PROGRAMMABLE CLOCK GENERATION AND SYNCHRONIZATION FOR USB AUDIO SYSTEMS

KENDALL CASTOR-PERRY

Cypress Semiconductor, San Diego, CA, USA

A USB audio link transports packets of sample data, but provides no associated sample clock. To reproduce samples at the right points in time, the two ends of the link must synchronize their sample rates without exchanging clocks. The paper reviews data transfer modes used in USB audio. In many of these modes an accurate master clock, on whose 'cleanliness' the audio SNR depends, must be generated at the receive end. The flexibility of recent programmable systems-on-chip (PSoC) delivers an effective clocking scheme usable for all modes and sample rates. The frequency synthesizer at its core is described in detail.

INTRODUCTION

USB (the Universal Serial Bus) is an example of a 'hidden clock' data transfer scheme. The clock isn't present on the interconnecting link, but must be inferred and reconstructed from the time behaviour of the link's information content.

The latest tablets, handhelds and media players are built on sophisticated hardware platforms, running new operating systems that are increasingly standardizing on USB as the wired link of choice for a wide range of accessories and enhancements. Some of these systems have combinations of requirements that aren't met by existing USB audio chipsets.

Following an overview of the various schemes used to move audio across a USB link, the paper describes an architecture for both the synchronization and the synthesis of high frequency clocks from information in a hidden-clock data transfer scheme, suitable for implementation on a modern Programmable System on Chip. The work was originally carried out to support the generation of multiple standard audio master clock frequencies from a local crystal clock, and to facilitate their exact synchronization to the 1ms USB packet interval coming from a mobile audio host. It has since found application in a wider range of communications systems.

The use of the techniques described in this paper to create a high frequency master clock synchronized to repetitive low frequency signal is the subject of a Cypress patent application.

USB AUDIO MODES

A USB audio link connects a master unit, the 'host', with a slave unit, the 'device'. Audio can be moved in both directions across this link. By the way, USB terminology means that one must be very careful when using the word 'device'. In this paper, it's always used in the specific sense of this slave form of USB interface.

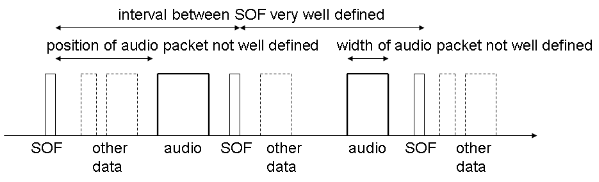
interval between SOF very well defined
position of audio packet not well defined
width of audio packet not well defined
SOF
other data
audio
SOF
other data
audio
SOF
other data

figure 1: audio packets on the USB bus

The audio-related traffic is in the form of packets of data, associated with a particular 'endpoint' of the USB link. There may also be other traffic on the link as well. Both the width and the location in time of these packets can vary, due to the dynamic load on the host's processor and USB controller. This is shown diagrammatically in figure 1. The one constant and reliable characteristic of the USB link is the timing of the start-of-frame (SOF) packets. In a full-speed USB system, these packets appear on the link every millisecond, and both the frequency tolerance and the maximum positional jitter are tightly specified in the USB standards documents.

In a correctly functioning link, there's no question that each end will be able to recover the data in any packet that arrives from the other end. In a bulk transfer application, that's enough. In audio replay applications, however, it's critically important that audio samples are effectively reproduced at *exactly* the correct point in time. If they aren't, the resulting distortion can cause perceptible, or even serious, deterioration of sound quality.

When playing back data from storage, there is of course no *actual* sample clock at the host, because there's no ADC. It's implied by the file format, and the host calculates the rate to send sample data out onto the interface from its local timing reference.

The two ends of the link each have their own local clock oscillator, and these oscillators define the timebases with which data is sent to, and taken from, the physical layer of the link. The USB standard limits the maximum relative frequency tolerance of these oscillators. Although the two ends of the link might agree (in a separate transaction) that the audio data has a nominal sample rate of exactly 48000 per second, the difference between their timebases mean that they will disagree on the *duration* of 'one second'.

Left unresolved, this disparity would result in the replay electronics trying to output either too many or too few samples in a given time period. This would be completely unacceptable.

TO FEED OR NOT TO FEED, BACK?

To ensure that the average rate of audio sample output at one end of the link equals the average rate of sample input at the other, the ends of the link can adopt one of two strategies: feedback, or no feedback. In feedback modes, the link ends communicate information that throttles the inbound sample rate to match what the replay rate is forced to be by the audio hardware.

Alternatively, no feedback is employed, the receiving device just takes what it's given, and fine-tunes its timebase to ensure that the audio replay rate matches exactly the rate that the data is being placed on the link by the host.

Technically, another no-feedback option is to reduce the replay device's timebase frequency far enough that you can guarantee to output samples more slowly than you receive them. Excess samples are stored in a buffer that grows, in principle, to an unlimited length. This technique of 'under-clocking' the audio into a very long memory buffer is unusable in any system where video images (or even PowerPoint slides!) need to be time-aligned to the audio.

This paper concentrates on a clock generation scheme to support the no-feedback case. This is a more general, more 'compatible' and more economical approach, highly suited to the target market of consumer audio accessories. A few comments on the feedback cases are in order, though, because they are growing in popularity at the higher end of the market.

If the replay system is a USB host, it gets to define the replay sample rate, and it ensures that it gets the right number of samples by telling the USB device providing the audio samples to send fewer or more samples in the data packets. This is quite an unusual configuration in modern audio systems, since the device providing the samples will often want to control other bus transactions itself. Nearly always, the replay hardware is located in a USB device. In that case, one of two feedback modes is employed: explicit, or implicit.

In explicit feedback mode, the replaying device maintains a separate endpoint with information that enables the host to shape the traffic so that the rates match (usually through the host adjusting its data packets, though in principle the host could trim its own timebase). This works well provided that you have a spare endpoint available in your USB hardware; this isn't always possible because they get used up supporting other functions such as MIDI, volume control buttons, custom protocols and so on.

An elegant scheme is used to bypass the need for an extra endpoint in implicit feedback mode. Here, it's assumed that the host and the device will be exchanging audio in both directions. The audio hardware in the device uses the local timebase to create audio samples (possibly null samples, if no actual audio is needed) to uplink to the host, but it "keeps an eye on" the interval between the SOF packets, which is a proxy for the host's own timebase.

It shapes the uplink packets to ensure that it is sending data to the host at the correct rate.

Here's the neat bit: the host then *echoes* the packet format for the replay sample data being sent back down to the device. These packets have the correct format, by construction, to ensure that the replay samples will exactly match the available replay sample rate. Because the clocks at the two ends of the link are never synchronized, this approach is called 'asynchronous mode with implicit feedback'. It is becoming popular for high end audio DACs.

DELTA-SIGMA BLUES

In most USB audio replay systems, it's up to the replaying device to adjust its timebase so that audio samples are reproduced at exactly the correct long term rate. In a typical system, though, there is an additional requirement on the local timebase clock. Not only must it have exactly the right long term frequency value, but it must also be very 'clean', i.e. free from jitter. The delta-sigma D-to-A converters (and digital amplifiers) used in modern audio systems need a high frequency master clock; low clock jitter levels are extra critical to achieving acceptable audio replay performance.

An hour of CD-standard music contains 158.76 million samples, requiring 40.64256 *billion* master clock cycles, all of which contribute to the output signal. This high frequency master clock is usually 256x or 384x the audio sample rate, and it is this clock that influences the audio SNR through the clockwork of the delta-sigma DAC. Jitter on the master clock causes out-of-band noise from the delta-sigma modulator to mix back down into the audio band, degrading SNR. If the jitter has dominant frequency components, so will the resulting audible signal, and this can be particularly objectionable.

SYNCHRONIZE AND ADAPT

The signal on the USB interface contains two forms of information that we might use to synchronize the replay timebase to conditions in the host: the data, and the link framing structure. We don't know exactly where each data packet will occur in time, or how many samples it will contain, but we do know the nominal rate at which the samples should be arriving. So the receiving device can keep a count of the incoming samples, and can adjust its local timebase so that it is outputting samples at exactly the same rate in the long term. This is called 'adaptive' mode operation.

The other approach is to examine the link framing structure, specifically the time interval between successive SOF packets. This interval is supposed to be *very* close to 1 ms, and it gives the receiving device an insight into the transmitting host's *definition* of a millisecond, which is in turn derived from the host's own timebase. This is called 'synchronous' mode.

Early USB replay interfaces used synchronous mode but acquired a reputation for poor quality of the recovered clock (and resultant poor replay quality). This was primarily due to deficiencies of clocking implementation rather than inherent shortcomings of the approach. Most dedicated USB replay ICs on the market now use adaptive mode operation, though.

Audio samples arrive from the host at the replaying device in a rather irregular fashion, especially on a link that's carrying other traffic and is being produced by a modestly-powered host. The challenge of adaptive mode is the extraction of a stable long-term estimate of the sample rate that is not contaminated by this irregularity. One very significant source of irregularity occurs when the most popular sample rate of 44.1ksps is used. Because this rate isn't a multiple of the 1kHz USB frame rate, the sample packet length has a systematic modulation on it, with nine packets of 44 samples per channel followed by one of 45 samples. The resulting 100Hz modulation is a detectable consequence in many clock recovery products on the market.

SYNCHRONOUS = SIMPLE

The goal of this work was to create a robust, good-quality device-mode USB replay system that would fit in a specific IC, the Cypress PSoC3. PSoC3's USB interface provides an accessible hardware trigger signal whenever a SOF packet is received, but doesn't provide hardware decoding of the position or meaning of the data traffic on the bus. This made synchronous mode the obvious choice for the design.

In essence, then, the audio master clock generation problem in synchronous mode reduces to a simply-stated goal: multiply up the 1kHz rate of SOF packet arrivals by the appropriate number that results in a clean audio master clock, equal to a specific multiple of the exact sample rate needed to replay the received audio samples.

To ensure a faster clock was available for the CPU and other hardware, it was decided to create a system clock of 1024Fs, and divide that down to produce the 256Fs audio master clock accepted by all the audio converters we considered.

In addition, the system needed nominal-value master clocks to be available even when there was no USB synchronization available. This permits stand-alone operation at both sample rates from a single crystal system clock, for instance when a mobile handset is removed from an accessory dock. The processor also needed to support many housekeeping tasks in the finished product in which this audio subsystem would be embedded. To summarize, a design was sought that was able to:

- Implement clean output system and audio master clocks for both sample rate trees from a single local crystal clock;
- Provide a competitive level of output jitter for use with switched-capacitor delta-sigma DACs, well under 1ns p-p with no severe sidebands;
- Robustly synchronize the output clock to a repetitive input pulse at a frequency ~50000x lower, without an integer relationship;
- Free-run at any of the exact nominal frequencies in the absence of a synchronization pulse train, and synchronize to the USB bus even when no audio traffic is present;
- Stabilize to the correct output frequency within a few audio sample periods (<100us);
- Be insensitive to the variable width and position of audio data bursts within the USB frame, on a highly loaded USB link from a modestly powered host;
- Provide extra endpoints to support MIDI, HID and custom protocols for accessory support on mobile players;
- Fit on a PSoC3.

NOT JUST A PLL

Having settled on synchronous mode to ensure the correct output sample rate, we need an adjustable clock generation method that can deliver the jitter and frequency resolution needed.

The simplistic approach would be to build a PLL-based frequency multiplier, directly converting the 1kHz SOF frequency to our 1024Fs system clock, but this would be entirely inadequate on many counts. The phase noise – and hence jitter – from the VCO output would be horrendous due to the multiplication factor of ~50000, and the settling time would be excessive due to the very low bandwidth required to eliminate ripple from a 1kHz reference frequency.

Worse than that, one of the system clocks required (45.1584MHz, which is 1024*44.1kHz) is not even an integer multiple of the 1kHz USB frame rate. This rules out integer-N PLL techniques. A two-stage PLL cascade is sometimes used to solve this problem, but even then, phase noise and settling time to a sample rate change are inadequate for this application. Specialized ICs using adaptation of loop time constants have found use in professional applications where the sample rate doesn't often change and where slow settling time to final frequency accuracy is acceptable.

FAST <u>AND</u> ACCURATE

Closed-loop control methods ensure frequency accuracy by adjusting an oscillator's output frequency so that it doesn't change over time relative to some timing reference. An FLL (frequency-locked loop) stabilizes the frequency over time but leaves an arbitrary phase relationship. A conventional PLL stops the *phase* from changing with time, and this has the knock-on effect of keeping the frequency locked as well. These loop methods rely on comparing some characteristic of the output waveform to that of an input waveform. This is feedback, and it has its attendant loop dynamics.

It's also possible to frequency-lock using *feedforward* methods. A slow input reference, such as our 1kHz SOF interval clock, can be used to gate a counter clocked by some convenient local high frequency clock. The output of the counter tells you, to some given resolution, what number you needed to divide the local clock by to get the SOF interval clock. This information is all you need in order to calculate *another* number, by which you need to divide the local clock in order to get the actual high frequency clock you require. It can be constructed *exactly*, and there's no need to compare it with anything, so there's no loop and far fewer dynamics issues. But of course the devil is in the details.

Let's look at a numerical example, with our USB SOF interval clock at ~1kHz and a local clock at ~24MHz.

Our counter will register around 24000 at the end of each SOF period. Let's say that the counter reading is actually 24003. This means that we know that if we divide our local clock by 24003, we'll produce the frequency that the host considers to be 1kHz. We can immediately say that to create a frequency that the host would think was 45.1584MHz (1024*44.1kHz), we would need to divide our local clock by (24003/45158400). Now, whilst it might not yet be obvious *how* we would do that, we nevertheless know that if we *can* do it, we'll get *exactly* the right output frequency.

ONE STEP FORWARD, ONE STEP BACK

The chosen implementation on the target PSoC3 IC was a two-stage approach exploiting the best characteristics of feedback and feedforward approaches. In part, this was driven by the particular blocks of hardware available on the chip.

The wanted 1024Fs clock is generated from a good quality PLL fitted as standard to the PSoC3. This can take a reference clock frequency of between 1MHz and 3MHz, and multiply it up by a programmable rational fraction to produce a clean output clock at up to 67MHz. The loop bandwidth of this PLL is around 50kHz and its response to changes in the input reference frequency is more than fast enough for the application.

The reference clock for the PLL is generated by a fast feedforward first stage, which calculates the fractional factor by which the local crystal source must be divided to generate the exact required reference clock, and then performs this division implicitly. This is done with a dual modulus prescaler whose divide control input is driven from a delta-sigma modulator fed by an input representing the fractional divisor. The noise-shaping loop provides a two-level output that represents, over time, the fractional component of the number with which we need to divide the local clock in order to get the reference frequency needed. The fractional part is designed to be as close to 0.5 as feasible given other constraints, to maximize the 'pull range' of the synchronization.

The modulus of the prescaler is continually being switched between two values by the output of the noise-shaper; this process is equivalent to FSK modulation. It creates a main fundamental component whose frequency is equal to the input clock divided by the long-term average of the fractional modulus. In addition, modulation sidebands are present. The job of the second stage PLL is to multiply the reference clock by a suitable small rational number, while filtering out the higher frequency harmonics and sidebands. The result should be a clean, accurate clock. Low-offset sidebands from the prescaler output will get through the PLL; ideally, we would have liked a rather lower loop bandwidth, but this wasn't available from the on-chip PLL.

The synthesizer has an option to support adaptive mode for those situations where the data is not actually arriving at a rate that can be correctly inferred from the SOF timing. This can happen in some embedded systems where audio data synchronized to a remote source is being relayed across the USB interface. To fix this, a separate input into the noise shaper allows adjustment of the free-run frequency in steps of ~0.8ppm in the standard configuration. This could also be used under processor control to implement the standard type of adaptive mode operation, but the extra CPU overhead and development time was not considered worth it for our project.

Sidebands in the clock modulate the reproduced audio in the system, degrading fidelity. The current design uses a first-order modulator, but it is straightforward to increase the order further. As might be expected from a first-order modulator, spurious sidebands are present at a frequency that depends on the instantaneous counter output value.

The 'pull' input can also be used as an additional 'dither' path into the system. Simulations show that dither can be effective at further reducing residual tonal components that can occur at particular frequency offsets in a first-order modulator system. No practical work has yet been done on this because the measured and auditioned system performance has been found satisfactory without it so far.

OVERALL SYSTEM ARCHITECTURE

The local clock for the system is derived from a 24MHz crystal that is also used to time the USB data recovery process when audio playback is enabled. When audio is idle, the interface clock switches over to a lower power RC oscillator.

The first stage is written in Verilog and implemented in programmable logic in the PSoC3. It is designed so that if there isn't any synchronization information available from the USB SOF packets, it free-runs at the exact nominal sample rate programmed. The parameters for this block set the frequency relationships in the system, and are detailed in the appendix.

The second stage uses the standard PSoC3 PLL; it produces the clock that the CPU and most of the digital hardware runs on. It wasn't specifically designed for audio applications, but the chip is fully characterized to run off the clock that it produces. The overall jitter of the generated audio master clock is around 600ps pk-pk at either common sample rate, and most of this is random phase noise from the VCO rather than tonal components from the synthesis.

A 768 byte USB data buffer provides room for a maximum of 4 48 sample pair packets. At 48ksps 16bit operation it typically runs half-full, and somewhat less for 44.1ksps operation. The replay latency through the USB interface is therefore of the order of 2ms, which is about as low as it is possible to go safely.

The audio data is clocked out of the buffer into a standard I^2S interface, implemented with the programmable digital blocks. This interface can connect to a standard audio DAC, processor or 'digital amplifier'. S/PDIF retransmission of the data is also supported, again implemented in the programmable digital blocks.

While S/PDIF receive wasn't implemented in this work, it's believed that the same synthesis process can be used to recover a suitable audio master clock from incoming coaxial or optical S/PDIF transmissions. In that case, the frame boundaries are marked by intentional Manchester code violations, and these would be used to synchronize the synthesizer.

The PSoC3 chosen for implementation has many additional capabilities that are attractive for mobile accessory development. A Digital Filter Block enables the creation of extensive audio filtering and other audio effects. This can post-process the recovered USB audio, for instance for response equalization and crossover filtering. Sufficient performance is available to render additional digital processing ICs redundant; at least ten second order biquad filters can be implemented on each channel of a stereo pair, giving very fine control over frequency response.

Other configurable analogue resources on the PsoC3 allow programmable current limiting, the support for battery charging strategies, capacitative button sensing, biological signal detection and many other useful circuits.

SYNCHRONOUS PLUS ASYNCHRONOUS

It has already been mentioned that asynchronous mode is becoming popular at the 'high end'. A disadvantage of asynchronous mode in low-cost products is the requirement for three different stable clocks, one for each of the standard sample rate clock trees and one for the processing hardware itself.

High-end purists might baulk at the idea that a synthesizer be used to create the audio clock in an asynchronous system. However, a compromise implementation can deliver some of the benefits of both modes. The delta-sigma synthesizer is versatile enough that it can support generation of one master clock rate from another. If we use say a 24.576MHz local crystal, for instance, we can generate audio clocks for the 48ksps family of rates directly from the crystal, and employ asynchronous mode to ensure the highest possible quality audio clocking. This might be used for 'prosumer grade' digital audio recording.

At the same time, this local clock can also be used to synthesize the required clocks for 44.1kHz operation, allowing still-good playback performance of 44.1kHz material in asynchronous mode in the absence of a high quality sample rate converter. A simple configuration change will allow the device to operate in synchronous mode for either sample rate with no change in hardware, ensuring compatibility with systems that do not support asynchronous mode operation. This is more versatile than any other USB audio interface currently on the market.

FURTHER COMMENTS

Some applications use different high frequency crystal-derived clocks. In Ethernet systems, 25MHz is a common clock; in GSM-based systems 26MHz is ubiquitous, and in systems with standard resolution video outputs, 27MHz is the standard. Exact-frequency audio master clock generation with USB synchronization is possible from any of these frequencies with the technique described here.

When the synchronization signal is present, the accuracy of the local oscillator does not affect output frequency accuracy. This means that a crystal-based local oscillator is not mandatory purely on frequency accuracy grounds if the synchronization signal is always present. The PsoC3 Internal Main Oscillator (IMO) has also been tried as the local oscillator. As long as the frequency error of the IMO is within the capture range of the set configuration, the long term frequency will be accurate. The short-term jitter performance will be poorer, since this is set by the jitter of the IMO, which is an RC oscillator. However, for some lower-end applications, ultimate jitter performance is not important, only long term frequency stability.

CONCLUSIONS

The two-stage clock synthesis process combines a fast-responding feedforward front-end and a classical, well-understood feedback PLL back-end. Arithmetic manipulation of the various clock frequencies leads immediately to the necessary small set of design parameters that optimize the bit depth required by the noise-shaped modulator driving the prescaler, and the frequency range over which the system will synchronize.

The resulting highly designable block can effectively multiply a low-frequency synchronizing event by a large, non-integer factor to create a stable, low-jitter clock suitable for driving audio converters and other high dynamic range mixed signal circuit blocks. It can be implemented in any PSoC3 (or PSoC5).

The implementation has been widely used during development of Cypress's USB audio reference design, on PC, Mac and mobile USB hosts, and performs extremely well. The author is using a version in his domestic audio system and finds the sound quality to be excellent.

Many Cypress colleagues were involved in implementing and testing portions of this design; Special thanks go to Brad Budlong and Isaac Sever.

APPENDIX: THE SYNTHESIZER IN DETAIL

A noise-shaped dual modulus prescaler is one whose divide-select input (which determines whether it divides by L or L+1) is driven from a noise shaper [see Bourdopoulos et al; 'Delta-Sigma Modulators: Modeling, Design and Applications'; Imperial College Press]. The shaper creates a two-state output whose duty cycle encodes a higher resolution input word that represents the fractional part of the actual factor with which the input clock needs to be divided. The basic idea is shown in figure 2.

The core of a noise shaper is a transfer function block whose coefficients are calculated to give a lowpass response to the input signal and a highpass response to the inevitable quantization noise caused by the restricted number of possible output states.

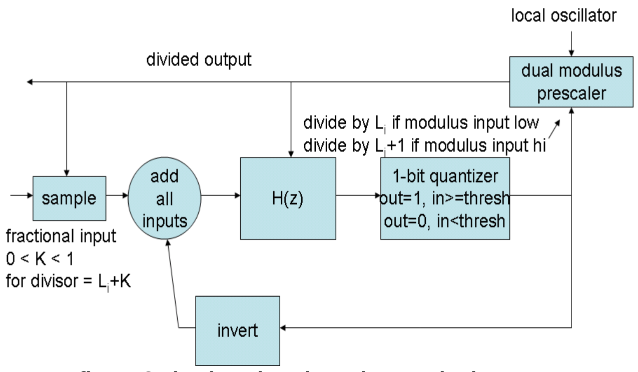
local oscillator
divided output
dual modulus prescaler
divide by L_i if modulus input low
divide by L_i+1 if modulus input hi
sample
add all inputs
H(z)
1-bit quantizer out=1, in>=thresh out=0, in<thresh
fractional input 0 < K < 1 for divisor = L_i+K
invert

figure 2: basic noise-shaped prescaler loop

Commonly used in ADC designs, the same concept can be used in many domains. Here, a noise-shaper is used on an input signal that can have many different values representing a desired fractional division ratio, to produce an output that can have one of only two values, realized by a dual modulus divider.

Simple loop functions are used in this design for ease of implementation. We define the Signal Transfer Function and Noise Transfer Function as usual in terms of the z-domain response of the sampled filters, which are clocked from the output of the filter:

$$out(t) = \frac{H(z)}{1+H(z)} \cdot in(t) + \frac{1}{1+H(z)} \cdot e(t) \qquad (1)$$

$$i.e.\ out(t) = STF \cdot in(t) + NTF \cdot e(t)$$

The first and second order implementations of H(z) are shown in figure 3. For the first application of this system, the simple first order solution was used, to economize on digital block usage in the PSoC3.

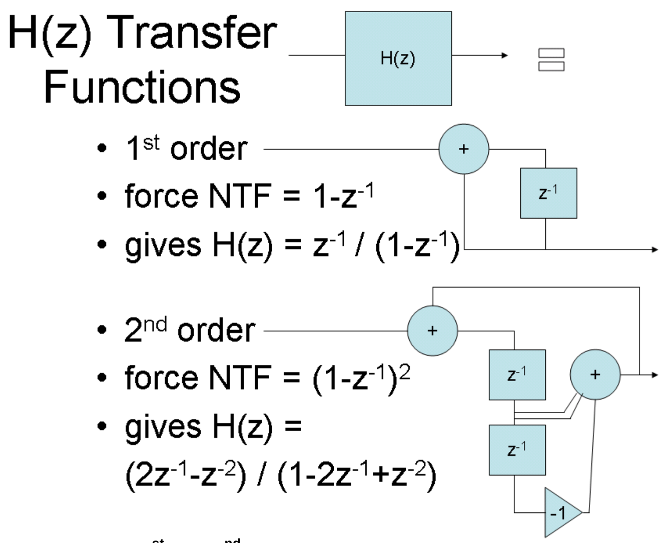

figure 3: how 1st and 2nd order functions are implemented

CALCULATIONS

The division factor is a fraction of potentially arbitrary resolution, so an arithmetical analysis of the clock relationships was done in order to guarantee the bit depth of the implementation and therefore the amount of PSoC3 programmable digital hardware required.

The signals present in the system have the following frequencies:

f_{outi}	the 'ith' desired output clock frequency
$f_{outinom}$	the *nominal value* of f_{outi} – not an actual signal
f_{sync}	the low frequency to which we must sync the output clock
$f_{syncnom}$	the *nominal value* of f_{sync} – not an actual signal
f_{osc}	the high frequency local oscillator
f_{oscnom}	the *nominal value* of f_{osc} – not an actual signal
f_{refi}	the reference frequency input to the PLL when generating the 'ith' desired output clock frequency
$f_{refinom}$	the *nominal value* of f_{refi} – not an actual signal

The relationships between these clocks and parameters are:

$f_{outi} = W_i \cdot f_{sync}$ always, i.e. $f_{outinom} = W_i \cdot f_{syncnom}$

where W_i is the desired *exact rational constant* multiplier relating the 'ith' desired output clock to the input synchronization frequency.

$f_{outi} = f_{refi} \cdot P_i/Q_i$ always, i.e. $f_{outinom} = f_{refinom} \cdot P_i/Q_i$

where P_i and Q_i are the *integer constant* values of the PLL's feedback and reference divider ratios when generating the 'ith' desired output clock. The *choice* of P and Q values will be limited by the available hardware (on PSoC3, $8 \le P \le 255$ and $1 \le Q \le 16$) and the *ratio* of these values is constrained by the valid reference frequency range of the PLL (on PSoC3, $1\text{MHz} \le f_{ref} \le 3\text{MHz}$)

$C = f_{osc} / f_{sync}$
with $C_{nom} = f_{oscnom} / f_{syncnom}$ and $\Delta C = C - C_{nom}$

where C is the *current measured ratio* between the local oscillator and the incoming synchronization frequency; it is the integer output of a counter and varies over time. C_{nom} is the *integer constant* value that *would be measured* with nominal values; it doesn't depend on i.

$L_i \le f_{osc} / f_{refi} \le L_i + 1$

where L_i is the modulus parameter (i.e. the prescaler can divide by L_i or $L_i + 1$ depending on whether the modulus input is low or high). This constraint says that the mean ratio between the input and output of the fractional divider is a number between L_i and $L_i + 1$. The L_i values may be the same or different for the various output frequencies required.

Empirically, we might expect that lower values of L might cause more of a 'shock' to the PLL, as the input frequency will be jumping back and forth with a larger frequency ratio. For a given set of PLL loop dynamics, this will lengthen the time during which significant activity will occur in the PLL charge pump. This is an additional source of phase noise and reference modulation.

The noise-shaped prescaler acts as a frequency divider, with an input frequency of f_{osc} and output frequency of f_{refi}. The division ratio required in the nominal case is therefore:

$$\textit{division ratio} = \frac{f_{osc}}{f_{refi}} = \frac{f_{osc} \cdot P_i}{f_{sync} \cdot Q_i \cdot W_i} = C_{nom} \frac{P_i}{Q_i \cdot W_i} \qquad (2)$$

This is a number between L_i and $L_i + 1$, say $L_i + K_{inom}$. So the fractional part K_{inom} is

$$K_{inom} = C_{nom} \cdot \frac{P_i}{Q_i \cdot W_i} - L_i \qquad (3)$$

and we try to set up the parameters so that K_{inom} has a value of around 0.5. We can simplify this, by dividing out the greatest common divisor of the fractional part (remembering of course that W_i is rational, so the numerator and denominator must be explicitly included in the calculation), to get the input signal to the noise shaper:

$$K_{inom} = C_{nom} \cdot \frac{N_i}{M_i} - L_i \qquad (4)$$

We can't easily perform the division in simple hardware, but we can scale both the input and the feedback in the noise-shaper by a factor of M_i. now, when the output of the integrator exceeds the threshold, M_i is subtracted from the integrator (instead of unity):

$$K'_{inom} = K_{inom} \cdot M_i = \left(C_{nom} \cdot \frac{N_i}{M_i} - L_i \right) \cdot M_i$$
$$= C_{nom} \cdot N_i - L_i \cdot M_i \qquad (5)$$

The base input word we have to add into the noise shaper's input summer, K'_{inom}, is precalculated from (5) and automatically added on every update cycle. When the actual loop is running and the counter value C differs from C_{nom} by an amount ΔC (which might be +ve or ve), we simply have to add an additional input of $N_i\Delta C$ to the integrator. Summing in an additional input value forces the loop to be slightly incorrect in a known way. This path can be used for dither (mean value of zero) to break up idle tones, and for a 'pull' input when a specific small extra frequency shift is needed.

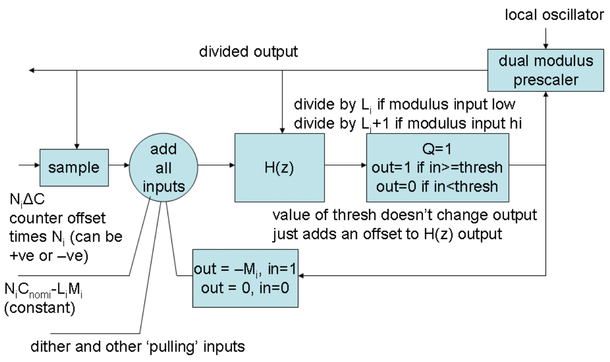
local oscillator
divided output
dual modulus prescaler
divide by L_i if modulus input low
divide by L_i+1 if modulus input hi
sample
add all inputs
H(z)
Q=1
out=1 if in>=thresh
out=0 if in<thresh
$N_i\Delta C$ counter offset times N_i (can be +ve or –ve)
value of thresh doesn't change output just adds an offset to H(z) output
out = $-M_i$, in=1
out = 0, in=0
N_iC_{nomi}-L_iM_i (constant)
dither and other 'pulling' inputs

figure 4: final scaled noise shaper loop

The operation of the final delta-sigma loop as shown in figure 4 is straightforward. On every transition made at the output of the prescaler, $N_i\Delta C$ is added to the integrator, along with either K'_{inom} or K'_{inom}–M_i depending on whether the current output of the accumulator is below the preset threshold. All we now need to do is determine the size of the registers needed to carry out the arithmetic, and the value of the decision threshold.

Because K_{inom} is a fraction less than one, we know that K'_{inom}<M_i and therefore M_i is the largest amount by which the integrator output can move in a single cycle when ΔC=0. The number of bits in the integrator's register must therefore be capable of representing M_i exactly. Also, the range 'left over' between M_i and the size of the register should be distributed evenly between upper and lower limits to permit equal values of ΔC in both directions before saturation is reached.

We therefore need a number of bits B=ceil($\log_2(M_i)$). In the PSoC3 UDB implementations, it's convenient to allow B to be a multiple of 8. The threshold T_i for the output decision is

$$T_i = 2^{B-1} - K'_{inom} + ceil\left(\frac{M_i}{2}\right) \qquad (6)$$

and the maximum possible capture range X_i limited by this mechanism, $\Delta C_{maxi}/C_{nom}$, turns out to be:

$$X_{\Delta Ci} = \frac{\Delta C_{\max i}}{C_{nomi}} = \frac{1}{C_{nomi}} \mathrm{int}\left(\frac{2^{B-1} - \mathrm{int}\left(\frac{M_i}{2}\right)}{N_i} \right) \qquad (7)$$

To maximize capture range, a set of parameters is selected that delivers the smallest value of M_i while meeting all other constraints. The ultimate limit for capture range is due to the finite division range available from the prescaler. The local oscillator can be divided by a number that's between L_i and L_i +1. This means that the ratio between maximum and minimum frequencies is (L_i +1)/L_i and the capture range limit from this mechanism is

$$X_{Li} = \min\left(\frac{L_i + K_{inom}}{L_i}, \frac{L_i + 1}{L_i + K_{inom}} \right) \qquad (8)$$

IMPLEMENTATION

The parameters in the previous section are shown with subscripts because in typical USB audio systems we must be able to switch between several different multiplication ratios. This block is required to generate system clocks of 49.152MHz (1024*48kHz) or 45.1584MHz (1024*44.1kHz) from the standard 24MHz crystal clock, synchronized to the 1kHz pulses from the 1USB SOF packet detector. In other words, there are two values of W_i, which will require us to define two different sets of L, P and Q. Eliminating the common divisors will result in two N,M pairs that define the operation of the noise shaper.

Initially, it was decided to set the prescaler modulus as high as possible; the limit is set by the minimum allowed reference frequency for the PLL, which is 1MHz. A prescaler with L_i=23 just meets this with the 24MHz local oscillator. This selection forced the reference divider ratio R to be unity. P values of 44 and 48 were chosen for the 44k1 and 48k cases respectively, with nominal division ratios (L_i +K_{inom}) of 23.384 and 23.475 ensuring that the modulator's ones-density level is reasonably high.

Cancelling out common factors resulted in N_i/M_i fractions of 55/56448 and 1/1024 for these two cases. However, it was decided to force the value of N to be the same in both cases, just in case the implementation of widely varying parameters proved difficult. So an N_i/M_i fraction of 55/56320 was used in the 48kHz case. If the values of N_i and M_i are multiplied by the same factor, the operation of the modulator is unchanged, except that the capture range in this case becomes much narrower, eventually becoming limited by equation (7) instead of equation (8).

The design equations can readily be incorporated into a spreadsheet. Use of the Excel function GCD() is made, in order to carry out the factorization. It is possible to automate the selection of optimal parameters by using the Excel 'solver' functionality to search the solution space. When this was tried, the spreadsheet actually found a superior solution to the manually determined first generation set of parameters, with capture range of over ±1.5% instead of ±0.34%

The spreadsheet-based design allows exploration of configurations with higher reference frequency, i.e. lower prescaler modulus. The trade-off is between the wider deviation of the prescaler output, and the better filtering of the reference spurs in the PLL.

RESULTS

Behavioural models of this process were created in both BASIC and SPICE, to study the dynamics of the synthesis process in the time and frequency domains, using the chosen setup parameters for the two output frequencies required in the audio application.

A first-order modulator version of this architecture was implemented in PSoC3. The simulated results for idle tone frequency and level were compared with baseband and output clock measurements, and quantitative agreement was observed on the cases tested. There's no such thing as a 'typical' simulation graph to show, since the performance varies so strongly with instantaneous frequency offset. In the configurations built so far, simulated idle tone offsets, when they occur, are at around -70dBc. They jump around significantly in frequency as the offset changes, so noise and jitter on the interface timing itself actually has a beneficial effect on the level of audible spurious components on the clock. It was found that FFT lengths of between 1M and 16M points were necessary in order to dig out the detail of the synthesized spectrum.

It's expected that a second-order implementation will show a lower level of sidebands in simulation, and that dither will further improve matters. It hasn't yet been found necessary to resort to either of these enhancements.