll assume that they are static and not user controllable.

These filters are IIR (Infinite Impulse Response) and their convolution formula is:

```
y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2]
                   - a1*y[n-1] - a2*y[n-2]
```

x[n]= input
y[n]=output
x[n-1] = delayed input by one sample
x[n-2] = delayed input by two samples
y[n-1] = delayed output by one sample
y[n-2] = delayed output by two samples

This is a standart formula and we have to find a way to implement it on the E-MU chips in the kX environment. The implementation is the standart EQs - Lowpass, Highpass, Bandpass, etc.

Because they are stereo, there are actually two filters, so we can get rid of the second (right channel) part, because it is the same thing.

NOTE:. Remember that the values of static registers are preserved for the next sample cicle and can be reused untill they are overwritten. Thus they form a delay line from which we get the delayed samples for the above formula.

```
; Registers
        input inl       ; inr
        output outl       ; outr
        static b0=0x1b40d9, b1=0x3681b2, b2=0x1b40d9; filter coefficients
        static a1=0x7b4cfa5d, a2=0xc446023e, sca=0x2;
        static lx1=0x0, lx2=0x0, ly1=0x0;static registers for the delayed samples
        static ly2=0x0  ; rx1=0x0, rx2=0x0; we don't need this
        ;static ry1=0x0, ry2=0x0; and this, because they are for the right cahnnel
        temp t1, t2

; Code
         macs   0x0,  0x0,  0x0,  0x0;
        ;we null the accumulator, so previous values don't get mixed with the ones we
        ;need. Take a look at the accumulator explanation in the guide.
         macmv          lx2,  lx1,  lx2,  b2;
        ;b2*x[n-2]
         macmv          lx1,  inl,  lx1,  b1;
        ;b1*x[n-1]
         macmv          t1,  t1,  inl,  b0;
        ;b0*x[n]
         macmv          ly2,  ly1,  ly2,  a2;
        ;a2*y[n-2]
         macmv          t1,  t1,  ly1,  a1;
        ;a1*y[n-1]
         macs   t2,  accum,  0x0,  0x0;
        ;we copy the value of the accumulator to t2, so we can use it
         macints        ly1,  0x0,  t2,  sca;
        ;we give it some gain
         macs   outl,  ly1,  0x0,  0x0;
        ;we copy everything to the output
end
;we get rid of the second part, because its the same as the first
;        macs            0x0,  0x0,  0x0,  0x0;
;        macmv           rx2,  rx1,  rx2,  b2;
;        macmv           rx1,  inr,  rx1,  b1;
;        macmv           t1,  t1,  inr,  b0;
;        macmv           ry2,  ry1,  ry2,  a2;
;        macmv           t1,  t1,  ry1,  a1;
;        macs   t2,  accum,  0x0,  0x0;
;        macints         ry1,  0x0,  t2,  sca;
;        macs   outr,  ry1,  0x0,  0x0;
```

;end

That's all for now. I hope this guide helps you to start programming dsp effects. Ater you learn the basics, you might want to start studying more complex kX effects and search for dsp code on the web, which you can port to the kX environment.

Feel free to ask me ANYTHING! I won't laugh at your questions, you can be shure about that, because not long ago I was a complete beginner myself.

You can write me an e-mail - martintiger@abv.bg, or start a thread on the kX forum. I'm also open for suggestions, new ideas, criticism, and of course, if anyone finds something that's wrong, contact me at once!

Have fun.

References:

1. As10k1 Manual

2. Dane help by Max Michailov