soundart

APPLICATION NOTE
AN001

a simple audio level meter for the **chameleon**

revision 3 | 04.2004
chameleon S.D.K. v1.2

Introduction

We will implement a very simple but still useful application in this tutorial: a stereo level meter.

The DSP will calculate the envelope follower and the ColdFire will read its processing output to display two horizontal bars in the Chameleon LCD display corresponding to both R and L channels audio level values.

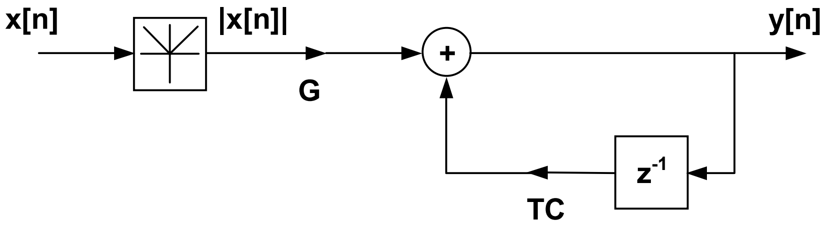
The level meter will monitor the device's audio input or output depending on the user's choice. When monitoring the output, volume changes will affect the level displayed, and when monitoring the input it won't. By pressing the "EDIT" key, the application will show the message "Monitoring Input" or "Monitoring Output" depending on the current state. By pressing the "VALUE UP" and "VALUE DOWN" keys, the user will be able to switch the monitor source. Simple.

Background

The level meter is based on an envelope follower system, whose flow diagram is shown in **Figure 1**.

Figure 1

Flow diagram for the envelope follower



The difference equation of the system is:

y[n] = G·|x[n]| + TC·y[n-1]

Where:

TC is Time Constant, which can be the Attack or the Release time, depending if the input signal tends to increase or to decrease:

- if |x[n]| >= y[n-1]

  TC = AT (Attack time)

- if |x[n]| < y[n-1]

  TC = RT (Release time)

G is the System Gain, whose value is:

G = 1 - TC

The attack time is selected to be as small as possible. Ideally AT = 0. A good working value is AT = 0.15

The release time is selected to be RT = 0.999

The DSP implements the difference equation. It processes each input sample to yeld an output envelope value. The two channel 24 bit values are truncated to 12 bit and concatenated into a single 24 bit word. The channel L corresponds to the 12 MSB, and the R channel does it to the 12 LSB. This has not a sensitive effect on the level display resolution and minimizes the communications overhead (This is not a problem in this application, since we have more than enough resources, but think on more complex real life programs).

The Coldfire application reads the envelope value each 20ms (Thus a kind of ratio 960 decimation is performed) and shows it in the LCD display in a logaritmic scale. This is due to the fact that the human eye won't be able to see smaller period value displays, so one sample each 20ms is more than enough. You could allways reduce the reading time to acheive higher precision.

Scilab simulation

We'll use the amazing tool SciLab to simulate our algorithm.

Start SciLab and load the file "levelmeter.sci" into the Scilab environment (Select File -> Getf and browse to the \Chameleon.sdk\src\examples\levelmeter\scilab). It contains the required functions to simulate the level meter application. You should change the SciLab working directory to that directory (use chdir()).

The functions in that file are:

```
function y = envelope(x,AT,RT);
```

Computes the evelope from an input x vector with the specified AT and RT time constants into the y output vector.

```
function levelmeter_test();
```

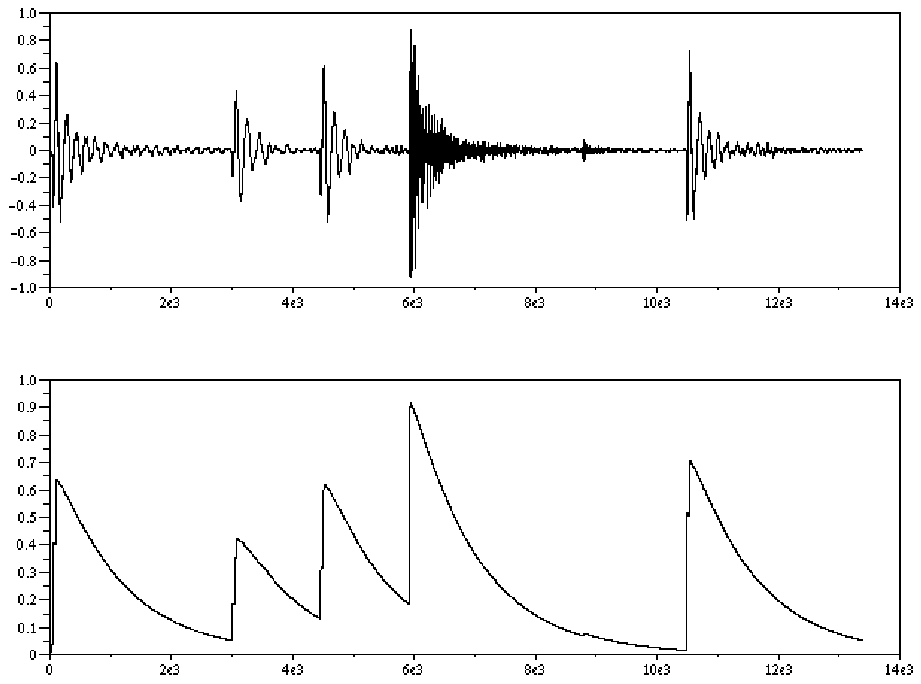
Type "levelmeter_test()" (without the quotes) at the SciLab prompt and see what it happens. That function uses the envelope() routine.

It loads the provided test file "test.wav", computes its envelope (only for one channel) with the default application coefficients and displays both signals (the input signal and its envelope). The waveforms displayed are shown in **Figure 2**. The above signal is the original input and below is the processed envelope.

Please take a look into these functions code to become more familiar with the SciLab language and programming environment.

Figure 2

Scilab plots of an audio input signal (above) and the its computed envelope (below)



DSP simulation

Once the algorithm is finetuned at theoretical level, it's time to start the DSP coding.

Start the Chameleon Development Envirnonment (CDE) and open the Level Meter project. Go to Project -> Open and browse to find the file \Chameleon.sdk\src\examples\levelmeter\levelmeter.chp. All the project souce code files will appear in the Project Browser window. You can see the main C source file, the main makefile, and the "lcd.h" header file which contains the symbols map to be displayed as the meter horizontal bars in the Chameleon LCD.

The Scilab simulation results are coded into DSP assembler in the files "levelmeter.asm", "main_loop.asm", and "process.asm".

At this level, the communication strategies are also considered. The file "levelmeter.asm" contains the main DSP application code. Here are defined all the necessary variables and all the initialization. For the peripherals, only the ESSI0 port (which comminicates the DSP with the audio codec) must be initialized to enable transmission and reception. We don't use interrupts, so no further initialization is needed. It's important to note that all the hardware setup was done for you by the system at startup, so you don't have to care very much about it.

"process.asm" contains the level calculation and the word concatenation routines. Finally the file "main_loop.asm" implements the DSP input sample reading, the call to the process functions and the output writing. Process is done sample by sample.

We added a 4th file, "main_loop_sim.asm", which is very similar to the main loop file, but this one is intended to repace it at simulation time. By giving the value of "0x01" to the SIMULATING label in the main code (in file "main.asm"), this file will be compiled instead of "main_loop.asm", and we will be able to load the compiled file into the Motorola DSP56303 Simulator to work with it. The only difference is that we don't have hardware specific code in it. The processing functions are exactly the same, so we'll be able to debug and profile them in the simulator.

If you are not very familiar to the Motorola DSP assembler and development tools, we encourage you to read the splendid tutorials provided by Motorola that you'll find with the Chameleon SDK in the folder \Chameleon.sdk\doc\dsp\other (files ONYXLABS.PDF and labscode.zip) before continue reading.

To compile the project, press F7 in the CDE. You'll see the compiler messages in the output window at the bottom of the main window. Make sure that the SIMULATING flag is set to 1.

Once our project is compiled with the SIMULATING flag set to 1, we will load the DSP code into the simulator (Start Menu -> Chameleon SDK -> DSP56303 Simulator or in the CDE select Tools -> DSP56303 Simulator).

***PLEASE NOTE**: Before to open the simulator, make sure that your PC local configuration is set as the decimal separator is a point ('.') instead a comma (','). Otherwhise, the simulator won't be able to read the text simulation stimulus files properly.*

In the simulator window, select File -> Load -> Memory COFF, and then press in the File button. Browse to find the compiled file "levelmeter.cld" to load it into the simulator, which is located in the Chameleon.SDK\out\model01\debug\examples\levelmeter folder. Open that file and finally press OK. You'll see the Level Meter DSP code loaded in the Assembly Wintow of the simulator. If you're working in release mode in the CDE, the cld file will be located in the Chameleon.SDK\out\model01\release\examples\levelmeter folder instead. To check whether you are in debug or release mode in the CDE, go to Build -> Set Active Configuration. The debug and release compile modes affect only to the ColdFire code, the DSP generated code is exactly the same.

Before to start the simulation, we'll load a text file as an input to simulate the DSP audio input data stream.

In SciLab, load the file "ioutils.sci" that you'll find in the "scilab" folder in this tutorial (use Getf() as before). It contains useful functions to read and write simulator files, and to translate wav files into text files and vice versa. Now type "wav2sim('test')". This function will convert the provided file "test.wav" into "test.io", which can be loaded into the simulator.

Back at the simulator window, now select File -> Input -> Open . Here we connect our binary file to the DSP ESSI0 input (address x:$FFFFB8). The input number is 1, the input is from a file, the file is connected to memory (memory space x, address $FFFFB8), the radix is fractional, and the filename is "test.io".

Finally we connect our output to another file that we'll be able to analyze later. Connect address y:$FFFFEF to the output file outpu.io (File-> Output -> Open). The output number is 1, the input is from memory to file, the radix is fractional, and the file is "output.io" (Place it in the same directory where the file "input.io" is. If the tools report that the file already exists, overwrite it).

NOTE: You can save these actions as macros for the simulator to avoid to repeat them any time you want to load and simulate your code with stimulus files. To learn how to use macros, please refer to the DSP Simulator Help.

Now we are able to simulate our code. In the disassembly window scroll to the code address labeled as 'End_Simu' and place a breakpoint. You'll see that there's a NOP at this address (the same NOP marked as dummy in the file 'main_loop_sim.asm'). Now run the simulator and wait a while. The simulator will execute the DSP code and will stop after processing 4095 input samples and generating 4096 output samples, which be stored in our output file. Once the simulator stops, close the output #1 (File -> Output -> Close).

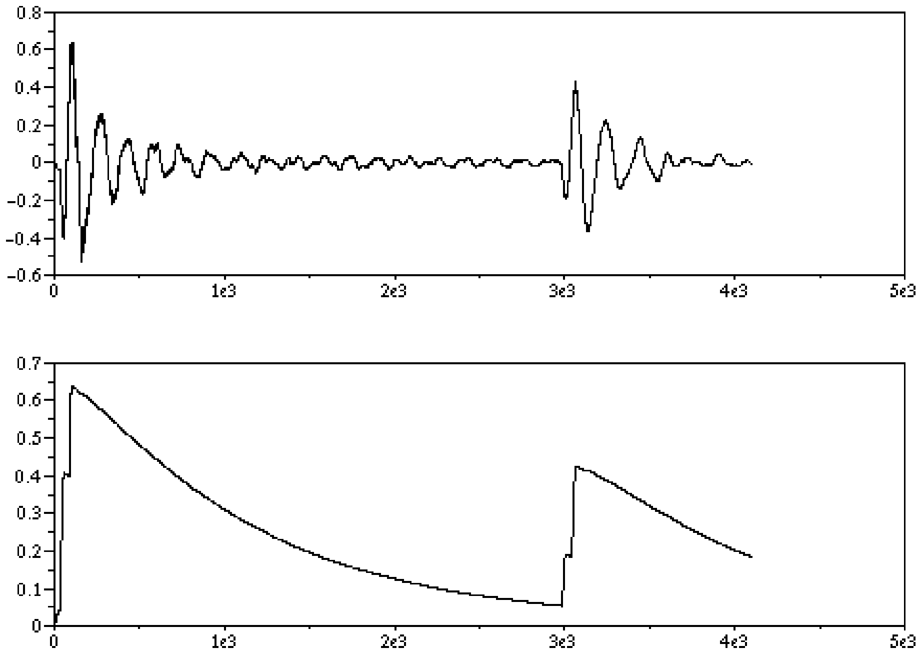
Back to SciLab again, we can analyze the results of our simulation. Type "y = loadsimst('out')" and plot one channel ("plot(y(:,1))"). It works!

Figure 3 shows the original file (above) and the file obtained from the simulator (below). Only the simulated size (4095 samples) is displayed, but is easy to see that the result is equal to the previously computed with the Scilab function (**Figure 2**) in these first 4095 samples.

You can now close Scilab and the DSP simulator. We have finished the simulation stage.

Figure 3

Scilab plots of the audio input file (above) and the result of the simulation of the DSP envelope follower code (below).



Chameleon implementation

Once tested, our assembler code is ready to be implemented into a complete Chameleon application, by adding the necessary ColdFire control code.

The DSP code consists of an endless main loop. The loop executes each two input audio samples (one for channel L and other for channel R). At the beginning of the loop, the last envelope values computed for each channel are sent to the ColdFire. Then the loop checks if a new volume has been received from the ColdFire in the Host port, and updates the volume variable if necessary. After that, for each channel the loop waits for the input sample and once it arrives the sample is multiplied by the volume, and its envelope is calculated. This envelope corresponds to the input signal (before aplying the volume) or to the output (after volume) depending on the status selected by the user. The ColdFire uses the Host Port Flag 0 to signal to the DSP whether it has to get the envelope of the input or the output.

The computed envelope values are “packed” into a single word ready to be sent to the ColdFire. Finally, a volume smoothing technique has been implemented. Instead of applying directly the volume changes received from the ColdFire, the current volume is progressively updated towards the target value. This way, sudden (and not so sudden) volume changes don’t cause audible “clicks” or “zipper noise” in the output signal when moving the volume potentiometer.

For the ColdFire control part of the application we will use two RTEMS tasks. The first one (*panel_task*) will take care of the front panel, and will constantly look for user input in some control (keys, pots or encoder). Once it detects some user input, it will perform the necessary actions, in this case it will respond to the movement over the Volume potentiometer and the "EDIT" key.

The second task (*level_meter_task*) will read the level values from the DSP host port and will display them in the LCD, performing some processing previously. To display the data in the LCD with the shape of a typical meter bars, we will redefine the 8 user definable characters, and map them into an array. The level values are the indexes into this array, and the array elements are the graphical bars corresponding to each level value. This way we will represent the numerical values graphically.

Before to create and run these two tasks, the DSP is initialized by downloading our code into it (calling the function *dsp_init()*). The compiler did translate the CLD compiled file into a C header file ("dsp_code.h") so we can access that code to download it.

Take a look at the files "main.c" and "lcd.h" in the CDE project. The code looks quite straight forward and easy to read.

Now set the SIMULATING flag to zero in the DSP main source file. Set the CDE configuration to Debug Mode (Build -> Set Active Configuration -> Chameleon Debug) and compile the project again. Now we can perform the most emotionant action: download your app into the Chameleon. Turn on the Chameleon and be sure that the RS-232 cable is connected to your computer. Start the Chameleon Toolkit (Start Menu-> Programs -> Chameleon SDK, or in the CDE select Tools -> Chameleon Toolkit), and press the "DEBUG" button and browse to the out folder where our compiled program is (\Chameleon.sdk\out\model01\examples\levelmeter\). Look for "levelmeter.elf" and click "open". Select the appropiate COM port and click "OK". The code will start to download. Once it finishes, you'll see the message "Audio Level Meter Example" in the Toolkit terminal window. The code is now running. Connect an audio signal to the Chameleon audio input and see how the application works. You should see the level meter bars in action!

To debug your app, you can use the TRACE() statement to display messages in the Toolkit window. Once your app is finished, compile it in Release mode and that's it. TRACE() directives won't execute in your code then.

Finally, you may want to store permanently this app in the Chameleon. In the Toolkit, pres the FLASH button. A dialog similar to the debug one will appear. Browse for file "levelmeter.cld" and press OK. The app will start to download. Once it finishes, the Toolkit will prompt you to actually store the application in FLASH. Confirm, and the levelmeter will be stored in the Chameleon.

Conclusions

This is a very simple application. Many improvements can be done to increase performance and functionality. This was just a sample application to illustrate the main aspects of the Chameleon applications development process, and to show you how easy this can be. The Chameleon development tools were introduced, the basic aspects of the communication between the Coldfire and the DSP (but not all of them!) were shown, as well as the most basic front panel controls handling into a Chameleon application, and some operating system features, such as tasks and rate monotonic services.

You could easily integrate that level meter application into your own more complex programs, as it consumes very little DSP power, and so adding them such interesting (and often fundamental) feature.

Keep on coding!