Design of an audio oscilloscope application

Fons ADRIAENSEN,
Casa della Musica,
Pzle. San Francesco 1,
43000 Parma (PR),
Italy,
fons@linuxaudio.org

Abstract

This paper documents some aspects of the design of zita-scope, an Audio Oscilloscope application for the GNU/Linux system. It is designed to permit accurate display and measurements on audio waveforms captured from any source via the Jack audio server. Topics covered include performance requirements, an analysis of some problems that need to be considered, and an overview of the implemention structure. The software will be available at the time this paper is presented at the 2013 Linux Audio Conference in Graz.

Keywords

linux, oscilloscope, audio measurement, time-domain, jack

1 Introduction

The oscilloscope has for a long time been a standard instrument for any engineer developing audio equipment, and in fact for almost everyone 'doing electronics'. In the all-digital era its importance in an audio related context may have declined a bit, except for debugging digital audio hardware. Fact is that many measurements on audio systems are better performed using spectral analysis or dedicated tools, but in some cases the ability to view the time-domain waveform and perform measurements on it remains essential.

Very few Linux applications for this use seem to exist. There are various 'scrolling scopes' which will display a waveform in real time, but don't permit any form of measurement. Some graphical synthesis environments include a 'scope' module or object, but these scopes are little more than a toy. They allow the user to see that a waveform is indeed a sine or a square wave, or to get an idea of the waveform envelope, but there it ends.

The only more ambitious application found by the author at the time of writing was something called *xoscope* [1]. After some patching it compiled, but it takes its inputs from */dev/dsp*, *EsounD* or some esoteric hardware only, doesn't know about ALSA or Jack, and the user interface really looks very dated. Probably its development has stopped years ago.

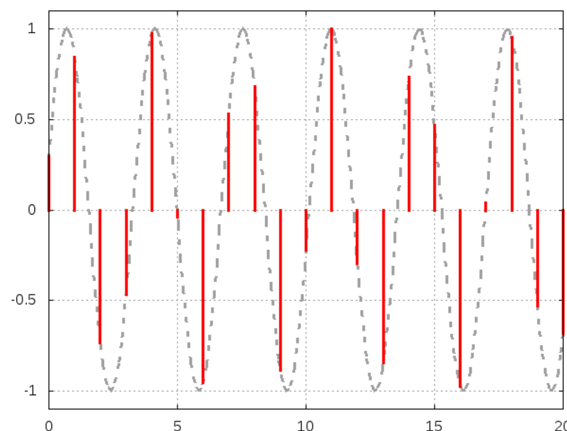

Figure 1: A sampled sine wave

Reasons for this state of affairs are clear enough: 'technical' applications (as opposed to those meant for creating music) are a minority interest, and actually creating a usable software scope isn't as simple as it seems — there is a lot more involved than just 'plotting the samples'.

2 Requirements

Displaying samples is what any serious oscilloscope application must *not* do. If a signal contains any significant energy above say 1/10 of the sample rate, the sample values provide a very bad or at least a quite unintuitive visual representation of the actual waveform. See for example Fig.1. After some training one may be able to recognise this as a 14 kHz sine wave sampled at 48 kHz, but in general it's near impossible to obtain any meaningful information from such a display.

Assuming a scope will be used to perform measurements and not just as a visual gadget,

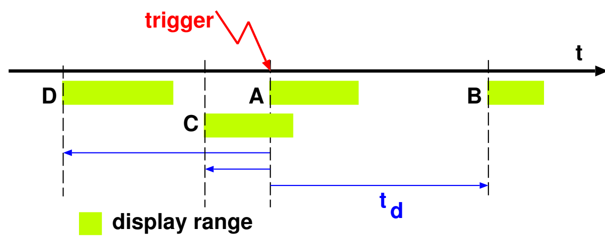

Figure 2: Trigger and display range

the following should be considered essential:

- An *accurate* and *stable* display of the analog waveform corresponding to a stream of samples.
- A wide range of *calibrated* display ranges and resolutions in both the time and amplitude domains.
- At least two and preferably more simultanuous channels.
- A *flexible* and *accurate* system allowing the user to capture particular events in an audio signal.
- The ability to store a signal and examine it at all available gain and time resolution settings.
- Calibrated markers to aid accurate measurement.
- Responsive user controls, e.g. changing display parameters should produce an almost immediate result.

And less essential but nice to have:

- Facilities to perform more complex measurements, e.g. the RMS value of a range, spectrum, etc.
- Remote control, allowing the applicatin to be configured by and report to automated test systems or scripts.
- 'Reasonable' CPU and other resource usage.

3 Problem analysis

3.1 Triggering

While a scope can be used in *free running* mode, in most cases a *triggered* display is essential. The principle is illustrated in Fig.2. The user will select a trigger condition, for example a positive going zero crossing. The start of the displayed range will then be at a fixed offset t_d from that point, selected by the user. In many cases the trigger point will be the start of the displayed range (case **A** in the figure), but even most analog scopes offer a *delayed trigger* option (case **B**), with a delay that can be much longer than the displayed range. A digital scope can easily store the signal, and allow to display part of the signal *before* the trigger (cases **C** and **D**). This is very useful when the trigger condition is the consequence of something that happened before and which the user wants to investigate.

Triggering can be *continuous* or *single shot*. In the first case, if a trigger has been found, and as soon as enough signal has been captured to fill the display and all of it is processed, the system can start looking for the next trigger and the cycle repeats. This could result in a very high update frequency (if the display range is short and close to the trigger) which would just lead to an excessively high CPU load without improving the visual result. In such cases looking for the next trigger should be delayed by 50 milliseconds or so.

In the *single shot* mode, signal capturing will stop at some point after the displayed range, allowing the user to examine all of the stored signal. In that case, the position of the trigger point in the stored buffer becomes a parameter that should be controllable by the user — this determines how much he/she will be able to scroll forward or back from the initial display range.

The usual trigger condition is the signal crossing a given value in a specified direction, up or down. This point needs to be determined with high accuracy. Consider the following conditions: we are looking in continuous trigger mode at some high frequency waveform, with a display range of 50 microseconds (one period at 20 kHz). Assume the display is 1000 pixels wide. Then each pixel corresponds to 0.05 microsecond, and if we want a stable display the jitter on the trigger position must be at least ten times smaller than that value, say 5 nanoseconds or around 1/4000 of a sample at a sample rate of 48 kHz. Simple linear or even cubic interpolation on the original samples won't be sufficient to achieve this .

The solution used in *zita-scope* is to first upsample the signal selected as the trigger source by a factor of 5. This means that even in the

worst case — a sine wave near half the sample frequency — in each half cycle there will always be samples covering the range of -0.95 to 0.95 times the amplitude, and triggering within that range will be reliable. Assume the trigger level is V with the signal going up. We scan the interpolated waveform for two consecutive samples v_0 and v_1 such that $v_0 \leq V \leq v_1$. When these are found, the signal is locally upsampled by a factor of 25, and we search for v_0, v_1, v_2 and v_3 such that $v_1 \leq V \leq v_2$. Given these we can find the best fitting parabola $f(x) = ax^2 + bx + c$ with $f(0) = v_1$ and $f(1) = v_2$. Solving the quadratic equation then provides the exact location of the trigger point, with a worst case error of around 1/100000 of a sample at the original sample rate. The calculations are quite simple but require some attention to cover special cases, e.g. the quadratic coefficient could be near zero. Four points rather than three are used to provide an estimate of the quadratic term at the center of the interval $[v_1, v_2]$.

Another option, usually not available on analog scopes, is to trigger on the first positive or negative peak exceeding a given value. This can be done using a similar method, in this case searching e.g. for three samples v_0, v_1, v_2 with $v_0 < v_1 > v_2$, and then solving the derivative of the quadratic equation.

The first release of zita-scope can have up to four displayed channels, and each of those can be the trigger source. Also a separate trigger input is provided. This can be used in the way described above, or it can be put in 'digital' mode, meaning that the trigger position will be the first sample crossing a given value, e.g. an impulse provided by some external software.

Another option is the *manual* trigger mode. Clicking a button in the GUI generates a single sample pulse on a trigger output, and the trigger point is exactly one period later (looping the pulse back to the digital trigger input would give the same result). This can be used to measure e,g, the impulse response of a filter.

Some other modes could be useful, for example triggering on a MIDI note-on event delivered via Jack-midi, for example to test the latency of a soft synth, or on Jack transport reaching a preset value. These could be built-in, or provided by a separate app connected to the external trigger input.

3.2 Waveform display

As already illustrated by Fig.1, displaying the waveform corresponding to a sampled signal involves more than just plotting the sample values. A digital audio scope could have a horizontal scale ranging from a second per grid division down to a microsecond, a range of one to a million. In all cases the user wants to see a more or less accurate representation of the waveform. For an analog scope this is no problem as both the signal and the display device have 'infinite' resolution. For a digital scope we need to consider that the waveform is sampled and the display consists of discrete pixels.

The first question is which graphics library will be used. On Linux, the choice is between the basic X11 drawing routines and Cairo [2]. GUI toolsets offering a 'canvas' object will also use one of these. X11 graphics are defined entirely in terms of pixels. Cairo offers subpixel coordinates and anti-aliased line drawing. This provides a much better visual quality, but not a higher resolution.

On recent multi-core hardware there is really no reason for not using Cairo or something similar. The situation is different if somewhat older systems are considered, e.g. a single core 2 GHz Pentium 4. On such hardware, when drawing four waveforms 20 times per second on a full screen window, using Cairo can easily take the CPU power to its limits.

The solution adopted in zita-scope is to provide both. By default Cairo will be used in all cases, but there is an option to use X11 when the display is updated at a high frequency, automatically switching to Cairo in all other cases.

Assume the display is showing one or a few cycles of a sine wave, so each cycle has a nontrivial width on the screen. An accurate display of say 1000 by 1000 pixels requires something like 70 points per cycle in that case. This ensures that the extreme values shown are no less than 0.999 times the real peaks (i.e. less than half a pixel error), and the waveform doesn't look like a series of connected straight lines. Since the frequency could be near half the sample rate, this would require upsampling by a factor of at least 35.

A brute-force technique would be to always upsample by a factor of at least 35 and plot all the points. But this would be very inefficient in almost all cases. Consider a display that is 100 ms wide — this would mean 168000 points after resampling, and most of the effort spent com-

puting and displaying them would be wasted as the display doesn't have the resolution required to show all that detail. Clearly some better idea is needed.

To get a grip on the issues involved we will use the following parameters:

- F_{sig} : the original signal sample frequency, e.g. 48 kHz.
- F_{pix} : the pixel frequency. For example if we have 1 millisecond per division and a division is 100 pixels, then F_{pix} is 100 kHz.
- F_{res} : the sample frequency after upsampling.

Zita-scope uses two different algorithms and display routines, depending on some of those parameters.

If $F_{pix}/F_{sig} \geq 35$, we compute one sample per k pixels on the x-axis, with k integer. These points are then plotted as a sequence of straight lines. This provides the best that can be done when using X11 (unless we would implement some ad-hoc anti-aliasing scheme), and Cairo will show a smooth anti-aliased line. In this case we have:

$$\begin{aligned} k &= \lfloor F_{pix}/(35 \times F_{sig}) \rfloor \\ F_{res} &= F_{pix}/k \end{aligned}$$

In practice the value of k is limited to some small value (currently 5, so there will be at least one point every 5 pixels) to avoid having too long straight lines.

In the other case, if $F_{pix}/F_{sig} < 35$, each x-axis pixel is assumed to represent a *range* of time, and we compute the minimum and maximum values the signal will take within that interval. The resulting data are then plotted as a series of vertical lines, one for each x-axis pixel. For X11 this is again more or less the best we can do. But this scheme doesn't work well when using Cairo if the signal doesn't contain significant high (relative to F_{pix}) frequency energy, and the resulting plot is reduced to a line instead of being a broader band of pixels. The result isn't much better than for X11 as we have in effect disabled Cairo's anti-aliasing capabilities. This situation arises if the waveform is monotonic within each time interval represented by a single pixel. Fortunately there is an simple solution, which is illustrated in Fig. 3.

In the right half of (a) we have a waveform that can be *assumed* to be representable by a

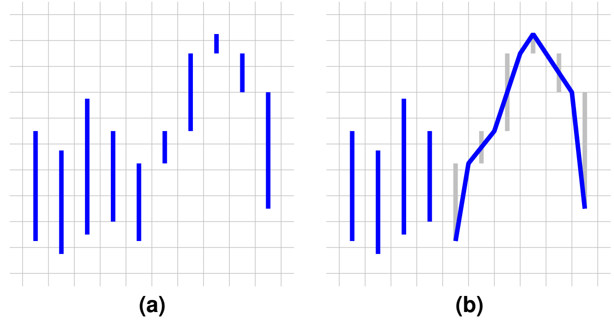

Figure 3: Connecting segments

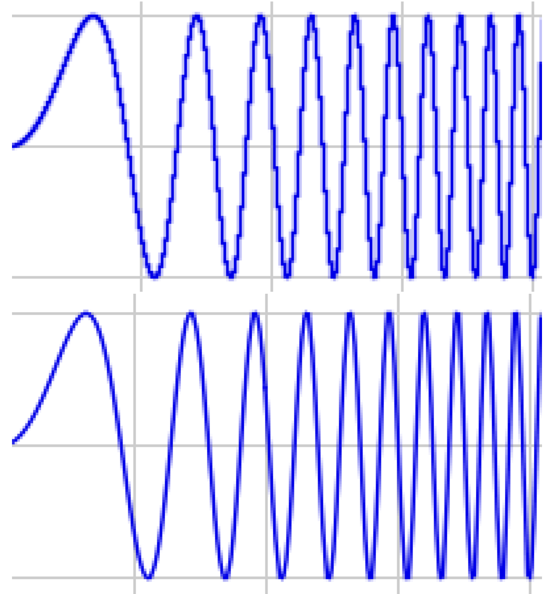

Figure 4: Visual effect of connecting segments

smooth line. In this case we can replace the vertical segments by connected lines just by moving the x-coordinates by half a pixel, and splitting the vertical segment at an extreme into two lines, as shown in (b). This only requires the original x, min, max data, and results in a dramatic improvement in display quality, as illustrated by Fig.4.

To compute the min, max pairs the display algorithm upsamples the original data by a factor of at least 6, and such that we have a sample on every border between two adjacent pixels — this ensures that there will be no gaps between segments. The extreme values can then be found using inverse quadratic interpolation. This is essentially the same algorithm used to trigger on a peak, except that the function value is computed instead of the argument, and considerably less precision is required.

In this case we have

$$\begin{aligned} k &= \lceil 6 \times F_{sig}/F_{pix} \rceil \\ F_{res} &= k \times F_{pix} \end{aligned}$$

The table below shows the resulting display parameters as a function of the horizontal resolution, for $F_{sig} = 48$ kHz, and 100 pixels per division. The SPP value is the number of samples (after upsampling) per horizontal pixel.

T/Div	F_{res}/F_{sig}	SPP
1 s	6.000000	2880/1
0.5 s	6.000000	1440/1
0.2 s	6.000000	576/1
0.1 s	6.000000	288/1
50 ms	6.000000	144/1
20 ms	6.041667	58/1
10 ms	6.041667	29/1
5 ms	6.250000	15/1
2 ms	6.250000	6/1
1 ms	6.250000	3/1
500 us	8.333333	2/1
200 us	10.416667	1/1
100 us	20.833333	1/1
50 us	41.666667	1/1
20 us	52.083333	1/1
10 us	52.083333	1/2
5 us	41.666667	1/5
2 us	104.166667	1/5
1 us	208.333333	1/5

In this example the switch between the two algorithms discussed above occurs between 100 and 50 usecs per division.

Note that in both these cases one sample per pixel is computed, but in a different way. For the first algorithm the single sample corresponds to the center of an horizontal pixel. For the second it is positioned on the border between pixels.

To obtain this exact alignment of the upsampled signal to the pixel grid we must initialise the phase of the polyphase filter used by the resampling algorithm to the required value. The current release of *zita-resampler* includes support for this.

4 Software structure

4.1 Data flow

Figure 5 shows the main elements of the implementation. Almost no work is done in the Jack callback, it just copies the input signals to a lock-free buffer. Apart from that it contains some code to support the manual trigger mode. All the rest is done in a non real-time context, so zita-scope will impose only a very light load on the Jack processing graph.

The lock-free buffer is around 1.5 seconds long. In single-trigger mode input is discarded until the user enables the next trigger, but the lock-free buffer it is used to store the last second of input. This ensures that this data is always available at the next trigger (which may be a manual one).

The trigger logic determines which part of the input is copied to the capture buffer. In continuous mode this will be little more than the displayed range — if the user changes the trigger position w.r.t. to the display range this is taken into account on the next trigger. In single-trigger mode the capture buffer can store up to a few seconds of data, allowing the user to examine any part of it. To allow triggering on a wide range of signal levels the input gains set in the GUI are taken into account by the trigger algorithms, but the signals written to the capture buffer are always the original ones without any gain applied.

The following step implements one of the two algorithms presented in the previous section, depending on the selected display range. These computations are performed when the contents of the capture buffer are updated by the trigger logic, or 'on demand' when the user changes the time axis parameters.

The plotting routines finally display the data on the screen. Any gain and vertical offset selected by the user are only taken into account at this point, so changing the these parameters does not require recomputing the display buffer data.

Some logic and state machinery is required to coordinate all of this. For example, in single trigger mode the display must be redrawn immediately if the user changes any parameters, while in continuous mode it could be better to wait until the capture buffer is updated.

4.2 Display markers

To perform accurate measurements zita-scope offers various types of on-screen markers, shown as vertical or horizontal dotted lines on the display. Their absolute and relative positions are also shown in numerical form. These numerical values are always computed from the original signal stored in the capture buffer, not from the

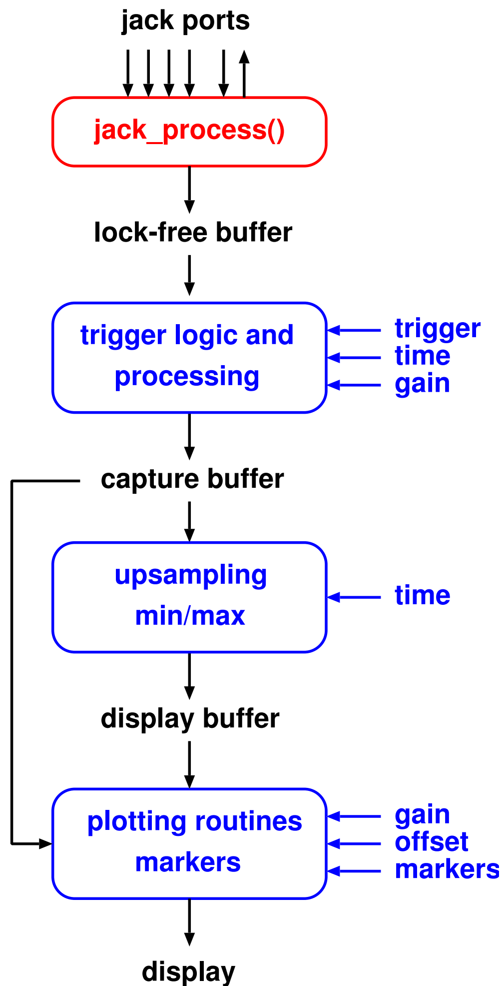

Figure 5: Processing flow

display data, and are not modified by any gain or offset settings.

Time axis markers can be positioned manually, or snap to a zero crossing or a peak, using the same algorithms as for triggering. Amplitude axis markers can be set manually, or they can follow the time axis ones on a selected channel, or snap to exact peak values. More complex measurements (RMS levels, spectrum,. . .) may be implemented in future releases of the application.

4.3 Additional facilities

Zita-scope offers some additional convenience functions:

- Storing and recalling the complete state of the instrument, including the capture buffer. The data is stored as a regular CAF audio file with the instrument settings in a dedicated GUID chunk.
- Creating a PNG file of the current display. For images to be included in printed documents the display background can be changed to white.

5 Acknowledgements

The author has contemplated writing an oscilloscope app for years, but kept postponing it until some Linux audio users got impatient and 'increased the pressure'. Without them zita-scope probably wouldn't exist.

Writing this application in the relatively short time it finally took was possible only because of the existence of some excellent and well-documented software taking care of some aspects, in particular Jack and Cairo.

A sincere thanks also to the (near future) beta-testers who will without doubt provide invaluable feedback and suggestions for improvements.

References

[1] T. Witham and B. Baccala, "Xoscope for Linux." `http://Xoscope.sourceforge.net/`, 2009. [Accessed 27/1/2013].

[2] K. Packard *et al.*, "Cairo." `http://www.cairographics.org/`. [Accessed 27/1/2013].