

Dante Embedded Platform ASRC Integration

Application Note

Version 1.0

15th November 2023

Copyright

© 2023 Audinate Pty Ltd All Rights Reserved.

Audinate®, the Audinate logo and Dante® are registered trademarks of Audinate Pty Ltd.

All other trademarks are the property of their respective owners.

Audinate products are protected by one or more of US Patents 7747725, 8005939, 7978696, 8171152 and other patents pending or issued. See www.audinate.com/patents.

Legal Notice and Disclaimer

Audinate retains ownership of all intellectual property in this document.

The information and materials presented in this document are provided as an information source only. While effort has been made to ensure the accuracy and completeness of the information, no guarantee is given nor responsibility taken by Audinate for errors or omissions in the data.

Audinate is not liable for any loss or damage that may be suffered or incurred in any way as a result of acting on information in this document. The information is provided solely on the basis that readers will be responsible for making their own assessment, and are advised to verify all relevant representation, statements and information with their own professional advisers.

Software Licensing Notice

Audinate distributes products which are covered by Audinate license agreements and third-party license agreements.

For further information and to access copies of each of these licenses, please visit our website: www.audinate.com/software-licensing-notice

Contacts

Audinate Pty Ltd

Level 7/64 Kippax Street

Surry Hills NSW 2010

AUSTRALIA

Tel. +61 2 8090 1000

info@audinate.com

www.audinate.com

Audinate Inc

4380 S Macadam Avenue

Suite 255

Portland, OR 97239

USA

Tel: +1 503 224 2998

European Office

Audinate Ltd

Future Business Centre

Kings Hedges Rd

Cambridge CB4 2HY

United Kingdom

Tel. +44 (0) 1273 921695

Asia Pacific Office

Audinate Limited

Suite 1106-08, 11/F Tai Yau Building

No 181 Johnston Road

Wanchai, Hong Kong

澳迪耐特有限公司

香港灣仔莊士敦道 181 號

大有大廈 11 樓 1106-8 室

Tel. +(852)-3588 0030

+(852)-3588 0031

Fax. +(852)-2975 8042

Contents

1. Introduction	5
2. Overview of ASRC Operation	5
3. Dante Time is Global	6
4. VCXO, Hardware Time.....	6
5. Software Time, DEP, and Optional VCXO	7
6. DEP Buffer and Time	8
7. When do we Need ASRC?	10
8. Software ASRC - Interpolation vs. Rate Tracking	14
9. Conclusion	17
References	17
Appendix A – DEP Shared Memory API.....	18

Figures

Figure 1, Using PTP and VCXOs to replicate the Dante clock.....	7
Figure 2, Time types	8
Figure 3, Back-to-back D/A and A/D model	10
Figure 4, AVIO with AES3	11
Figure 5, connecting HDMI audio with Dante	11
Figure 6, DEP and asynchronous USB sound card.....	13
Figure 7, ASRC and DSP	13
Figure 8, Software ASRC ingredients.....	15
Figure 9, Interpolated sine wave.....	15
Figure 10, Typical sample time difference between asynchronous audio domains, raw and pre-filtered.....	16

1. Introduction

ASRC stands for Asynchronous Sample Rate Conversion. In essence, ASRC converts a stream of regular digital samples into another, while the two sample rates are independent, asynchronous and slightly different from each other.

While mostly used in the domain of digital audio, ASRC exists for other forms of sampled measurements. For instance, in video it is known as frame rate adaptation.

This application note document provides an overview of the ASRC problem space, with guidance as to when ASRC is needed and the engineering trade-offs that go with it.

We will look at how Dante deals with time, and how it integrates into other systems, both existing and hypothetical. And, of course, where and how ASRC fits in there.

There is more detailed focus on the Dante Embedded Platform (DEP), the software Dante node, and how it relates to ASRC. While we use Dante products as examples, the problem space is universal to all audio systems.

Maybe unexpectedly, this topic is mostly about time. Reference time, from which sample rates and audio times are derived, and how to cross between different reference time domains.

2. Overview of ASRC Operation

The slow drift of two clock sources manifests itself as a subsample audio delay when passing audio data between DEP and other parts of the overall system which is using another clock source. This additional and growing audio delay, if not compensated for, can ultimately introduce audio discontinuities and/or loss of original audio samples.

In order to compensate for this variable delay a constant adjustment of sample rate conversion is needed - this is typically done as part of an Adaptive Sample Rate Converter block/component.

In general, an ASRC block provides two functions:

- fractional sample rate conversion e.g. between 48,000 to 48,001 Hz - the rate of conversion typically changes in response to differences between clock sources used in different parts of the system
- adaptive tracking of the offset between the two clock sources which allows to pick the optimal fractional sample rate ratio

The implementation of an ASRC block as part of the overall solution can have impact on overall system specification in terms of:

- extra delay (latency) added to overall system audio delay introduced when integrating ASRC
- overall system performance, primarily additional processing cycle (CPU impact) caused by the addition of extra signal processing
- audio quality impact due to the inherent distortion introduced by less-than-perfect interpolation

Therefore an integrator should consider the following trade-offs when implementing ASRC as part of their solution:

- *Conversion accuracy and audio quality* - are directly related to the necessary additional delay and CPU load. As a result, ASRC will add to **overall delay** of the system interfacing with Dante Embedded Platform. It can also introduce a small amount of **audio distortion** to the audio samples as part of the fractional sample rate conversion. ASRC solutions often allow a choice of interpolation algorithms which differ in those parameters.
- *Fractional sample rate tracking accuracy and performance impact* – the exact physical sample rate is often unavailable to software, and has to be deduced from other related measurements. These measurements are subject to jitter and other noise and require substantial signal processing for the determination of the exact sample rates. This will scale with the number of individual sample rate ratios that need to be tracked.

3. Dante Time is Global

A Dante network synchronises time references between all nodes, based on a PTP clock master.

This means that all connected devices use a replica of the same clock. Data sources and sinks (A/D converters, D/A converters) are all running at the same sample rate and sample phase. That is, samples are processed at the same moment in time everywhere.

This way, Dante can guarantee low and stable end-to-end latency.

Dante time is kept and replicated using either a hardware device (VCXO), or as a software mapping to another time reference (in DEP).

4. VCXO, Hardware Time

Most Dante devices have a physical clock reference in the form of a VCXO (Voltage-controlled crystal oscillator), which can be controlled by software.

A VCXO is a crystal oscillator whose frequency that can be adjusted, in the order of a few 100 ppm (parts per million). Crystals tend to have tolerances of a few 10s of ppm, so the adjustment range of the VCXO allows a crystal oscillator to match the exact frequency of another one.

A Dante device determines and tracks the exact frequency and phase of the clock master's reference clock and adjust its VCXO to match.

Thus a replica clock is created in each node that tracks the master clock and can be considered identical to it. From this clock we derive, among others, the Dante sample clock, which underpins the audio sample transport throughout the Dante network.

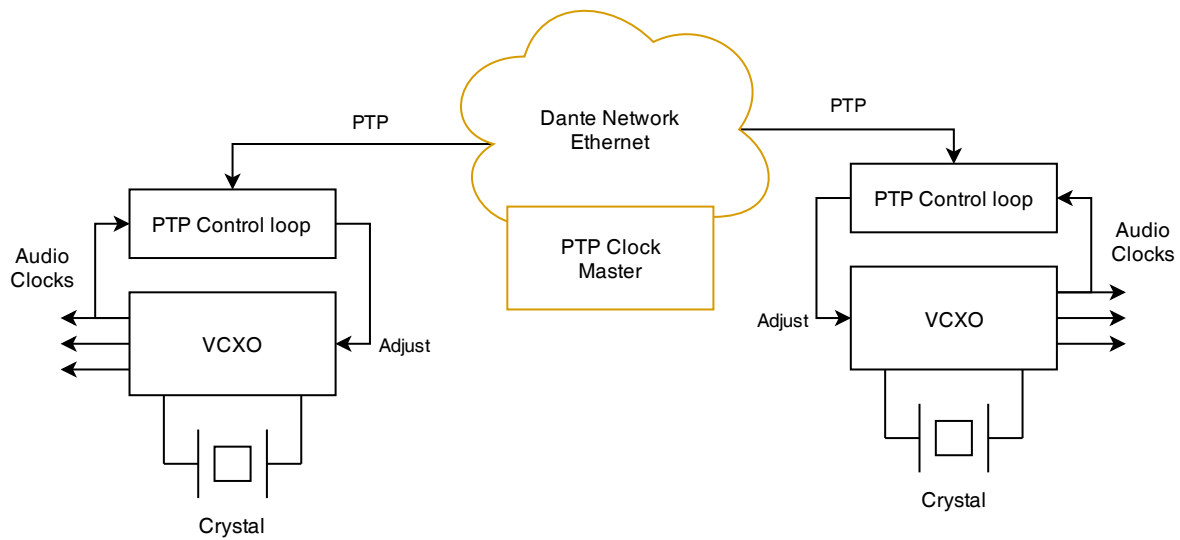


Figure 1, Using PTP and VCXOs to replicate the Dante clock

At the edge of a Dante system, a digital I2S bus running at the Dante sample rate, and/or an audio analogue codec is typically used to transport the samples. In the case of the digital I2S bus, it is a bus master that runs in the Dante clock domain, at Dante sample rate.

The typical implementation of a hardware clocked Dante node operates the CPU and most of the system from the VCXO reference. That is, the system time itself is adjusted to match Dante time.

5. Software Time, DEP, and Optional VCXO

DEP is a software implementation of a Dante audio node, allowing the addition of Dante into a Linux system.

As such, it acts as a "Dante sound card", running in the Dante clock domain. Samples and timing information are presented through either the DEP shared memory API, or optionally mapped as an ALSA sound card (e.g. dsoundcard [2]).

Sound cards, in general, often have their own independent clock references, so DEP operating in the Dante clock domain doesn't present a challenge for typical sound card applications.

To implement a sound-card style media API, the physical sample clock (word clock in the context of I2S) is not available or required. The media APIs provide timing information through comparatively rough buffer levels, and which only matches the Dante sample rate in the long term average.

Thus DEP merely maintains a software mapping which allows it to accurately track Dante time and sample rate relative to a local system clock, without the Dante clock being physically present in the system.

However, DEP can be used in applications that need to cross over into the analogue domain. To this end, it is optionally able to discipline and drive the usual VCXO hardware [3], synchronise it to the Dante clock and provide the physical sample clock to the system. Just like with other Dante nodes, the physical sample clock can drive I2S or codecs.

Note that unlike hardware clocked nodes, DEP with VCXO does not run the system from the VCXO. Instead, a clock feedback from the VCXO to a software accessible timer/counter is required to close the control loop.

6. DEP Buffer and Time

Software runs in, and has access to system time. Linux provides a large collection of time types, and it can be confusing. Generally, `CLOCK_MONOTONIC_RAW` is the foundation, providing a steadily advancing clock at a, maybe inaccurate, but constant rate.

The mathematical definition of a monotonic function means that it either always increases or decreases, smoothly, with no jumps or discontinuities. Such that, for a function $y=f(x)$, there is exactly one x for every possible y value – it's unambiguously reversible.

When it comes to the monotonic clock, the same applies – it always increases, has no jumps or discontinuities. Additionally, the raw monotonic clock also guarantees no sudden slope changes (kinks), and any changes to the slope are gradual and only caused by physics, e.g. crystal oscillator drift over temperature.

Other clock types build on the monotonic time, applying rate adjustments and corrections and offsets and leap seconds for various time standards (TAI, UTC, local time, daylight savings, etc). Each one of these has features or events that violate the raw monotonic condition.

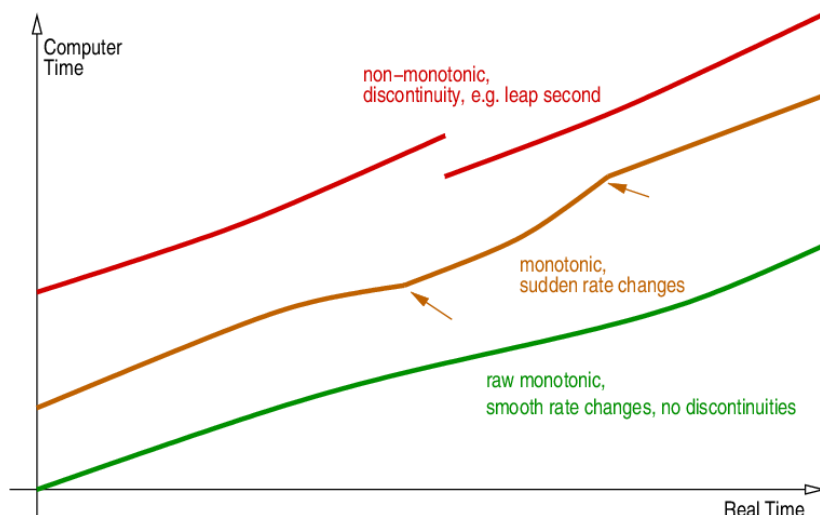


Figure 2, Time types

When we talk about system time here, we mean the `CLOCK_MONOTONIC_RAW` type, the only one that is guaranteed stable and smooth.

The DEP shared memory buffer forms a media API (sound card interface), allowing applications to exchange audio data with the Dante network. The full definition and documentation of the API can be found in `Buffers.hpp`, in the DEP examples.

To aid with accurate timing extraction, the DEP shared memory buffer provides the period count (`time.period_count`) with a timestamp (`time.monotonic`).

It is crucially important to access these two values in an atomic fashion, or an update race can ruin the measurement.

However, the usual synchronisation mechanisms (e.g. lock, mutex), come with a performance cost, so this API is set up to use lockless updates. To this end a third value is provided, an 'updating' flag (in `metadata.flags`). DEP sets the flag, updates period count and timestamp, then un-sets the flag.

The application reads all 3 values in a loop, and exits when they agree with the previous read and the flag is not set. The astute reader may notice that this covers all possible race conditions with one exception - if the application has a higher real-time priority than DEP, **and** runs on the same processor core, **and** the CPU load is such that the previous period processing pre-empts the next period update, it is possible to deadlock. At least one of these conditions should be false.

Please see the `DanteTimingTrace.cpp` application in the *DEP_examples*, which implements an example of the atomic data read, in `traceNetworkMonotonicTime()`.

The period count represents the Dante sample time and indirectly the sample rate. The period is a (configurable) block of samples, 16 by default. When the period count updates, it marks the point in Dante time corresponding to that number of samples becoming ready.

However, by the time software (both DEP and application) can become aware of a period count change, it is in the past, and the exact elapsed time is unpredictable.

In order to compensate for the software delay, the provided time stamp represents the true moment when the period count changed, but expressed in the local system time.

It is thus possible to calculate the Dante time (or sample position) for 'now', by taking the system time now and fractionally adjusting the period/sample count by the elapsed time from the timestamp.

$$DanteTime = \frac{DantePeriod * SamplesPerPeriod}{SampleRate} + (TNow - TimeStamp)$$

Note that technically, this calculation should use the current fractional sample rate of Dante relative to the system clock. However, when using the nominal sample rate (e.g. 48000) instead of the relative one (e.g. 48001), the error over the elapsed time interval of a few 100us can be safely ignored.

7. When do we Need ASRC?

Now we can talk about where ASRC fits into a system. As a useful mental model, ASRC is the digital equivalent of crossing between sample clock domains by using back-to-back D/A and A/D converters.

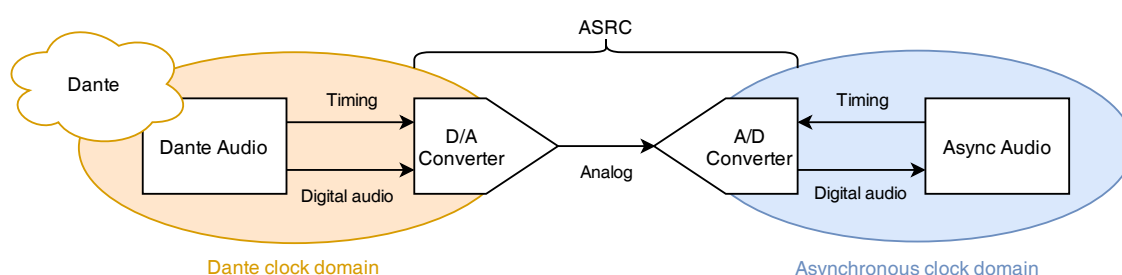


Figure 3, Back-to-back D/A and A/D model

ASRC is reconstructing (mathematically) the original analogue signal from the input samples and resamples it at different moments in time, corresponding to the output sample timing.

We need ASRC in situations where we have to move digital samples between independent (asynchronous) clock domains.

The Dante network is fully synchronous, so we don't need ASRC within Dante. We also don't need ASRC to cross into the analogue domain, as long as the analogue converters are synchronous with the Dante clock.

For the purpose of this discussion, we consider "*asynchronous*" to mean "asynchronous relative to the Dante clock". Conversely, "*synchronous*" means using the Dante clock or a replica of it.

Here are a few examples of where ASRC is required (also see Clocking and Audio Bridging section in System Pre-qualification guide [1]).

AES3

We'll use AES3 as a stand-in for asynchronous (not Dante) digital audio buses – there are others. Such buses often are source-synchronous. That is, the transmitter of a signal generates the clock and sends it alongside the samples.

Attaching this to a Dante system means that while transmitting samples from Dante is trivial - we can use the Dante clock, receiving samples into the Dante system requires ASRC.

As an example, the AES3 AVIO uses a hardware ASRC that sits between the Dante clocked I2S bus and AES3.

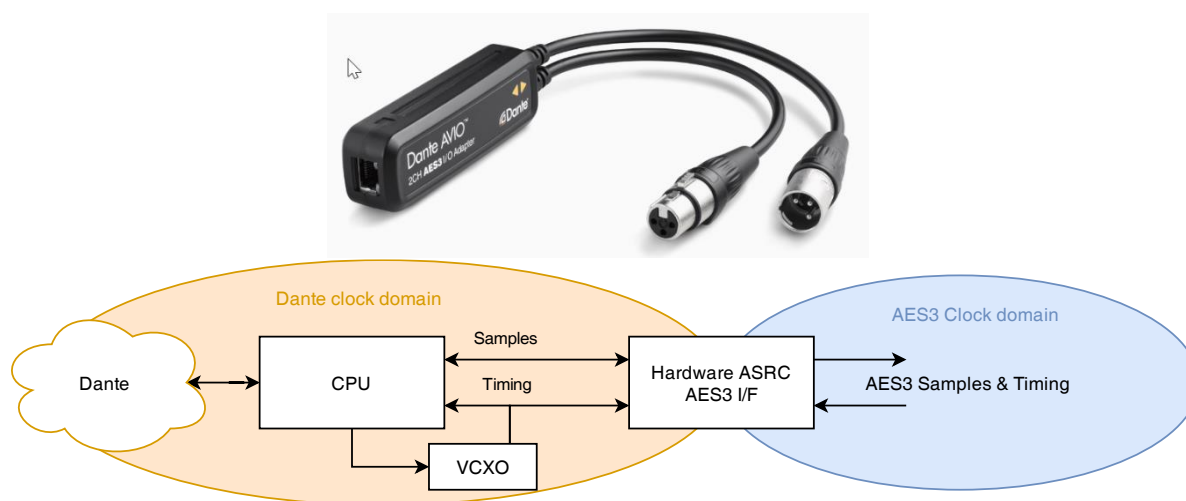


Figure 4, AVIO with AES3

At this point we should note that performing ASRC between sample streams that come with physical sample clocks (word clock), is relatively easy, in hardware. While the mathematics of digitally emulating the back-to-back perfect D/A & A/D converters are the same, the hardware sample clocks indicate exactly where to interpolate each sample.

HDMI

A common scenario is converting between Dante and digital video sources and sinks. Video signals are typically source (or genlock) synchronous, and those time references are independent of, and asynchronous to the Dante clock.

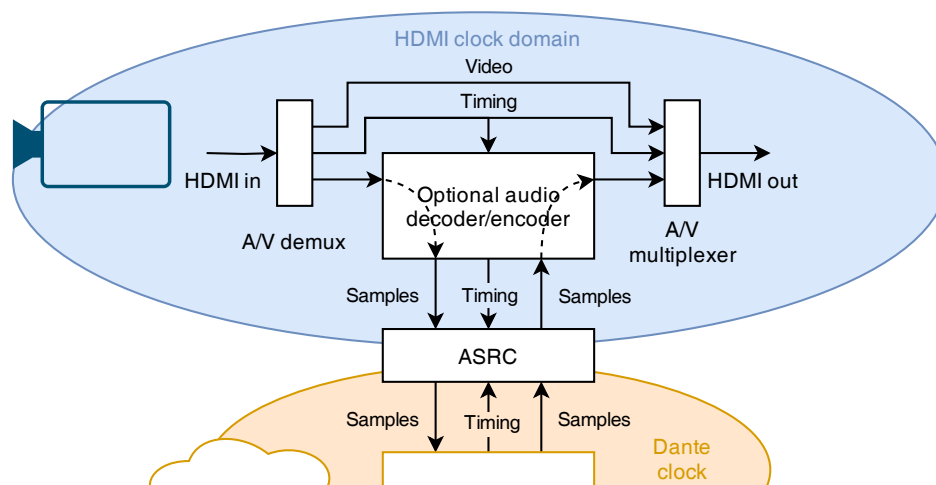


Figure 5, connecting HDMI audio with Dante

Note that some HDMI audio encodings (e.g. Dolby surround, compression) are not, directly, representing PCM samples. Rather, it is a digital data stream that must arrive bit-accurately at the relevant decoder. ASRC would ruin the encoding.

However, the decoded digital audio PCM sample channels are in the HDMI clock domain still, and need ASRC to bridge into the Dante clock domain.

Asynchronous Sound Cards

Many sound cards have their own independent clock reference, and, in a manner similar to the DEP shared memory API, present sample streams with timing information, at varying levels of reliability.

Beside interfacing with the analogue world, sound cards are often the common software abstraction that is used for digital audio transports. The dsoundcard example [2] provided with DEP represents the Dante transport, most PC sound cards will have representation for S/PDIF or AES3 for instance, HDMI audio transmitters or receivers show up as sound cards, etc..

So this category covers a lot of the others, from an application / OS point of view.

We need to distinguish between situations where all used sound cards in a system are in the Dante time domain - no ASRC needed, or when we must bridge between independent clocks where we do need ASRC.

For instance, Dante Virtual Soundcard (or dsoundcard [2]) is a sound card that runs in the Dante domain, and doesn't need ASRC, if that's the only thing the application works with.

The same applies when an application works with multiple media APIs, but they all are running in the Dante clock domain – this is the case when the other sound cards / codecs are also clocked by the Dante VCXO. There is an example (DepAlsaBridge.cpp) that moves samples between DEP and an ALSA device, assuming the ALSA device is synchronous to Dante.

Conversely, Dante Via needs to bridge between a Dante soundcard and other independent ones, so it uses ASRC. This makes it possible, for instance, to play back a Dante stream through PC speakers, which use an asynchronous sound card.

A typical example for mandatory ASRC would be a Dante to USB-Audio bridge. USB-Audio devices usually have their own crystal reference (or worse), transport the samples over USB which is asynchronous to either clock, and terminate in an media API that attempts to represent the USB device's clock domain.

Of course, even embedded or on-chip sound cards can be asynchronous relative to the Dante clock domain, and often relative to the system clock. The only exception are Dante products that clock everything from a replica Dante clock.

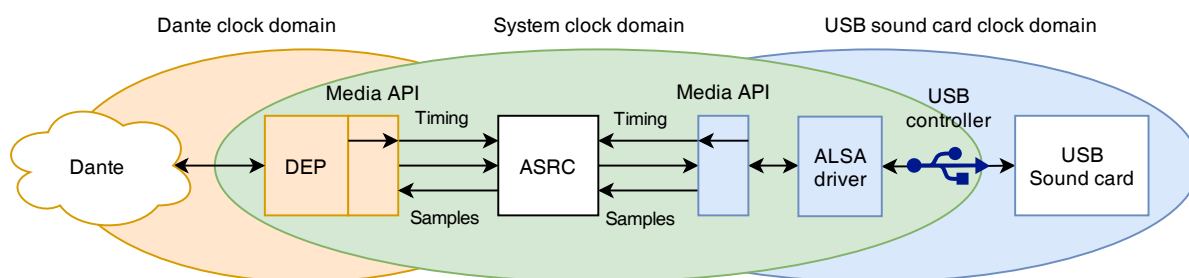


Figure 6, DEP and asynchronous USB sound card

Note that ALSA comes with an ASRC solution called alsaloop [2]. This can be set up to perform ASRC between arbitrary ALSA sound cards, including dsoundcard [2].

Asynchronous DSP

Frequently, DSP systems, just like Dante, pick a clock reference and run everything within that clock. To interface Dante to such a system, the obvious solution is to pick the Dante clock for everything.

When that is not an option, ASRC is needed to adapt Dante channels to DSP channels.

A typical challenge in such a system is that both (software) ASRC and the DSP system are expecting to talk to, and move samples between media APIs (like sound devices).

A media API, while often symmetrical in regards to sample transport (capture/playback), is asymmetrical for timing. That is, an API provides timing to the application, and an application consumes it. The source of the timing is the driver and ultimately the underlying hardware.

To connect ASRC to an asynchronous DSP system, we have a problem, as a conventional media API between the two has to face one way or the other. ASRC needs to operate on the application side at both ends, and so do most DSP systems. A proxy API between them would need to provide timing to both ASRC and DSP, and extract it from the DSP's clock domain, e.g. a sound interface at the far end.

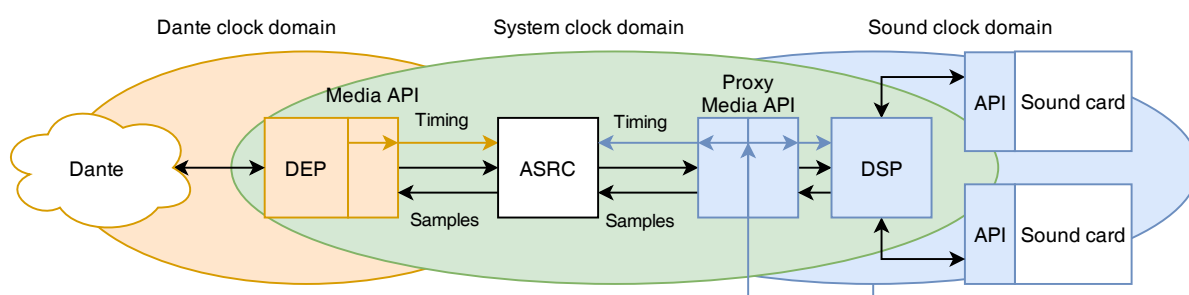


Figure 7, ASRC and DSP

In a sense, the proxy media API is comparable to dsoundcard, which acts as a proxy for the Dante clock domain.

Without creating such a proxy media API to interface with an asynchronous DSP system, the remaining option is to make ASRC part of the DSP system itself. There it has access to both the Dante clock (via DEP or dsoundcard) and the DSP clock (via the sound card APIs).

8. Software ASRC - Interpolation vs. Rate Tracking

Software ASRC consists of two main components, a resampling block and a rate tracker. A resampling block acts nominally like a back-to-back D/A and A/D arrangement, while the rate tracker is in charge of determining the exact points in time at which output samples are generated.

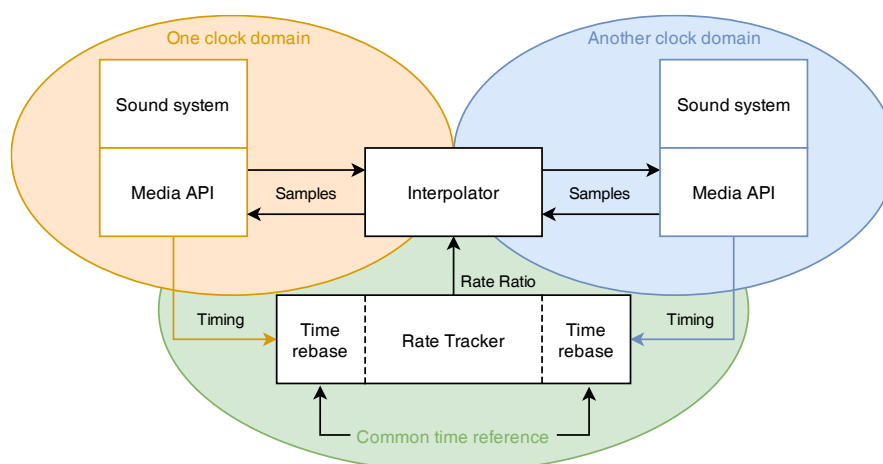


Figure 8, Software ASRC ingredients

Maybe surprisingly, resamplers tend to be the easier part of the system. They are typically implemented as digital, fractional interpolators, which have been the subject of research a long time ago, and which are, mathematically speaking, solved.

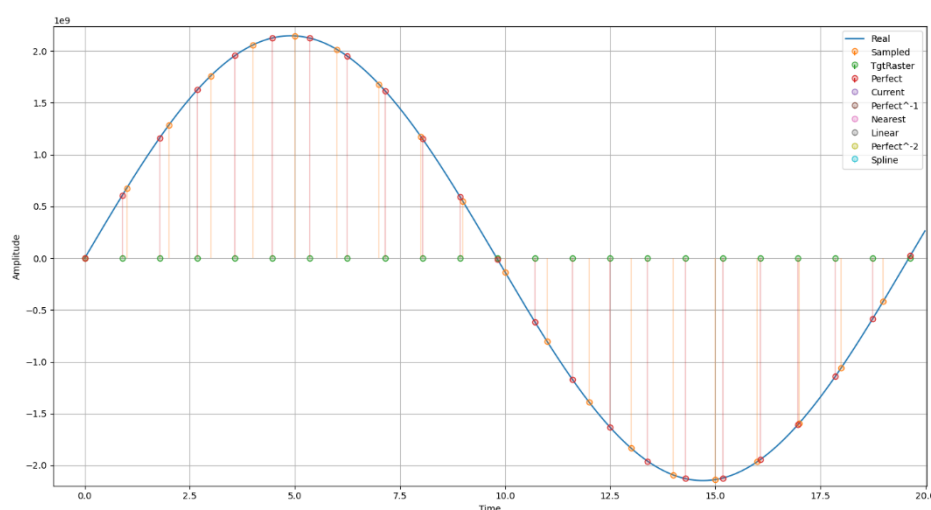


Figure 9, Interpolated sine wave

Current research around this topic mostly covers implementation efficiency on various platforms, hardware or software. While important, it mostly affects the quality / performance trade-off.

That is, CPU requirements go up when better interpolation precision is needed. This is also correlated with latency, as better interpolators must delay the signal more - although interpolators precise enough for 24bit audio only require an extra delay of around 10 samples.

Beyond that, better efficiency means that for a given precision, more channels can be resampled on a given platform, or higher sample rates may be accommodated.

The resampler will generate a slightly different number of output samples relative to the input, corresponding to the rate ratio. This ratio comes from the rate tracker.

The rate tracker can span a wide range of difficulty levels. It often takes the form of measurement filters and control loops, with the aim of producing an accurate estimate of the sample rate ratio.

While the two sample rates are usually derived from relatively stable clock references (crystal oscillators), they will have a small offset in the order of, say, 40ppm (parts per million) and are also subject to drift over temperature and age, which is smaller still.

In the best case - a hardware ASRC system, we can directly look at the two sample clocks and measure the phase offset for each sample. This can be made very precise with a modest hardware effort, and can drive the interpolator directly.

As we move away from the naked hardware and further into software, the sample clocks are not typically available any more, and must be estimated. This means using other measurement data as a stand-in for the sample clocks, like buffer fill levels and interrupt timing.

These are typically subject to massive measurement noise, caused by the time jitter that is inherent in software processing.

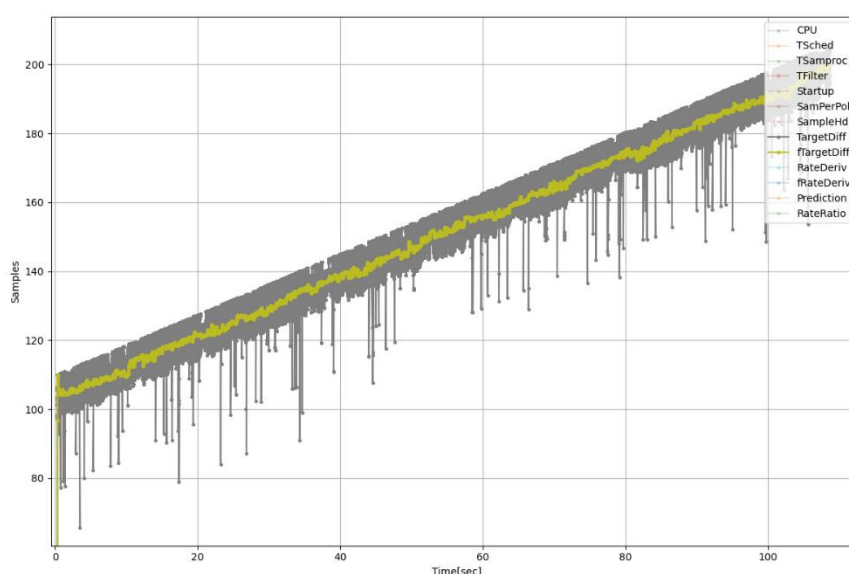


Figure 10, Typical sample time difference between asynchronous audio domains, raw and pre-filtered.

So, we must correct data where possible, reject bad data, filter out the noise and run control loops with the purpose of driving the interpolator with just the right rate ratio and sample phase.

Such that, in the long run, and with minimal jitter, the two sample streams can each be accurate in their own clock domain, while buffers between them maintain an exact nominal level. Incorrect rate estimation leads to the buffers slowly emptying or filling over time.

9. Conclusion

In some situation Adaptive Sample Rate Conversion (ASRC) needs to be considered as part of integration of Dante Embedded Platform (DEP) into overall system design e.g. when integrating DEP with systems with HDMI, AES3 or asynchronous audio sound cards.

DEP shared memory interface provides the timing information as well as notification mechanisms for audio sample data exchanged between OEM application and DEP internal buffers which allow to integrate a ASRC block with DEP.

While use of ASRC is necessary and completely typical approach to integration of Dante into OEM devices, it is necessary to consider the overall system performance requirements and possible trade-offs ASRC introduces.

Audinate's sales team as well as support and field engineering teams can offer further advice and recommendation on what Dante products meet specific performance recommendations and to consider when integrating ASRC with Dante Embedded Platform.

References

1. The 'Clocking and Audio Bridging' section of [Key Platform Considerations](#) in the DEP System Pre-qualification Guide
2. [DEP Example Audio Applications](#) in the DEP Programmer's Guide
3. [Hardware Clock](#) in the 'Clocking' section of DEP Programmer's Guide

Appendix A – DEP Shared Memory API

Below code extracted from DEP_examples `Buffers.hpp` header file (found in `dep_examples/dep_audio_buffers/include` folder) shows the memory buffer structure available to developers building interface to application using DEP.

```
///  
/// Overlay structure for the shared memory buffer header block  
///  
typedef struct buffer_header  
{  
    /// Describes the shared memory layout and state  
    struct  
    {  
        /// Set to zero when the shared memory buffer is not yet configured or becomes  
invalid.  
        /// Set to DANTE_BUFFERS_HDR_MAGIC when the shared memory buffer has been fully set  
        /// up and is valid for use.  
        uint32_t magic_marker;  
  
        /// Total length in bytes of the shared memory. Includes headers and audio channel  
buffers.  
        uint32_t buffer_length;  
  
        /// Total length in bytes of the metadata header  
        uint32_t metadata_header_length;  
  
        /// Flags used for various buffer operations. See DANTE_BUFFERS_FLAG defines.  
        uint32_t flags;  
  
        /// Offset in bytes to the firxt TX channel buffer from the start of this header  
        uint32_t first_tx_channel_offset_bytes;  
  
        /// Offset in bytes to the firxt RX channel buffer from the start of this header  
        uint32_t first_rx_channel_offset_bytes;  
  
        /// Offset in bytes to the timing object subheader from the start of this header  
        uint32_t timing_object_subheader_offset_bytes;  
  
        /// Incremented at the start and end of a buffer reset operation  
        uint32_t reset_count;
```

```
} metadata;

/// Describes the audio data stored in the channel buffers
struct
{
    /// When non-zero, this value is the currently configured device sample rate.
    /// When zero, it means the sample rate is in the process of being changed.
    uint32_t sample_rate;

    /// Encoding of the audio data in the channel buffers. Currently always
    DANTE_ENCODING_PCM32.
    uint32_t encoding;

    /// The total number of samples that can be stored in the buffers for each channel
    uint32_t samples_per_channel;

    /// Total size in bytes for each channel buffer
    uint32_t bytes_per_channel;

    /// Total number of available TX channels (minimum of configured and activated number
    of channels)
    uint32_t num_tx_channels;

    /// Total number of available RX channels (minimum of configured and activated number
    of channels)
    uint32_t num_rx_channels;
    uint32_t pad7;
    uint32_t pad8;
} audio;

/// time fields
struct
{
    /// The epoch represents the PTP time at which the device last achieved PTP
    synchronisation.

    /// To get the complete epoch time, convert @ref epoch_seconds and @ref epoch_samples
    into the same units

    /// and add them together. For example, to get epoch time in samples:
    /// (epoch_seconds * sample_rate) + epoch_samples

    /// @note The epoch is initially zero when the device first starts up before it has
    achieved
```

```
    /// PTP sync for the first time. Subsequently the epoch values are only updated upon
each sync

    /// event but not on sync loss. That is, this value cannot be used to determine the
    /// device's current PTP sync state apart from the initial state (yet to sync for the
first time).

    uint32_t epoch_seconds;

    /// The sub-second portion of the epoch in unit of samples. See @ref epoch_seconds for
more details.

    uint32_t epoch_samples;

    /// This value gives the number of samples that will elapse before DEP signals to the
application

    /// that there is audio data available. A lower value can allow the application to
wake up

    /// more often to process audio data but with the tradeoff that CPU load will
increase.

    uint32_t samples_per_period;

    /// This value is incremented every time a period has elapsed.
    /// Applications can use this value to determine how many periods of audio data can be
read/written

    /// since the last time it woke up.

    uint64_t period_count;

    /// Currently unused

    uint32_t clock_drift_ppb;

    /// The local monotonic time at the current period_count value.
    /// The monotonic clock source is CLOCK_MONOTONIC_RAW.
    /// (monotonic, period_count) timing pair values can be used to track the rate change
    /// of the local time domain against the PTP time domain.
    /// @note The (monotonic, period_count) values are not updated atomically. But such
updates are

    /// bracketed by setting DANTE_BUFFERS_FLAG__TIMING_UPDATE_SYNC prior to starting an
update and cleared

    /// when both fields have been updated. See the DanteTiming example code for a guide
to how

    /// these two timing fields can be read to ensure correctness and consistency.

    uint64_t monotonic;
} time;
} buffer_header_t;
```