PortAudio and Media Synchronisation - It's All in the Timing

Ross Bencina

email: rossb@audiomulch.com
website: http://www.portaudio.com

Abstract

PortAudio is an open source 'C' language API and library for implementing cross-platform real-time audio applications. This paper describes recent additions to the PortAudio API designed to assist in implementing synchronisation between real-time audio and other time-based data such as MIDI and computer graphics. Examples are presented that illustrate synchronisation of a graphical display to real time audio, low jitter MIDI triggering of software synthesized audio, and synchronising software synthesized audio to an external time source such as MIDI clock. Some of the challenges of implementing the PortAudio synchronisation infrastructure on Microsoft® Windows® are discussed.

1 Introduction

PortAudio[1] is an open source 'C' language API (application programming interface) and library for implementing real-time audio applications. PortAudio is implemented as a layer on top of native platform specific audio services and is available on a variety of operating systems including Windows®, MacOS® (9 and X) and Linux®. PortAudio is used by computer music systems such as Miller Puckette's Pd and the author's AudioMulch®, and as a platform for education in audio programming.

Over the past two years a range of revisions to the PortAudio interface have been proposed.[2] At the time of writing these proposals had been approved and were being implemented for the forthcoming V19 release. A number of the revisions are designed to increase the ease and accuracy with which PortAudio applications may generate audio synchronised to, or triggered by, external sources. This paper explains three common synchronisation scenarios and the timing information they require. It then presents the interfaces through which the PortAudio V19 API provides this information. Finally, issues that have been encountered while implementing these interfaces on Microsoft Windows are discussed.

The three synchronisation scenarios considered are: displaying visual feedback synchronised with audio output; generating audio events under MIDI control; and generating audio synchronised to an external time-base such as periodic MIDI clock messages.[3] These scenarios were chosen because they are commonly encountered in computer music systems.

2 Buffering Basics

Before presenting the three synchronisation scenarios, we introduce a conceptual model of real-time audio generation. This model assumes a 'buffered' or 'block-wise' approach, where by a continuous stream of audio samples is generated a buffer at a time – each buffer consisting of a sequence of audio samples. The audio subsystem periodically calls upon the application to generate a buffer of samples. For low-latency applications buffer lengths are typically between 16 and 256 samples per channel, although buffer lengths of thousands of samples have not been uncommon on recent desktop operating systems.

In order for the audio hardware to play generated audio buffers as a continuous stream of samples it is necessary to use at least two buffers. This is so that one buffer may be filled while the other is being read by the audio hardware. The two buffer case, also known as double buffering, is shown in figure 2.1 (a) – when the playback position moves past the end of a buffer, it is immediately returned to the application to be refilled, once refilled it is placed at the end of the buffer queue ready for future playback. A common, although undesirable, feature of desktop operating systems is that unpredictable delays may occur in returning buffers to the application or requeuing buffers with the audio driver. One practical method for reducing the impact of such delays is to utilise more than two buffers. Figure 2.1 (b) shows the case of using three buffers. Using more than two buffers increases audio latency but also increases reliability by reducing the risk of 'buffer underrun', a condition wherein the playback position reaches the end of one

buffer without the next buffer being available for playback.

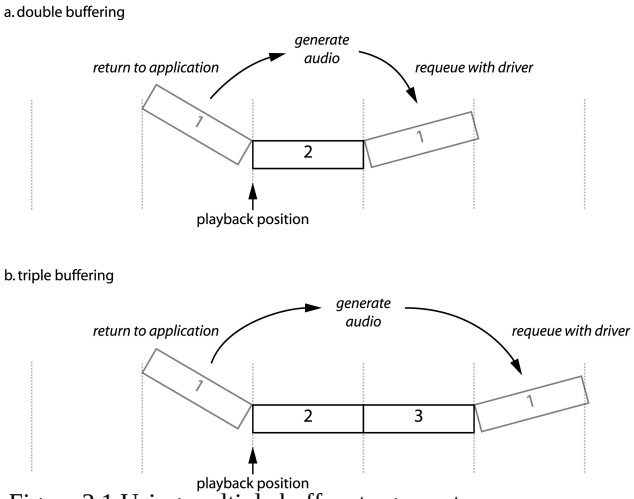

Figure 2.1 Using multiple buffers to generate a continuous stream of audio samples

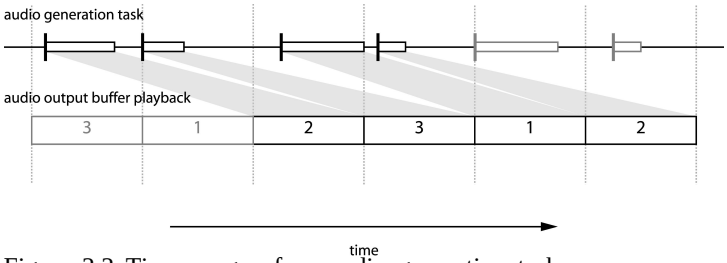

Figure 2.2 Time usage of an audio generation task and relative playback time of corresponding audio output buffers.

As alluded to in the previous paragraph, at any instant, the audio currently being played by the hardware was generated by the application at some time in the past. Conversely, at any instant, the audio currently being generated will be played by the audio hardware at some time in the future. The time taken to generate each buffer must be the same as, or less than, the time taken to play the buffer. On a general-purpose computer, audio generation must proceed fast enough to service all buffer generation requests in a timely manner and also leave sufficient time for the computer to perform other tasks such as updating the user interface or reading and writing from and to disk.

Figure 2.2 shows an example of the temporal relationship between audio generation tasks and audio buffer playback in a well behaved triple buffered system. The CPU time used to generate each buffer stays below the buffer period (the time it takes to play the buffer). The delay from when samples are generated to when they are played is somewhere between one and three buffer periods. Although this delay cannot be known precisely, it has an upper bound of one less than the total number of buffers and is referred to here as the audio generation latency.

The block-wise model described above is just one way to organise audio buffering. Other methods, such as a ring buffer with write and playback pointers, allow for non-uniform block sizes. Some audio APIs expose their internal buffering model to the application programmer, while others hide their internal buffering model. PortAudio is built on top of host audio APIs, and as such does not have the option of dictating a buffering model. Instead it hides the host's model of buffer organisation and simply provides a callback requesting the application to fill a buffer. In addition to a callback, the V19 API offers an alternative: a write() function which allows the client to enqueue arbitrary-length buffers of samples. In both cases the buffers may be of fixed or variable length depending on the client's requirements.

The remainder of this paper uses the conceptual model described in the first part of this section, that is, of the application being called at regular intervals to generate audio into a ring of fixed-length buffers. Although some host APIs employ different techniques, and/or the client may choose to use variable-length buffers, the fixed buffer size model provides a useful framework with which to explain the synchronisation concepts presented in the following sections. Once understood, the concepts may easily be extended to the variable sized buffer case if necessary.

3 Synchronising Graphical Display

The first scenario we will consider is the display of visual feedback synchronised with generated audio. The need for synchronised visual feedback often arises in graphical user interfaces and may take the form of real-time level indicators such as VU meters and spectral analysers, or of playback position indicators as seen in audio editors and music sequencers. This scenario may also be considered as an example of implementing audio-driven synchronisation in visually dominated systems such as video playback and interactive computer games, although in practice it may not be practical to lock such systems directly to the audio playback rate.

The central issue in synchronising a visual display with generated audio is determining when a particular audio event is going to emanate from the computer's

audio hardware. In a well behaved system the audio generation task will always precede playback of the generated audio.

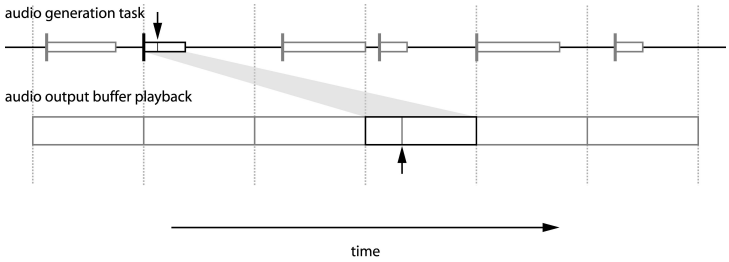

Figure 3.1 Correspondence of generation time and output time assuming constant CPU usage per generated sample.

Figure 3.1 shows the relationship between an audio generation task and the playback time of the corresponding audio buffer. The figure also shows the relationship between the time at which a specific audio event within the buffer is generated and the time at which it is played back. In this and later examples, the figures show markers positioned according to the assumption that every audio sample takes equal time to generate, even though this may not be the case in practice.

Both the time at which the audio task begins generating a buffer, and the rate at which it generates the buffer are uncertain, and as a result may not be usefully applied to the goal of determining when the audio event will emanate from the computer. On the other hand, the playback rate (also known as the sample rate) and each buffer's start time are known or may be predicted with relative accuracy.

Figure 3.2 shows the temporal relationship between a buffer start time and the offset of an event within that buffer. Given an event offset in samples from beginning of the current buffer, its playback time may be calculated as follows:

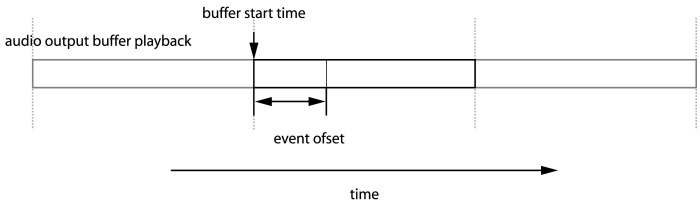

Figure 3.2 The relationship between buffer start time (the output time of the first sample in a buffer), and an intra-buffer event offset

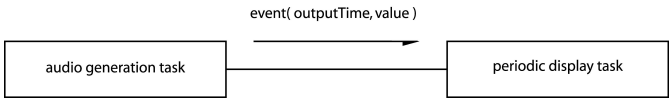

Figure 3.3 Visualised events are sent to the display task as asynchronous messages

```
eventPlaybackTimeSeconds =
        bufferStartTimeSeconds +
        (eventOffsetSamples /
        sampleRate)
```

As the audio generation task produces events, or detects features of the generated audio (such as VU level changes) it can calculate the output time of these events according to the above equation and post them asynchronously to a display routine.

The relationship between the audio generation task and a display task is shown in figure 3.3. The display routine runs periodically, checks the current time, and updates the display according to recently received audio events whose playback times are at or before the current time.

In order for the above scheme to work, the audio generation task must have access to the sample rate and the buffer start time for each buffer. The display task must have access to the current time according to the same clock that is used to express the buffer start times. The way in which PortAudio provides this information is described in section 6.

4 Low-Jitter MIDI Triggered Audio Generation

The second scenario we will consider is generating low-jitter audio events triggered by externally generated MIDI messages. This situation may arise, for example, when implementing a software synthesizer. The same technique can also be applied to implementing low-jitter control by other message based sources such as a mouse.

Figure 4.1 shows the relationship between a MIDI reception task, such as a MIDI callback or operating system queue, and the audio generation task. MIDI events arrive asynchronously and are later processed by the audio generation task. Although the audio generation task may theoretically access received events at any time, it is most common to process all events prior to commencing audio generation for each buffer. Unless the audio buffers are extremely short it is important to calculate an appropriate offset from the beginning of the buffer for each event. The alternative is to quantize event onsets to buffer boundaries, which will likely introduce unwanted rhythmic artefacts.

Figure 4.2 shows the temporal relationship between the reception time of MIDI events, the

execution time of the audio generation task, and the

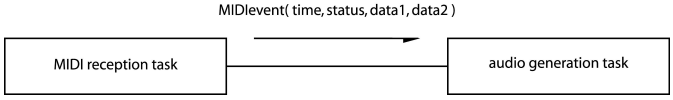

Figure 4.1 MIDI events are sent to the audio generation task as timestamped asynchronous messages

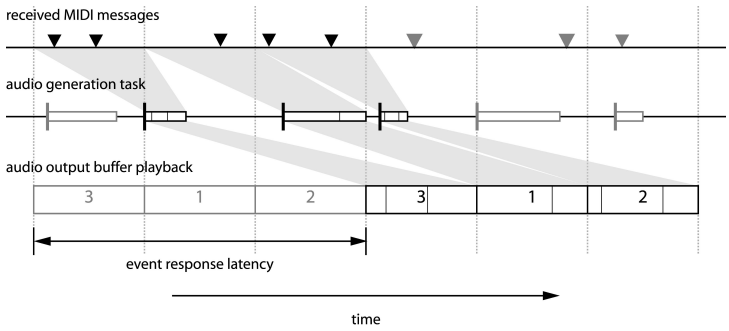

Figure 4.2 Generation of audio events triggered by MIDI input using a fixed event response latency

playback time of generated audio buffers. The audio generation task is only guaranteed to have access to MIDI events that were received prior to the commencement of the current buffer period. Although more recent MIDI events may be available (as is the case in the third period of figure 4.2, where the task executes late enough to have access to an event within the same buffer period) these events should be retained for generating later buffers.

Given a set of MIDI events that are to be synthesized within the current buffer, the offsets into the buffer may be calculated as follows:

```
bufferOffsetSamples = sampleRate
    * ( receptionTimeSeconds -
(bufferStartTimeSeconds –
eventResponseLatencySeconds) )
```

The event response latency is the total duration of all buffers in the buffer ring, or alternatively, the audio generation latency (as defined in section 2) plus one buffer period.

In order to implement the above low-jitter method of synthesizing MIDI triggered audio events, the audio generation task needs access to the buffer start time, the event response latency and the sample rate. The reception time of each MIDI event must be available and expressed according to the same time base used for the buffer start time. The way in which PortAudio provides this information is described in section 6.

5 Synchronising Generated Audio with an External MIDI Clock

The final scenario we will consider is that of synchronising generated audio to an external MIDI clock. Synchronising to an external MIDI clock source is common when multiple rhythmic generators, be they software or hardware, are employed within the same musical piece. Synchronising to other clock sources such as MIDI Time Code (MTC) or SMPTE linear time code is common in film and video post production. The concepts described below may be equally applied to implementing these forms of synchronisation.

The discussion here is limited to determining the location of output clocks (i.e. the MIDI clock phase) in generated audio buffers – this may be sufficient for triggering a low-resolution MIDI sequence, however more demanding applications, such as continuous synchronisation of audio playback to timecode, require the use of additional techniques that may include high-resolution clock reconstruction and continuous playback rate variation. These techniques are beyond the scope of the present paper.

Figure 5.1 shows the temporal relationships between MIDI clock message reception times, output times of predicted MIDI clocks, audio generation task execution times, and generated audio buffer playback times. For each audio buffer, the audio generation task uses previously received MIDI clocks to predict the MIDI clocks (or the continuous MIDI clock phase). Note that the audio generation task has access to all prior clocks, perhaps even clocks within the current buffer period, and unlike the scenario described in the previous section may make use of the most recent clocks to increase the accuracy of the prediction.

The MIDI clock rate and phase may be recovered from recently received timestamped MIDI clock messages using a number of techniques which vary in accuracy and complexity. For example, the clock rate may be determined by calculating the time between adjacent clocks, and the phase may be derived either directly from the reception time of the most recent clock, or by using some form of jitter reduction, such as averaging the phase of recently received clocks. More accurate methods include taking a Least Squares line of best fit through a window of recently received clocks,[4] or by employing an adaptive filter such as a Kalman filter variant.[5]

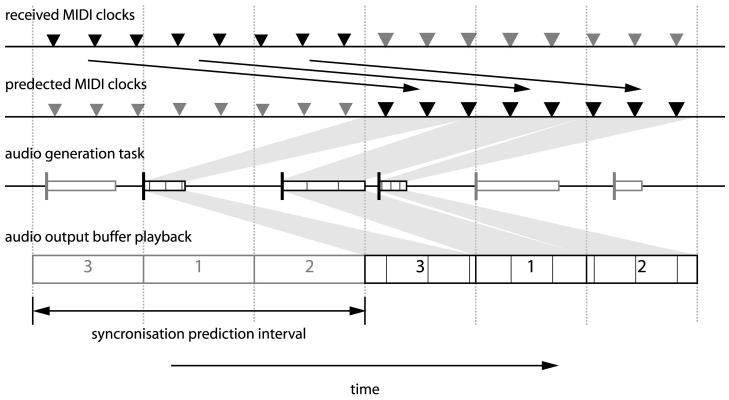

Figure 5.1 Generation of audio events synchronised to a time base provided by incoming MIDI clocks

Assuming that the received MIDI phase and rate are known for some recent time point, the MIDI phase at the beginning of an audio buffer may be predicted by linear projection of the phase from the known time point as follows:

```
predictedMIDIPhaseClocks =
knownPhaseClocks +
(bufferStartTimeSeconds –
knownTimeSeconds)
    * knownRateClocksPerSecond
```

In practice, the above calculation is not sufficient. Potential variations in MIDI tempo and the inherent timing instability of MIDI mean that it is unlikely that MIDI phase predictions will be totally aligned from one buffer to the next. As a consequence, calculating the MIDI clock phase independently for each buffer may introduce (possibly backwards jumping) discontinuities. Therefore both recently received clocks and the MIDI clock phase used for previously generated audio buffers should be taken into account. The MIDI clock phase should be smoothly modulated to avoid discontinuities.

Implementing the MIDI clock synchronisation technique described above relies on the audio generation task having access to buffer start times and sample rate, and to MIDI clock events timestamped using the same time base as used for the buffer start times. The way in which PortAudio provides this information is described in section 6.

The problem of synchronising generated audio to an external clock may be restated as implementing a phase-locked loop with a delay element in the feedback path as shown in figure 5.2. The presence of the delay element makes the system prone to overshoot and ringing. If latency can be tolerated, or if the external clock source can send clocks early, it may be preferable to use a reactive method based on the techniques described in section 4 rather than the predictive approach described in this section.

6 PortAudio Features Supporting Synchronisation

The implementation techniques described in the previous three sections each require specific timing information. All of the techniques require buffer start times and the sample rate. The MIDI triggered audio scenario requires the event response latency. The display scenario requires the current time, and the MIDI triggered event and MIDI clock synchronisation scenarios require timestamped MIDI events. The interface for accessing this information that has been proposed for the PortAudio V19 API is described below.

PortAudio represents time measured in seconds using a double-precision floating-point data type. The double data type was chosen after considerable deliberation because it provides sufficient resolution to represent time with high-precision, may be manipulated using numerical operators, and is a standard part of the C and C++ languages. Other representations with sufficient precision that were considered either used compound data structures which couldn't be manipulated numerically, or were only available as platform specific language extensions.

```
typedef double PaTime;
```

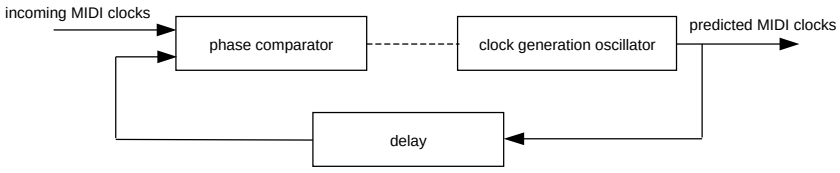

Figure 5.2 A phase-locked loop with a delay element in the feedback path

The playback time of each buffer is provided to the audio generation callback through a pointer to a `PaStreamCallbackTimeInfo` structure. The `outputBufferDacTime` field contains the output time of the first sample in the current buffer.

```
typedef struct PaStreamCallbackTimeInfo{
    PaTime inputBufferAdcTime;
    PaTime currentTime;
    PaTime outputBufferDacTime;
} PaStreamCallbackTimeInfo;
```

A stream's sample rate and latency may be retrieved using the `Pa_GetStreamInfo()` function which returns a pointer to a `PaStreamInfo` structure:

```
typedef struct PaStreamInfo{
    int structVersion;
    PaTime inputLatency;
    PaTime outputLatency;
    double sampleRate;
} PaStreamInfo;
```

The current time is made available on a per-stream basis through the `Pa_GetStreamTime()` function. This function could be used directly to timestamp incoming MIDI events, or to compute an offset between the operating system MIDI timestamp time base and the stream time base. Note that this function has completely different behaviour to the similarly named `Pa_StreamTime()` function that was present in the V18 API.

7 Implementation Issues

This section discusses some of the issues that have been encountered while implementing the PortAudio features described in the previous section under Microsoft Windows. Although some of these issues may be unique to Windows, it is expected that similar issues will arise on other platforms and that it may be necessary to develop common infrastructure to address them.

7.1 Sample Rate(s)

A sample rate is typically specified when opening an audio stream, for example the CD sample rate 44100Hz. Most host audio APIs, with the notable exception of OSS®,[6] treat this as a nominal request without regard for the actual sample rate of the audio hardware. In practice it is not uncommon to observe significant deviation from the nominal sample rate. Table 7.1 shows measured sample rates for a selection of consumer and professional sound cards running at a nominal sample rate of 44100 Hz. The measurements were made by averaging the data rate over a ten minute period. Although this method does not measure the actual sample rate, but rather the rate relative to the system clock, it is a useful measurement for the scenarios described in this paper because they depend on the relationship between system clock time and sound card clock. It is interesting to note that studies which compared sound card sample rate differences using only sound card clocks as time references produced similar results.[7, 8]

sound card or audio chipset	**host API**	**measured sample rate (Hz)**
Analog Devices SoundMAX	MME	44099.7
Avance AC'97	MME	44105.4
Intel 810	MME	44102.0
RME Digi96	ASIO	44098.8
SB AWE 64 Gold	MME	44106.1
SB Live #1	MME (kx drivers)	44110.4
SB Live #2	MME	44096.8
SB Live #3	DirectSound	44107.5
Turtle Beach Montego A3D	MME	44092.0
M-Audio Audiophile 2496	MME	44098.8
VIA AC'97 #1	MME	44099.1
VIA AC'97 #2	DirectSound	44097.8

Table 7.1 Sample rate measurements from a sample of PC soundcards running at a nominal rate of 44100 Hz

These measurements should not be interpreted as reflecting the precision of particular soundcard brands, but rather as supporting the general claim that observed sample rates deviate from nominal rates enough to impact the syncronisation applications described in this paper. If the nominal sample rate is used for synchronisation calculations significant errors may result. This is one reason PortAudio provides an actual sample rate in the `PaStreamInfo` structure. Measuring the actual sample rate accurately

requires accurate buffer timestamps or, at a minimum, access to an accurate high-resolution system clock. As described below this is not an easy requirement to fulfil.

7.2 One Shared Time-base (please)

One of the reasons PortAudio provides a separate `GetStreamTime()` function for each stream, rather than a single global `GetTime()` function is that PortAudio supports multiple host APIs on some platforms, and some of these APIs use different time bases internally. For example, the Windows multimedia timer is specified as the default time base for ASIO®,[9] however this timer only has millisecond accuracy at best. PortAudio chooses to use the more accurate Performance Counter timer for its MME implementation, however the Performance Counter has a documented 'feature' whereby it may jump forward unpredictably.[10] Another candidate for high-resolution timing under Windows is the Pentium 'time stamp counter', however this counter measures time in clocks, so it is necessary to use one of the other (inaccurate or unreliable) time sources to calibrate a conversion into seconds.

The Windows MIDI API provides timestamps for all received MIDI messages. However, the time base for these timestamps is unspecified, so that synchronising them with another time base is difficult. The simplest solution is to discard the driver supplied timestamps and re-stamp the events upon reception with times returned by the `GetStreamTime()` function. This solution is not totally satisfactory as the original timestamps are likely to be more accurate. A better solution would be to employ an algorithm that can measure the offset and skew between the MIDI driver time base and the PortAudio stream time base and use these measurements to convert between MIDI driver time and PortAudio stream time.

7.3 Buffer Playback Times

The accuracy of buffer start time information varies depending on the audio hardware and drivers in use. Some host APIs such as ASIO and Apple's CoreAudio[11] directly provide this information to PortAudio, while others do not – leading to additional implementation effort in the PortAudio library. Sometimes timing information is provided with limited resolution, for example the default ASIO buffer timestamps generated using the multimedia timer have a best-case resolution of one millisecond – this is significantly worse than the accuracy necessary for implementing sample-level synchronisation. The buffer playback times may also contain significant jitter noise depending on the implementation quality of the driver.

8 Future Work

At the time of writing, the features described in section 6 have been implemented in the PortAudio V19 implementations for the MME, DirectSound, ASIO and ALSA host APIs, with the exception of the sample rate feature, which is currently hard-wired to the nominal sample rate. Implementing code to supply the actual sample rate of running streams and to improve the accuracy of buffer play back time stamps by filtering operating system induced jitter and quantization noise are currently a priority.

Developing techniques and code for resolving multiple independent time bases such as the various Windows system clocks described in section 7.3 would be very useful. This would allow PortAudio to provide higher-resolution timing information to the computations described in this paper. Similar techniques could be useful for synchronising driver supplied MIDI event timestamps with a PortAudio stream time base. A substantial body of literature concerning clock synchronisation algorithms is available to assist with this task,[12,13,14] and the problem of synchronising multiple time bases on a single machine has been studied elsewhere.[15]

9 Conclusion

This paper has presented a conceptual framework for analysing synchronisation scenarios in fixed buffer size multiple-buffer real-time audio systems. Three different synchronisation scenarios were presented: synchronising a graphical display with generated audio, low jitter generation of audio events triggered from MIDI, and synchronising generated audio to incoming MIDI clocks. The requirements of these scenarios were established and the features of the PortAudio API that support these scenarios were presented. Some of the difficulties that arose while implementing these features on Microsoft Windows were discussed. It is hoped that this paper can provide a starting point for PortAudio users and others faced with the task of implementing media synchronisation in their applications.

10 Acknowledgements

Thanks to Phil Burk for contributing valuable suggestions during the development of this paper, and for his tireless work as master of the PortAudio code base; to John Lazzaro for introducing me to Matthew Delco's research; to Dean Walliss and Merlijn Blaauw for reviewing drafts of this paper; and to the members

of the #musicdsp IRC channel for contributing the sample rate measurements listed in section 7.

References

1. Bencina, Ross and Burk, Phil, “PortAudio - an Open Source Cross Platform Audio API,” *Proceedings of the 2001 International Computer Music Conference*, Havana Cuba, September 2001. pp. 263-266.
2. Bencina, Ross ed. “Proposed Changes to PortAudio,” available online at http://www.portaudio.com/docs/proposals/, May 2003.
3. Musical Instrument Digital Interface (MIDI) Specification 1.0, The International MIDI Association (IMA), Sun Valley California, August 1983.
4. Hassanzadegan, Hooman and Sarshar, Nima, “A New Method for Clock Recovery in MPEG Decoders,” *Second Online Symposium for Electronics Engineers (OSEE)*, Bedford Massachusetts, January 2002. pp. 1-8.
5. Kim, Kyeong Soo and Lee, Byeong Gi, “KALP: A Kalman Filter-Based Adaptive Clock Method with Low-Pass Prefiltering for Packet Networks Use,” *IEEE Transactions on Communication*, 48(7), July 2000. pp. 1217-1225.
6. Tranter, Jeff and Savolainen, Hannu, Open Sound System Programmer's Guide 1.11, 4Front Technologies, available online at http://www.opensound.com/pguide/oss.pdf, Culver City California, November 2000.
7. Fober, Dominique, “Audio Cards Clock Skew Compensation over a Local Network,” Computer Music Research Lab., Technical Report 020401, GRAME Research Laboratory, Lyon France, April 2002..
8. Kouvelas, Isidor and Hardman, Vicky, “Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool,” *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim California, January 1997. pp. 235-242.
9. ASIO Interface Specification v 2.0, ASIO 2.0 Audio Streaming Input Output Development Kit, Documentation Release #1, Steinberg Soft- und Hardware GmbH, Hamburg Germany, July 1999.
10. PRB: Performance Counter Value May Unexpectedly Leap Forward, Microsoft Knowledge Base Article #274323, Microsoft Corporation, available online at http://support.microsoft.com, Redmond Washington, May 2003.
11 Audio and MIDI on Mac OS X, Preliminary Documentation, Apple Computer, Inc., available online at http://developer.apple.com/audio/pdf/coreaudio.pdf, Cupertino California, May 2001.
12. Brandt, Eli and Dannenberg, Roger B., “Time in Distributed Real-Time Systems,” *Proceedings of the 1999 International Computer Music Conference*, Beijing China, October 1999. pp. 523-526.
13. Anceaume, Emmanuelle and Puaut, Isabelle, “A Taxonomy of Clock Synchronization Algorithms,” Internal publication #1103, Institut de Recherche en Informatique et Systemes Aleatoires, Rennes France, July 1998.
14. Anceaume, Emmanuelle and Puaut, Isabelle, “Performance Evaluation of Clock Synchronization Algorithms,” Research report #3526, Institut National de Recherche en Informatique et en Automatique, Rennes France, October 1998.
15. Delco, Matthew R. “Production Quality Internet Television,” Berkeley Multimedia Research Center, Technical Report 161, University of California Berkeley, August 2001.