Using a DLL to filter time

Fons ADRIAENSEN
Alcatel Space
fons.adriaensen@skynet.be

Abstract

A new mechanism to obtain an accurate mapping between samples and system time was recently introduced into JACK[1].It is based on a the use of a Delay Locked Loop (DLL). This paper discusses the problem that was solved, and introduces the reader to the concept of control loops in general, and to the DLL solution adopted in particular.

Keywords

DLL, synchronisation, filtering, JACK, Ardour

1 Introduction

In audio applications it is often required to have an accurate mapping between the time a particular sound is reproduced or recorded, and the actual samples that make up the signal. For example, a synthesis system such as SuperCollider will receive commands to produce a sound at a given system time[2], and has to find out some in way which samples are going to represent that sound. Other places where this relation is important include DAW software such as Ardour, where one often needs to find out which sample is audible 'now', for example when the user clicks a mouse button. The techniques explained in this paper can be used for other purposes as well, for example for tempo tracking.

To be useful, the mapping between system time and sample count needs to have some qualities. It has to be:

- **Smooth.** This means it should maintain a 'constant speed'. We expect a given difference in time to always correspond to the same distance in samples.
- **Monotonic.** If event A comes before B in real time, we want A to map to a sample that comes before the one that B maps to.
- **Continuous.** If two events are very close together, we expect them to correspond to samples that are close together too – there should be no jumps.

Typically, audio processing is done in *periods* — samples are processed in blocks of fixed size. In particular, all JACK-enabled applications work this way, and the situation is not really different when using ALSA directly. There is no hard relation between the time a given block of samples is actually available for processing, and the time the signal represented by those samples will exist, or existed as a phsysical sound or as an analog electrical signal. Once a signal is represented by data in computer memory, its real timing has become data too, which in some way has to be stored along with the samples.

We can, each time we start processing a period, read the system timer. This will give an approximate idea of when (in system time) the sound was or will be audible. Doing this, there are five more factors to consider:

- **Latency.** This is the time between the sample was (or will be) converted from (or to) to the analog domain (or to a format that has physical timing), and the time the audio hardware will generate an interrupt that will trigger the processing of the block that this sample is part of. Latency as defined like this is only a function of the amount of buffering performed by the audio hardware, and will be not be considered further in this paper.
- **Delay.** There will be some delay between the HW interrupt, and the momemt we can read the system time. This is made up of interrupt, processing and scheduling delays.
- **Jitter.** The delay mentioned in the previous point will not be constant. Advances

[1]This paper was written before the actual implementation. The code that is currently in JACK does not use system time but JACK's internal *frame time*.

[2]More accurately, an OSC time, but this has a fixed relation to system time.

in Linux kernel design and implementation have greatly reduced this variation, but it still exist, and will remain. A typical well-tuned system will show a small average delay with some occasional peaks.

- **Timer quantisation.** The system timer may have a considerable quantisation error, for example it could increment in steps of a millisecond. When the period time is an exact multiple of the timer step, this will generate a constant error. In the other case timer quantisation manifests itself as additional jitter.
- **Sample frequency errors.** On most hardware, the sample clock is not locked in any way to the one that drives the system timer. This means that the real sample frequency (when measured against system time) will not be exactly equal to the nominal one, and that any mapping based on the nominal sample frequency will show an error.

Latency can be compensated for when we know the driver and HW configuration. The following sections will show that jitter, quantisation, and sample frequency errors can be removed as well. The only remaining error then is the *average* interrupt to timer read delay. On a system configured for audio work this should be small, and a constant offset will remove most of it.

2 Mapping between sample counts and system time

Let p be the period size, F the nominal sample frequency, and $T = 1/F$ the nominal sample period. At the start of a period i, we are at sample n_{0_i} (this value will increase by p for each period), and we read the system timer and obtain t_{0_i}.

With this information, we have a mapping between system time t and sample index n:

$$t = t_{0_i} + (n - n_{0_i}) * T \quad (1)$$

$$n = n_{0_i} + (t - t_{0_i})/T \quad (2)$$

Modulo some implementation details, these two equations describe how JACK's `frame_time()` function [3] operated originally.

[3] Names such as *frame_time* can be quite confusing. Is it the time of a frame, or some other time expressed in frame units ? In this paper we'll try to use a consistent naming convention: the phrase *A_time of B* always means "the time of the event B, expressed on the time scale A".

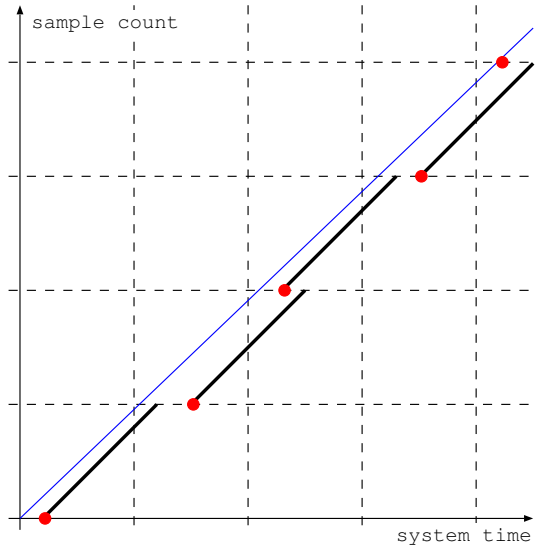

Figure 1: Discontinuous mapping

There are some problems with this mapping, as shown in fig.1. In this figure the grid represents nominal periods on both axes, and the (red) dots are the points (t_{0_i}, n_{0_i}). The thin (blue) line is the exact mapping we want. Note that in this example the real sample frequency is slightly lower that the nominal — the thin line is lagging the grid as time advances. The thick lines show what we get when using the equations above. The mapping is smooth, but only within one period. It is certainly not continuous and can even be non-monotonic. At some points it is ambiguous or undefined.

What we want is more something like fig.2. Here the thick lines are connected, and the mapping is continuous and monotonic. We will also want to remove as much as possible of the 'wobbling', i.e. obtain a straight linear mapping.

Define n_{1_i} and t_{1_i} as our estimates of the sample count and system time at the start of the *next* period. Of course $n_{1_i} = n_{0_i} + p$, and our best guess for t_{1_i} so far is $t_{1_i} = t_{0_i} + pT$.

The continuity requirement means that:

$$n_{0_i} = n_{1_{i-1}} \quad (3)$$

$$t_{0_i} = t_{1_{i-1}} \quad (4)$$

Equation (3) is already satisfied, and (4), in practice, means that we should not try and find t_0 at the start of each period, but that we must

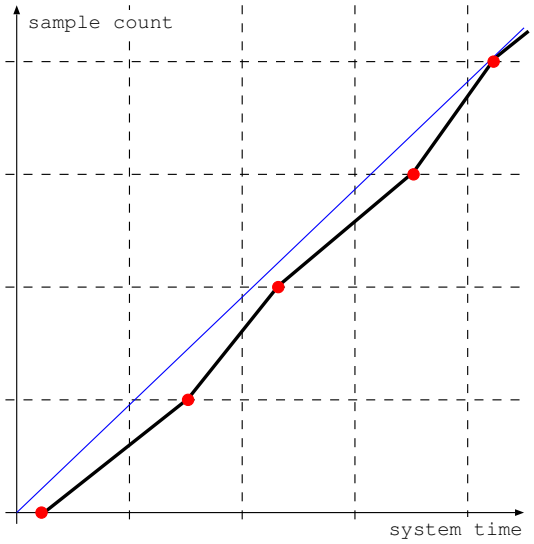

Figure 2: Continuous mapping

look ahead and find t_1, and use the t_1 from the previous period as t_0 in the current one.

Using these definitions, the mapping equations (1,2) above become:

$$T_e = (t_{1_i} - t_{0_i})/p; \quad (5)$$

$$t = t_{0_i} + (n - n_{0_i}) * T_e \quad (6)$$

$$n = n_{0_i} + (t - t_{0_i})/T_e \quad (7)$$

This means that we have replaced the nominal sample period T by an estimated one, T_e. As a result of the jitter on the system timer, this T_e shows considerable variation. The following section will show how this can be reduced, thereby straigthening the thick line in fig.2. Note that it will always remain at a small distance to the right of the ideal mapping. This is the average delay mentioned above.

3 Control loops

The problem we face when trying to remove the timing errors is one of *filtering*: we want to remove the random fluctuations but to follow the 'average speed' of time.

In electronics and digital signal processing many filtering problems are solved by feedback loops. The classical example is the Phases Locked Loop or PLL, which enables a radio receiver to track the frequency of the signal it is receiving even when that signal is erratic and corrupted by much noise.

Now what is a control loop ? A good example is what happens when you are driving and you want to follow another car at a constant

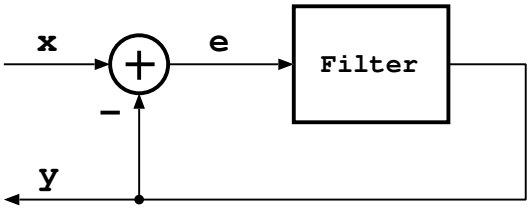

Figure 3: A general control loop

distance. When you notice that the distance increases, you will accelerate. When you come too close, you will decrease your speed. The thing that drives this mechanism is the difference between the distance you observe, and the one you want to maintain — this is called the *loop error* in feedback theory.

All loops operate in the same way (see fig.3): there is some input quantity x and an output y that tries to follow x, and y is driven in some way by a filtered version of the error $e = x - y$. To apply this to our problem, let x be the jittering system time we are reading at the start of each period, and then y is supposed to be a smoothed version of the same.

In control loop theory a very important parameter is the *loop order*. The loop order determines in which way the loop takes time into account. For example, one way to react to a given loop error would be to just use the value of the error as it is, without taking into account its history. That would be a zero-order loop. Another way would be to increase the effect of an error as it persists for a longer time – a first or higher order loop will do exactly that.

More formally, the loop order is given by the number of integration steps in the filter. What is an integrator ? Basically a thing that outputs an accumulated (over time) version of its input. If x_i are the inputs, then the outputs are

$$y_i = y_{i-1} + x_i \quad (8)$$

In C this reduces to `y += x;` An integrator has the interesting (for control theory) property that when its output is in some way constrained to remain bounded, for example because it is part of a feedback loop, then the average value of the input must be zero. So if there is at least one integrator in the loop filter, and the average speed of the input is zero, then the average loop error must be zero as well.

Figure 4 shows the structure of a zero, first and second order loop. In a zero-order loop we

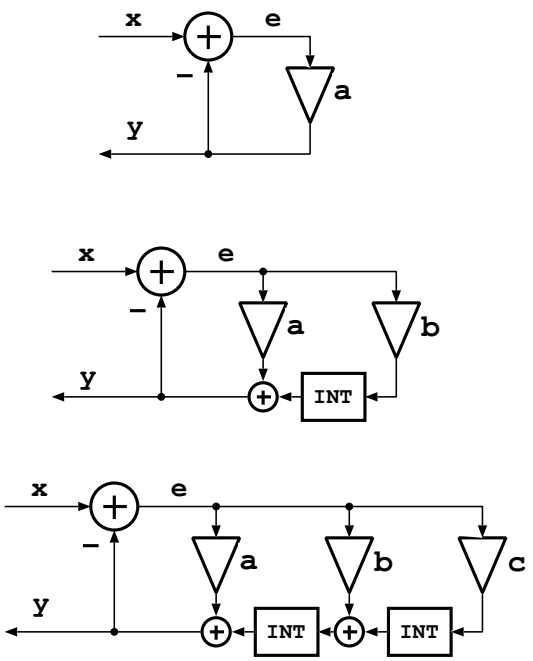

Figure 4: Zero, 1^{st} and 2^{nd} order loops

have just $y = a * e$, so the average error will be zero only if the input has the same property.

In a first order loop there are two feedback paths, one direct, and the second one through an integrator. This is equivalent to a single pole lowpass filter. The average loop error will be zero if the average input speed is zero. The car chasing example given above is also a first order loop. The loop error is expressed as *distance*, but this error is used to control your *speed*. Distance is the integral of speed, so there is an integrator in the loop.

In our problem x is moving at a nonzero speed, so the condition to have zero average loop error is not satisfied. We could try to solve this by subtracting the integrated speed of system time from x and adding it again to y. But since the nominal sample frequency is not exact, we do not know this speed exactly — the difference in system time between two interrupts is not exactly equal to pT. So some error will remain. What is worse is that this error will increase as we decrease the filter bandwidth in order to remove more of the jitter. In a practical situation this rapidly leads to unacceptable error magnitudes.

The solution is to use second order loop. In this type of loop there is is one feedback path that passes through two integrators. This will give a zero loop error if the input x has zero acceleration, which is the case in our DLL. The circuit is equivalent to a two-pole lowpass filter, such as used for example in a synthesiser.

This presentation is not the proper place to repeat standard control loop theory — for a complete treatment, see the classic works of Gardner (Gardner, 1966) and Best (Best, 1984). Both authors mainly discuss phase locked loops, but most of the theory is directly applicable to a DLL as well.

We will only give here the equations for a, b and c in the second order loop shown in fig. 4. Let F be the sample frequency of the loop (i.e. the period frequency) and B be the required bandwidth of the filter, then:

$$\omega = 2\pi B/F \quad (9)$$
$$a = 0 \quad (10)$$
$$b = \sqrt{2}\omega \quad (11)$$
$$c = \omega^2 \quad (12)$$

Here b is set to give a critically damped loop.

4 Implementation

The following code illustrates how the DLL can be written in C. The code shown here is a straightforward implementation of the second order loop in fig.4, using the same symbols as in the previous sections. Times are expressed in seconds. Referring to the second order loop in fig. 4, $a = 0, x = read\_timer(), y = t1$, and $e2$ is the state of the second integrator.

First we need some constants:

```
nper = period_size;
tper = period_size / sample_rate;
```

The first iteration sets initial conditions. This is executed in the first process cycle when JACK starts, or after an xrun:

```
// init loop
e2 = tper;
t0 = read_timer();
t1 = t0 + e2;

// init sample counts
n0 = 0;
n1 = nper;
```

The following code is executed in all following iterations:

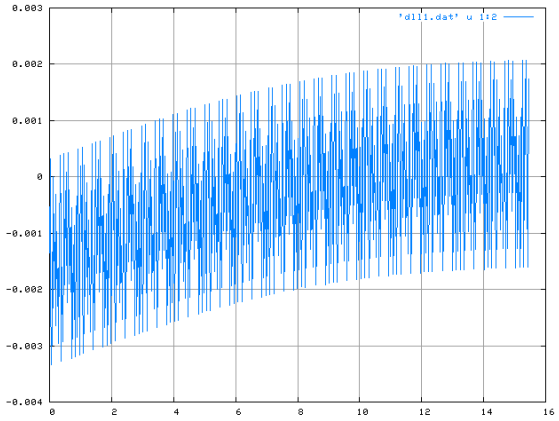

Figure 5: Jitter with USB audio card

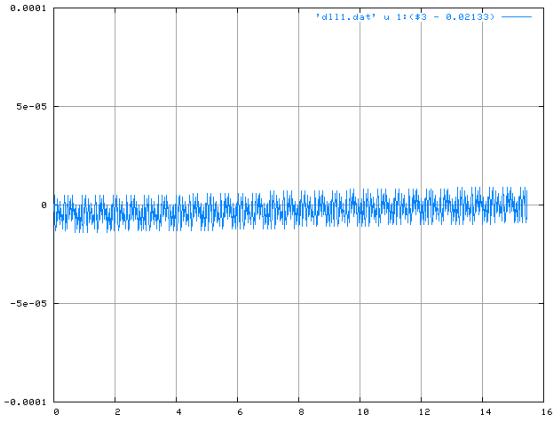

Figure 6: Remaining jitter with DLL filter

```
// read timer and calculate loop error
e = read_timer() - t1;

// update loop
t0 = t1;
t1 += b * e + e2;
e2 += c * e;

// update sample counts
n0 = n1;
n1 += nper;
```

5 Some measured results

The DLL described in the previous sections can easily reduce the system time jitter by a factor of 100. This is in particular important for USB cards, and may also provide a solution to the problem of finding accurate system time to sample count mapping for networked audio.

While for most (PCI based) audio cards the jitter is mainly determined by scheduling delays, USB audio interfaces show and additional problem: the period timing jitter is mainly the result of ALSA's repackaging of the samples into periods of the requested size. In theory this jitter should be in a range of 1ms (the USB interrupt period), but in practice, variations in a range of up to 4 ms are observed[4]. As an example, fig.5, shows the loop error for the author's USB interface. This also shows the loop adapting to the mean interrupt to timer read delay, wich is quite high in this case.

Figure 6 shows the remaining jitter after the DLL filtering. This is reduced from the original $\pm$ 2 ms to a range of about $\pm$ 10 μs. Most PCI sound cards have significantly less jitter to start with, and the filtered result will then be better than one microsecond.

[4]This probably indicates a problem with the ALSA implementation.

6 Acknowledgements

This paper is the synthesis of a very long discussion (by e-mail) of the author with Florian Schmidt and Paul Davis. Many thanks to both of them for their patience ! Florian Schmidt wrote the code that is now part of JACK.

References

Roland E. Best. 1984. *Phase-Locked Loops - Theory, Design and Applications*. McGraw-Hill, New York.

Floyd M. Gardner. 1966. *Phaselock Techniques*. John Wiley and Sons, New York.