PROGRAMMABLE CLOCK GENERATION AND SYNCHRONIZATION FOR USB AUDIO SYSTEMS

Kendall Castor-Perry
Cypress Semiconductor
kvcp@cypress.com

Tell ‘em Why (in reverse order)

Why ‘USB Audio Systems’?

- USB is the most widespread consumer digital audio transport.
- It is also being adopted by tablets and mobile media players, encouraging a new ecosystem of audio accessories.

Why ‘Synchronization’?

- The two ends of the link don’t exchange an audio clock, yet need to agree on an *exact* sample rate.
- Otherwise the DAC will continually run out of, or lose, audio samples.

Why ‘Clock Generation’?

- Well, a clean audio master clock has to be produced somehow.
- This isn’t trivial, if that clock frequency could vary over some range.

Why ‘Programmable’?

- There are several different ways of transferring audio across a USB link.
- Products often need other customizable end-points and features.
- Existing ASSPs aren’t flexible enough, and ASIC development is too expensive.

What’s a Millisecond Between Friends?

The host transmits a SOF (Start of Frame) packet every ms.

Or, more precisely:

The host transmits a SOF (Start of Frame) packet **every 12000 counts of its local USB clock** (12MHz ± 0.2%).

This interval is the host’s **definition** of 1 ms.

If the current audio sample rate is Fs, by default the host sends one data packet containing Fs/1000 samples in each frame.

If Fs is not an integer multiple of 1000Hz (e.g. audio at 44.1ksps) a finite, repeated sequence of packets with a mean content of Fs/1000 samples is sent.

For 44.1ksps, 9 frames with 44 sample packet and one frame with 45 sample packet, mean = 44.1 samples per frame.

Device-mode audio replay

We're covering the most challenging case here, where the USB *device* is replaying the audio sent to it by the USB *host*.

If the equipment doing the replaying is also the host on the link, everything is much easier because the host always knows how much data to suck out of the device so that it doesn't get ahead or behind.

This approach is hardly ever used; the host should generally be the most powerful, application-intensive thing in the system, and it might be talking to many different devices.

It crops up in some specialized mobile player replay applications, but it's surprisingly difficult and in some cases actually deprecated by the player vendor.

"We do these things because they are hard..."

Synchronization modes without Feedback

In modes without any feedback, the device is left to its own... devices ☺ All the device has is the data coming from the host, and the timing of the underlying frame structure.

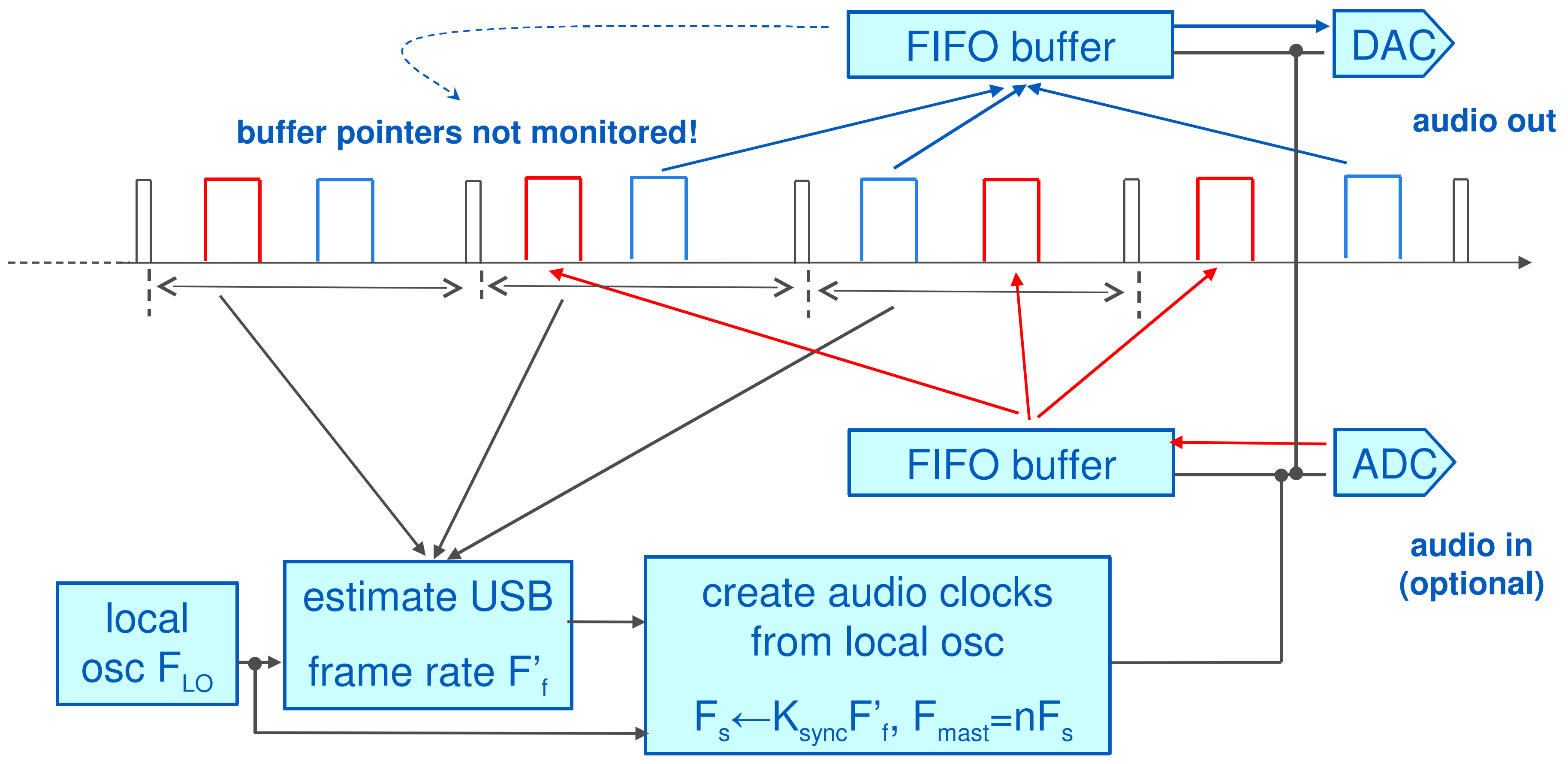
In <u>synchronous</u> mode, the device synchronizes its audio i/o timing with the host by using the detection of the SOF packet as a ‘tick’ that *defines* the host’s millisecond.

In <u>adaptive</u> mode, the device measures the rate of sample arrivals, and adjusts its audio i/o timing so that it matches the known sample rate of the material.

In <u>adaptive synchronous</u> mode, the usual synchronous mode operation is augmented by a fine-tuning loop that detects slippage of the data rate against the host timing.

The host doesn’t need to know which of these philosophies the replaying devices is adopting.

Synchronous Mode

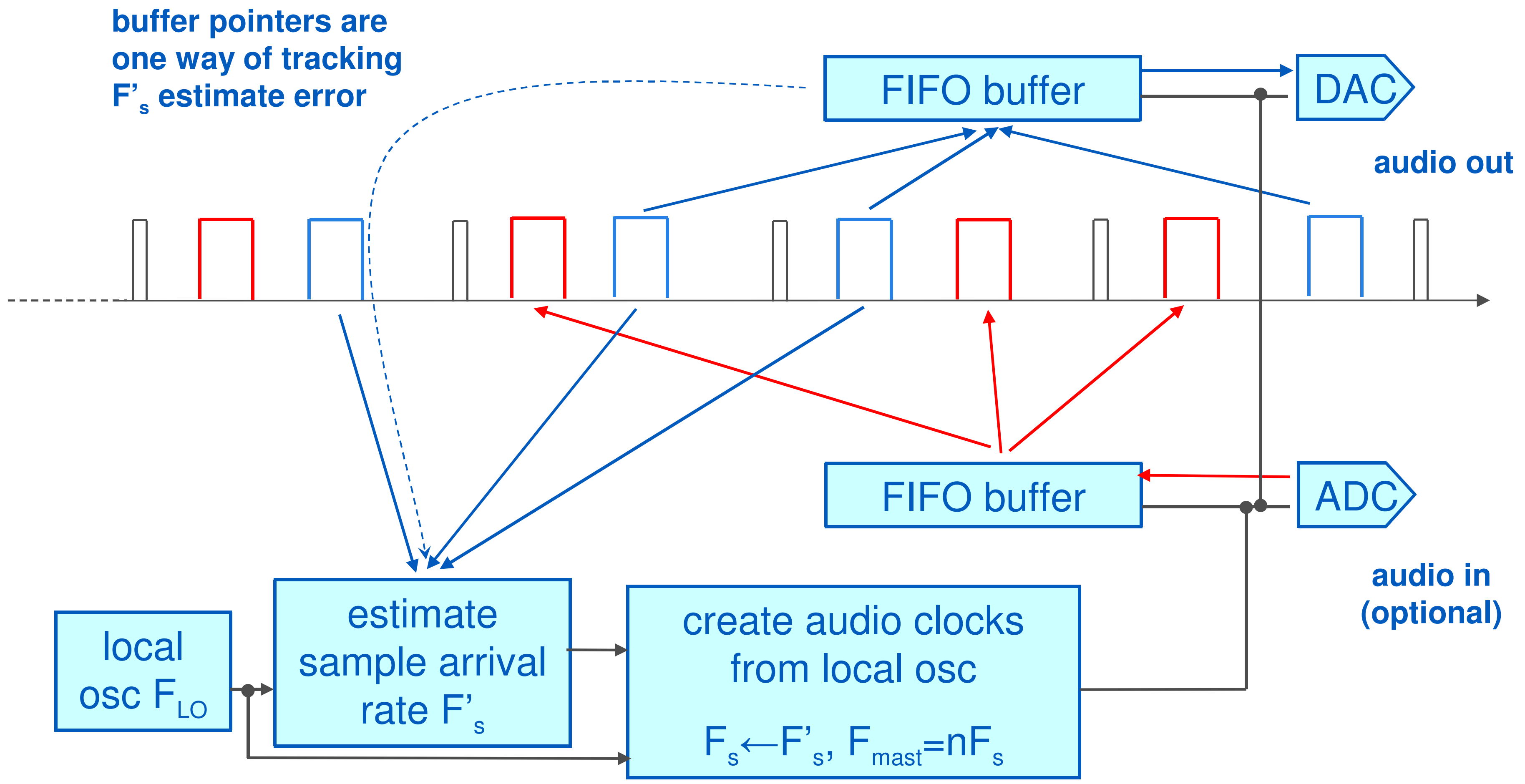


FIFO buffer
DAC
audio out
buffer pointers not monitored!
FIFO buffer
ADC
audio in (optional)
local osc F_{LO}
estimate USB frame rate F'_f
create audio clocks from local osc
$F_s \leftarrow K_{sync}F'_f$, $F_{mast}=nF_s$

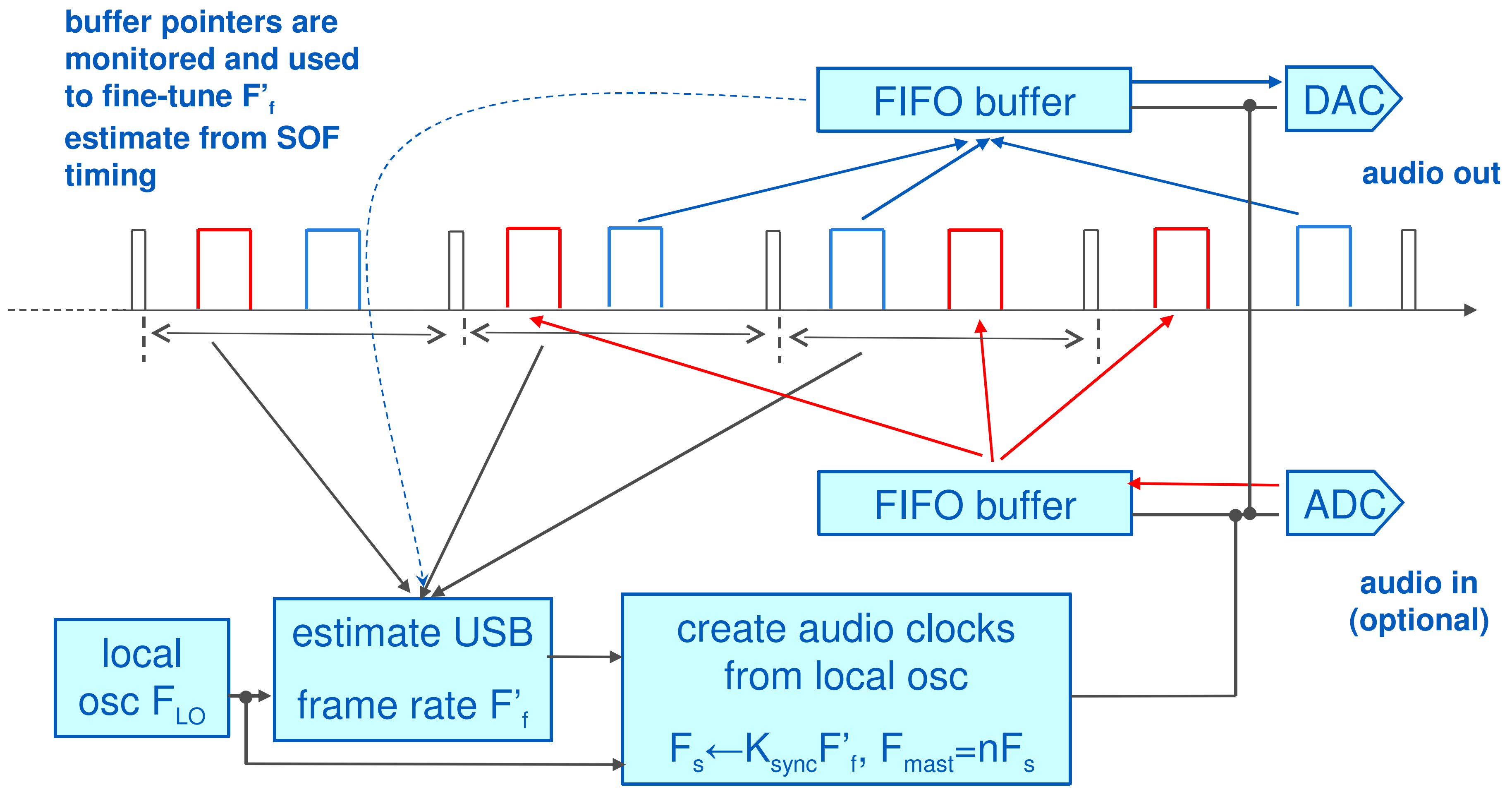
Audio in and out packets can appear in any order in the frame (doesn't usually change dynamically)

Audio in and out sample rates don't need to be the same (but nearly always are)

Adaptive Mode


buffer pointers are one way of tracking F'_s estimate error
FIFO buffer
DAC
audio out
FIFO buffer
ADC
audio in (optional)
local osc F_{LO}
estimate sample arrival rate F'_s
create audio clocks from local osc
$F_s \leftarrow F'_s$, $F_{mast} = nF_s$

Adaptive Synchronous Mode


buffer pointers are monitored and used to fine-tune F'_f estimate from SOF timing
FIFO buffer
DAC
audio out
FIFO buffer
ADC
audio in (optional)
local osc F_{LO}
estimate USB frame rate F'_f
create audio clocks from local osc
$F_s \leftarrow K_{sync} F'_f$, $F_{mast} = nF_s$

Synchronization modes with Feedback

In these modes, indirect timing information is exchanged between device and host over the USB bus. This allows the audio clocks to be independent of the interface clockwork.

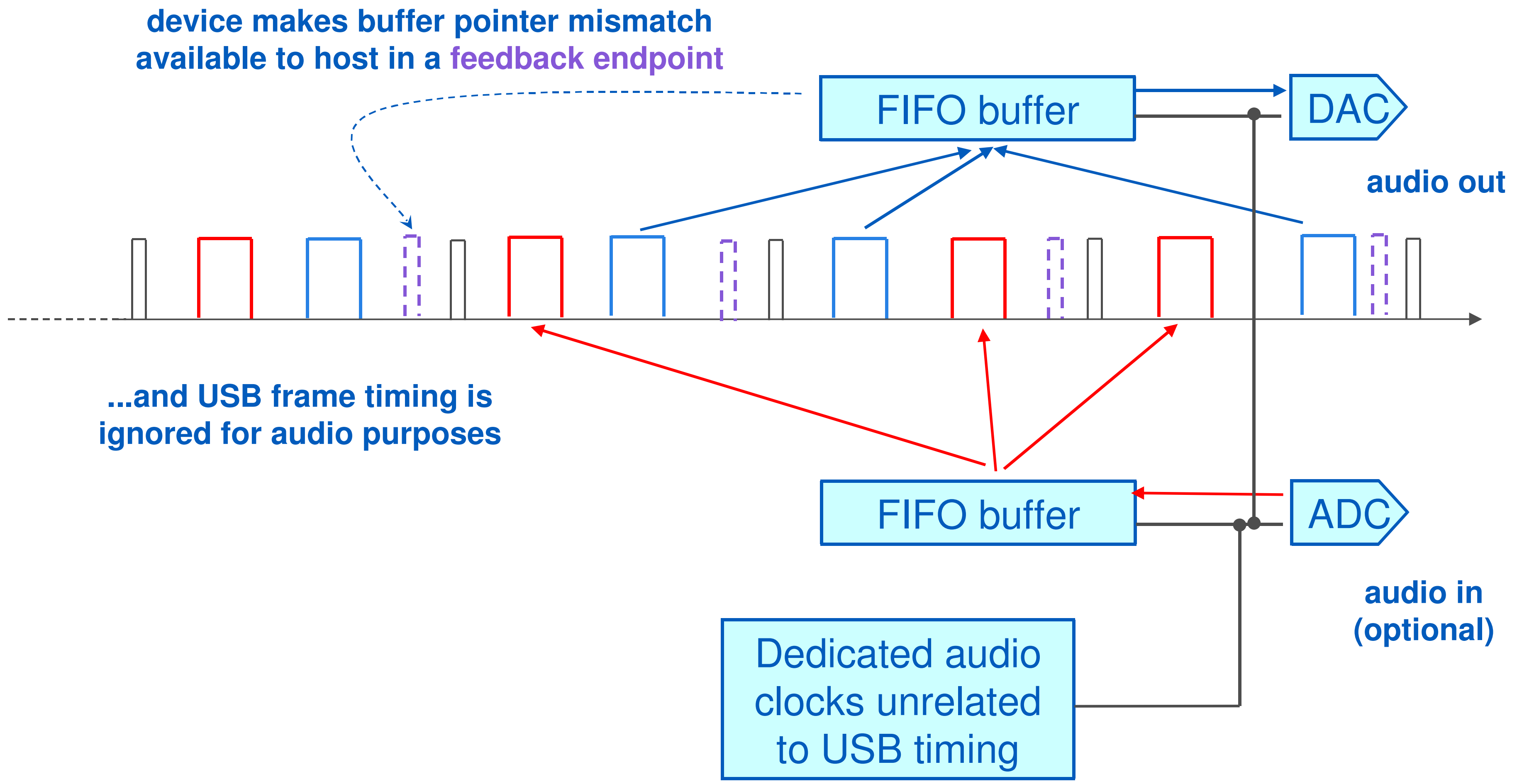
Data flow is managed so it can be reproduced by the device's own local clock, without losing or doubling up on samples.

One way of doing this is for a USB endpoint in the device to carry information about how fast the ends are gaining or losing; <u>asynchronous with explicit feedback</u>. That's a bit inconvenient if you haven't got a spare endpoint, or enough bus bandwidth for the extra traffic.

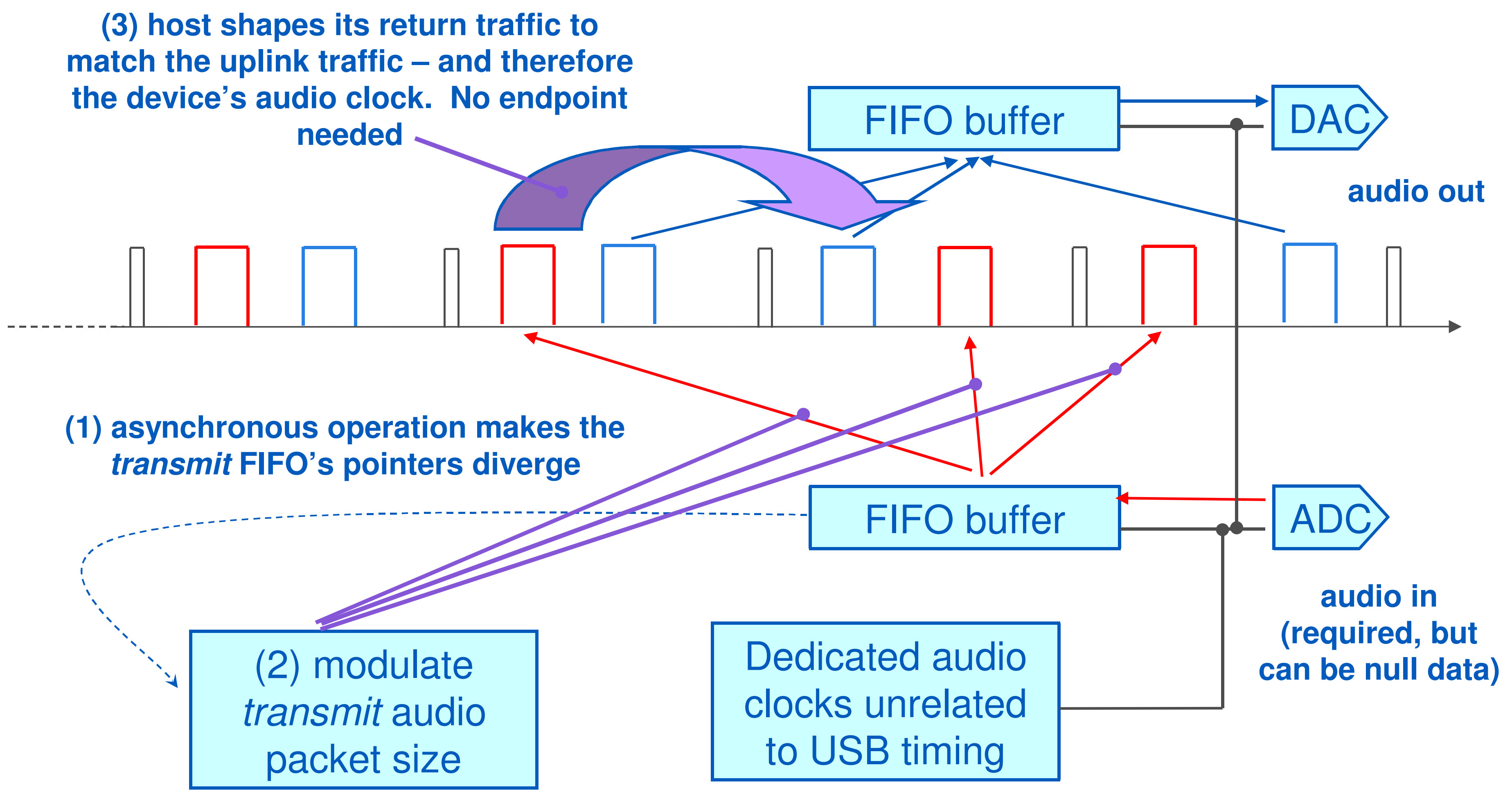
The other way, <u>asynchronous with implicit feedback</u>, is a neat "hiding in plain sight" method that hijacks normal link functionality to provide a hidden feedback path.

The host needs to know which of these schemes to follow. Not all hosts support asynchronous modes.

Asynchronous Explicit Mode


device makes buffer pointer mismatch available to host in a feedback endpoint
FIFO buffer
DAC
audio out
...and USB frame timing is ignored for audio purposes
FIFO buffer
ADC
audio in (optional)
Dedicated audio clocks unrelated to USB timing

Asynchronous Implicit Mode


(3) host shapes its return traffic to match the uplink traffic – and therefore the device's audio clock. No endpoint needed
FIFO buffer
DAC
audio out
(1) asynchronous operation makes the *transmit* FIFO's pointers diverge
FIFO buffer
ADC
audio in (required, but can be null data)
(2) modulate *transmit* audio packet size
Dedicated audio clocks unrelated to USB timing

Jitter, Phase Noise and converter SNR

Everyone looking at this knows that if the audio clocking isn't perfectly 'clean', audio quality will suffer in some way.

The cleanest clocks come from carefully constructed crystal oscillators. These won't be related to the clocking on the interface and so modes with feedback (i.e. asynchronous) must be used for correct transfer between host and device.

Good oscillators are expensive and Asynchronous modes are sometimes unsupported in the host hardware.

Modes without feedback are prevalent in consumer-grade audio. They require you to somehow synthesize a variable frequency signal of arbitrary frequency resolution.

This synthesis process will add some unwanted phase noise and spurious frequencies to the master clock that goes to the converters. Consumer-grade delta-sigma converters can be particularly sensitive to this.

Take *two* Delta-Sigmas into the shower?

The implementation described here addresses the sensitivity of the delta-sigma loop in a consumer DAC with... another delta-sigma loop.

The synchronization mode chosen for this project was Synchronous. It’s unfashionable, largely due to bad experiences in the early days of USB audio.

The fine resolution needed for the recovered audio clocking is achieved with a delta-sigma synthesizer.

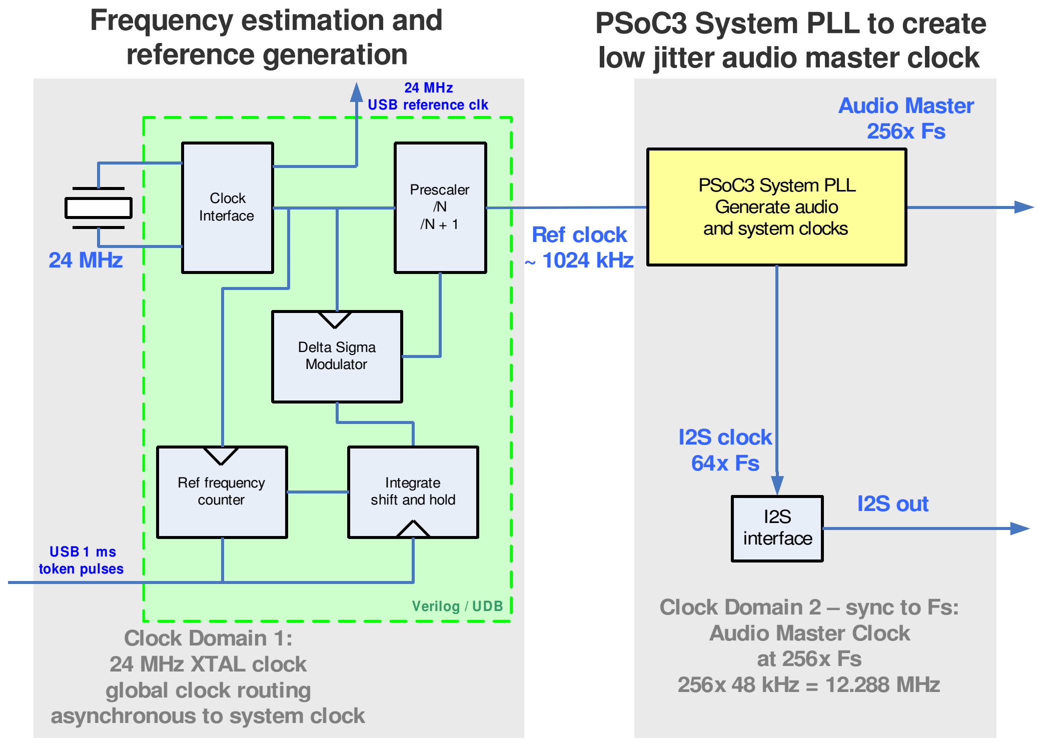
This uses noise-shaping around a frequency synthesis system of moderate resolution to suppress error components, completely analogous to the use of such a loop in an audio DAC.

In an ideal world, all the discrete frequency error is taken away, and all that’s left is noise (phase noise, in this case), and hopefully not much of that.

Don’t panic, it’s all in the text of the paper!

We first divide down our local oscillator by a carefully defined and dynamic constant which has a fractional part.

The delta-sigma loop turns the fractional part into a sequence applied to the divide-control input of an L~L+1 prescaler.

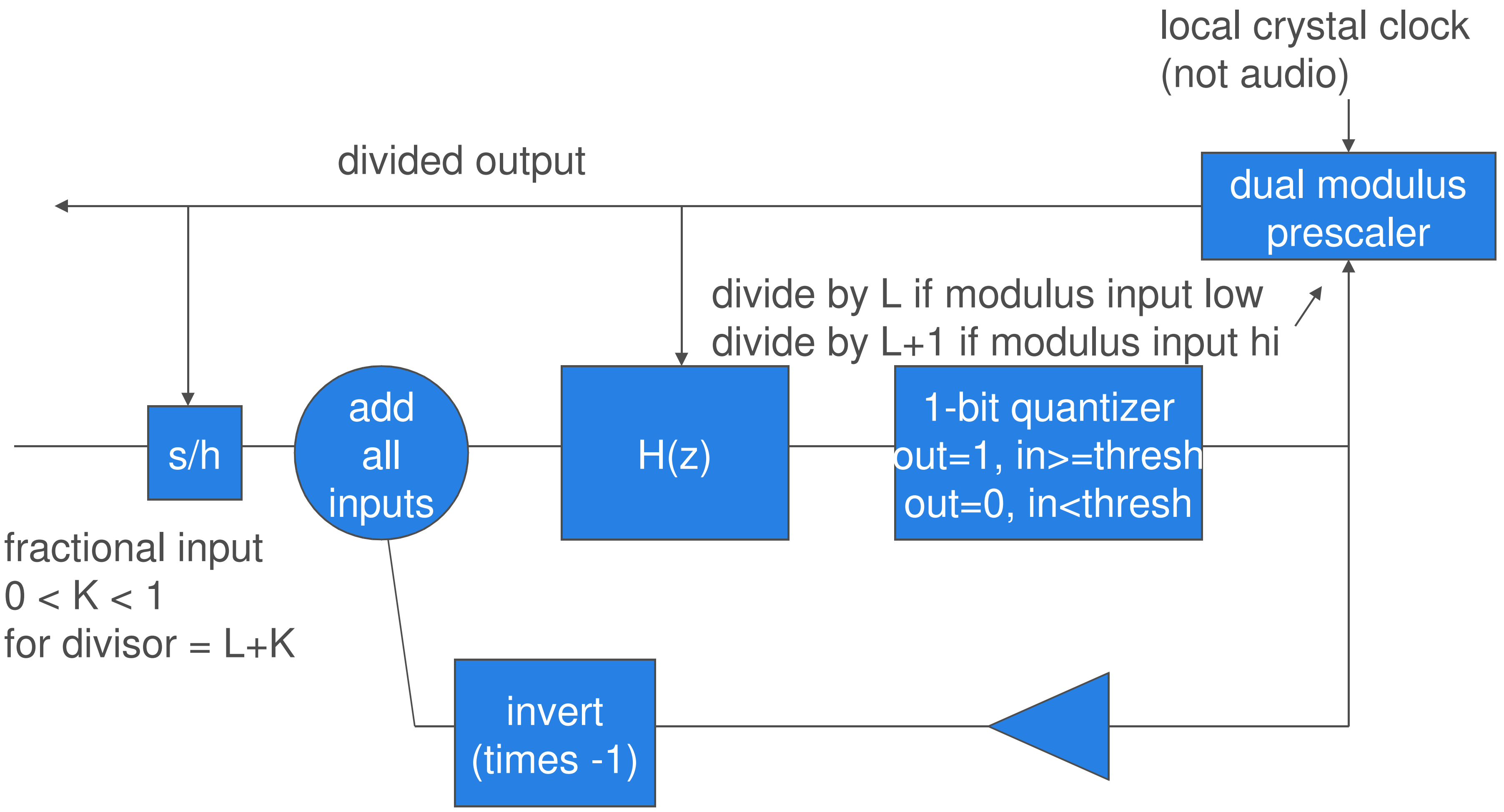


Frequency estimation and reference generation
PSoC3 System PLL to create low jitter audio master clock
24 MHz USB reference clk
Clock Interface
Prescaler /N /N + 1
24 MHz
Delta Sigma Modulator
Ref frequency counter
Integrate shift and hold
USB 1 ms token pulses
Verilog / UDB
Clock Domain 1: 24 MHz XTAL clock global clock routing asynchronous to system clock
Ref clock ~ 1024 kHz
Audio Master 256x Fs
PSoC3 System PLL Generate audio and system clocks
I2S clock 64x Fs
I2S out
I2S interface
Clock Domain 2 – sync to Fs: Audio Master Clock at 256x Fs 256x 48 kHz = 12.288 MHz

When the sums are done right, this process is exact, and no feedback is needed, so the loop can be very fast.

The prescaler output is used as the reference for a conventional PLL multiplier that provides a final rational step-up.

A Fractional Input Noise-shaper

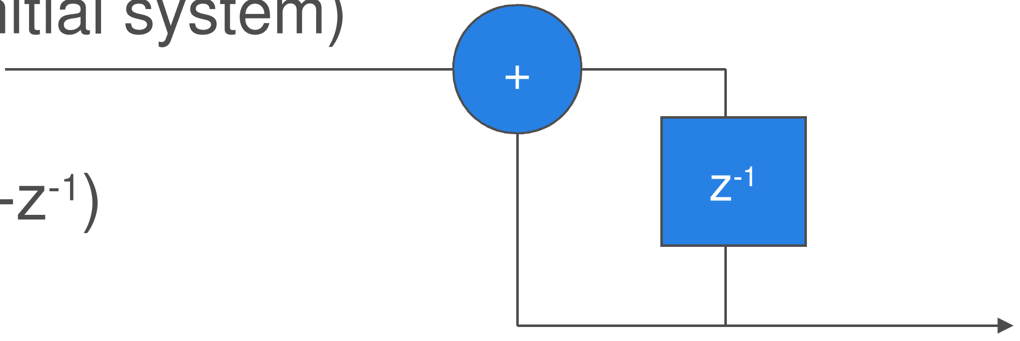


Noise-shaper H(z) transfer functions

1st order (used in initial system)

force NTF = $1-z^{-1}$

gives $H(z) = z^{-1} / (1-z^{-1})$

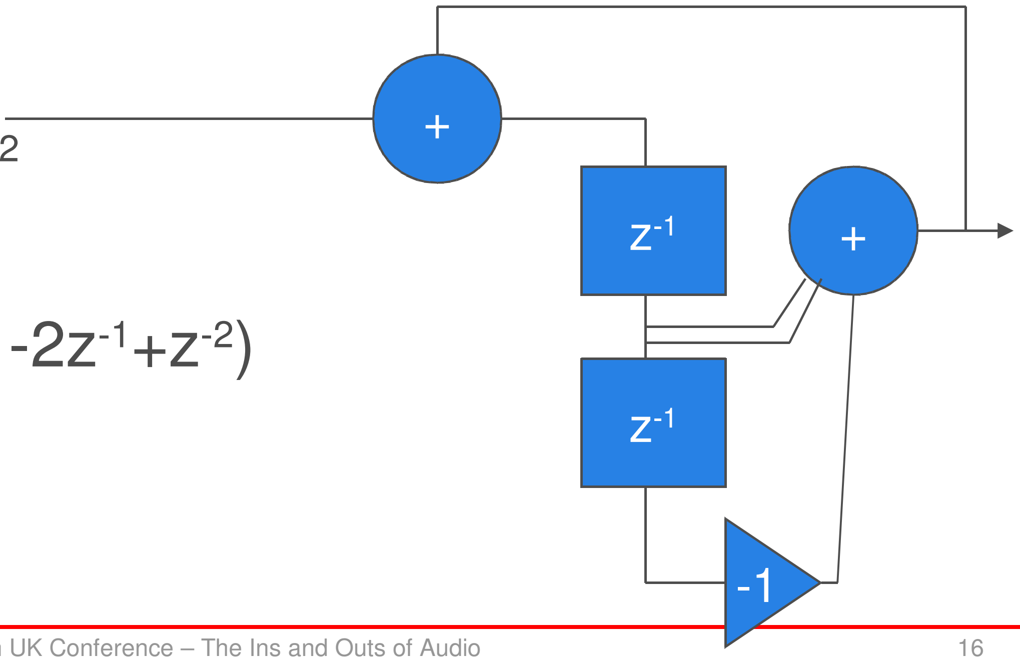


2nd order

force NTF = $(1-z^{-1})^2$

gives $H(z) =$

$(2z^{-1}-z^{-2}) / (1-2z^{-1}+z^{-2})$



Implementation

The entire synthesizer is implemented in the programmable digital hardware of an off-the-shelf PSoC3 Programmable System-on-Chip.

No CPU activity is needed except to change settings if a different master clock frequency is required.

The local oscillator is a standard 24MHz crystal which is also used to clock the USB bus interface.

The adjustable parameters in the design are calculated with a simple spreadsheet. Both standard sample-rate trees are supported, and can be derived from many other standard local oscillator crystal values (25MHz, 26MHz, 27MHz...).

If there's no synchronization input, the synthesizer just free-runs at the desired frequency.

The synthesizer locks in a single frame; buffer length is minimal (768 bytes for 48k/16b stereo in and out, ~2ms).

Further Improvements

Magnitude of jitter, around 600ps pk-pk, is mainly due to the PSoC3 system PLL, which wasn’t designed specifically for audio. But it is nearly all random phase noise.

Tonal behaviour can be seen in simulation with constant input from the SOF measurement. We’ve yet to try higher order loops or dither, because what we already have is deemed good enough by current customers.

Adaptive Synchronous mode can be added trivially by offsetting the modulator’s dither input. ‘Regular’ adaptive mode would be possible with a hardware redesign. Asynchronous modes should be straightforward (but just as messy as with any other solution).

‘Preferred value’ asynchronous mode allows operation with one direct crystal clock, which is also used as the local oscillator to generate the other clock tree.

Sneaky Commercial Pitch

The USB Audio System is a ‘poster child’ for the versatility, ease-of-use and short design cycle time of the PSoC3 Programmable System-on-Chip.

Accessible USB FS Device interface lets you get at the SOF packet timings and customize your endpoints.

“Universal Digital Blocks” have CPLD elements for custom combinatorial logic. plus a datapath array for hardware support of sequential ALU-driven stream processing.

Flexible clock tree design allows multiple independent clock domains for different on-chip processes.

Digital Filter Block gives up to 67M 24x24 MAC/sec for additional audio signal processing not discussed here.

Substantial analogue hardware resources (opamps, comparators, ADC, DAC) support lots of system management and auxiliary product features. Tasty!