# A new audio device driver abstraction

**Dickon Reed**
**Computer Laboratory,**
**University of Cambridge,**
**Cambridge CB2 3QG, U.K**
**Dickon.Reed@cl.cam.ac.uk**

## Abstract

*Traditional audio device drivers are unable to provide low latency audio services and often do not provide audio mixing services. High quality audio has, compared to other media types, relatively low bandwidth, but is very sensitive to latency and errors. This paper presents a device driver abstraction that exploits the low bandwidth and constant bit rate of audio to provide multiple channels of low latency audio efficiently. It provides applications with a great deal of flexibility with regards to latency, blocking behaviour and mixing.*

## 1 Motivation

Until recently, those operating systems that provided support for audio tended to do so in a rather inflexible manner. Typically, only one client could use the audio service at once and that client was often given little information about the timeliness of the audio delivery. Audio is important in modern multimedia systems; there may be several applications wishing to play audio simultaneously – an MPEG audio decoder playing music, a game, user interface sound effects and a video conference, for instance. Similarly, multiple clients may wish to record; a video conference, speech recognition for the user interface and a volume meter, for example. Some of these applications, such as most computer games, play several effects at once and so have to perform their own mixing on the various audio channels they play.

Some applications are very sensitive to latency. Holding a conversation with audio round trip time of greater than approximately 100ms is distracting. The difference of an attack of 10ms and 50ms is perceptible in music. The delay between a sound off to one side of a human head reaching one ear and reaching the other is about 1ms, yet the brain takes this in to account when locating sound sources. General purpose operating systems are typically unable to handle audio with low latency and jitter and so are not usually used for latency critical audio applications such as musical signal processing in real time.

When an application is processing an audio stream from one audio device with one clock source and playing it on one with another clock source, as will typically be the case in an audio conference, we have observed that the two clocks are not generally running at exactly the same rate. Even on identical hardware, operating temperature differences can have a significant effect. So, if the application has a buffer between the two sources then that buffer will tend to either overflow or underflow. There is no portable way for an application to query the exact playback speed or playout queue length, so there is no portable way of correcting clock rate drift.

## 2 Approaches to audio device drivers

One of the better documented kernel audio device driver schemes is Sun's [2] which has become a *de facto* standard for Unix. It treats the audio device as a number of special files, which makes it easy to program to but enforces certain restrictions; there will typically only ever be one process which has the audio device open at a time. Furthermore, on audio playback, the programmer is typically unaware of how much audio data has been placed in the file already, and so is unaware of exactly when any particular sample is played. Since Unix programmers are contractually unaware of the passage of real time anyway, this is not too much of an issue in the kind of applications written for Sun's audio system.

Early Microsoft operating systems such as MS-DOS typically did not provide any support for audio, leaving it up to applications to drive sound cards themselves. Therefore, various ad-hoc approaches were taken. These were generally acceptable, but necessarily suffered from the limitations of not using an operating system service. At first, this meant that configuration of the audio device had to be handled per program using audio. Later, as the Windows en-

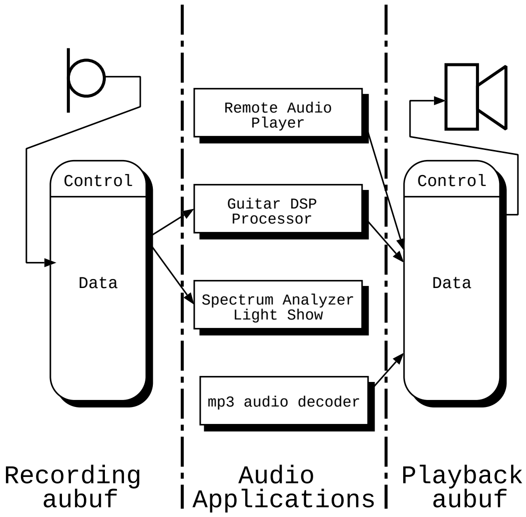


**Figure 1.** ***aubufs*** **in action**

vironment was bolted on top of MS-DOS and multitasking was retrofitted, it became clear that operating system support was necessary. Microsoft's first attempts at this were largely ignored by the programmers who had been writing MS-DOS programs with sound support; they still preferred to stay with third party sound driver libraries. Lately, with DirectX 5.0 [1], Microsoft have provided an abstraction that can be used for most applications. The DirectSound abstraction provides support for mixing, in some cases with hardware assistance. Applications may each have multiple channels of audio played simultaneously. Most PC games recently published use this architecture. It does not address latency issues very precisely, however, nor does it appear to provide any indication of the exact play-out rate of the device.

Most systems research dealing with audio have employed a fragment based scheme, sometimes providing direct DMA buffer access. Much work has been performed on higher level synchronisation issues in distributed systems ([6, 5, 3, 7]) which could be applied on top of the abstraction described below.

## 3 Shared Audio Buffers

Audio buffers, or *aubufs* have one master client and many slave clients, and shared between all of them is one control record belonging to the master but readable by the slaves and one data record readable and writable by all clients. The control record consists of a timestamp, a count and a description of the format of the data stored in the data record. The master reads or writes from the buffer continuously, touching the timestamp and advancing the counter in the control record as it does so. The control block also contains the time at which the master began reading or writing.

For playback, an aubuf works in *reader master mode*; the server reads from the buffer and maintains the control record. Clients use the contents of the control record to calculate where the master is currently in the buffer, then perform scaled add operations to the buffer just after that point. The reader zeroes the buffer as it reads it. For recording, an aubuf works in *writer master mode* in a similar fashion; the server now writes to buffer and still maintains the control record. Clients still use the control record to calculate the position of the master, then read from the data record just before that point.

Slaves do not allow themselves to work more than a certain distance away from the point where they have calculated the master to be at work. The clients are free to use the thread that was playing audio for other purposes, or to block the thread until, for instance, they calculate that half of the data they have placed in the buffer has been used. They then can top up the buffer as they see fit. This provides the back-pressure mechanism, without enforcing a blocking policy on applications. The distance that clients work ahead is equivalent to their latency for that audio device and so some applications will wish to minimise the distance. The master never waits for the slaves; if the slaves are not given enough CPU time to operate on the buffer their data will be lost. This scheme therefore requires an operating system capable of giving sufficient time to applications regularly. Hard real time systems should provide this, but more usefully general purpose soft real-time systems providing quality of service guarantees on CPU time should also be applicable, such as Nemesis [4].

The aubuf start up time and counter are used to measure the rate at which the master is operating on the buffer. In cases where audio is being recorded and replayed in real time using audio devices with different clocks, this can be used to provide data for sample rate micro-adjustment, so buffer sizes can be kept under control.

## 4 Implementation on Nemesis

The above scheme has been implemented on the Nemesis OS running on conventional PCs with Creative Labs Sound Blaster 16 cards, so that timely CPU guarantees can be made for audio applications. Originally, a fragment (i.e. packet) based driver based on the Linux kernel source was used on Nemesis, but the latency was unacceptably high.

Under the new scheme, for playback, the sound card is programmed to repeatedly play CD quality stereo audio from a DMA buffer, issuing an interrupt each time it loops back to the start of the buffer. The device driver has a thread that clears parts of the buffer after the Sound Blaster has played it, because there is no hardware facility to do this. On each interrupt, the aubuf control record's timestamp and counter are updated. The device driver calculates the observed sample playback rate of the sound card and stores this in the aubuf control record.

When a client wishes to play audio, it reads the aubuf control record. The current position of the audio device in the playback buffer is then calculated by subtracting the aubuf timestamp from the current system time and converting that time into a number of samples using the exact play out rate data stored in the aubuf. The maximum of the sound hardware packet size and the number of samples played in the scheduling period of the client is then added on to the calculated position, giving a position in the buffer the client can safely start writing to. Nemesis is able to guarantee that the applications will be given time to do their processing regularly, independent of whatever else is happening on the system. Note that some clients can be configured to run with higher latency and then can be scheduled less stringently without breaking up.

Clients are allowed to write directly into the data buffer of the aubuf. They perform only addition operations (which are atomic on x86 machines) on the buffer. A number of protection mechanisms could be deployed, such as the secure opcode facility provided on Nemesis for framebuffer access. However, the extra expense would not be worthwhile, because clients could still damage audio playback by placing malicious data in to the buffer. Note that the clients cannot crash the server or the machine by writing bad data in the buffer, so system stability is not affected.

Recording is easier to implement. Any number of clients can easily read from the same aubuf data record, using the aubuf control record to calculate where to start reading from. The sound card runs in a mode whereby it repeatedly records into a buffer, issuing an interrupt that records a timestamp and increments a counter when it passes the top of the buffer, much as before. There are no protection issues with recording, however, as it is not necessary for the clients to have write access to anything. Multiple aubufs may be used by the same application because the aubuf abstraction never requires a client to block.

Applications typically use a library providing implementations of two client-side playing objects– a synchronous and an asynchronous version. The synchronous version never works ahead of the sound card by more than the total size of the DMA buffer. A thread is used by the asynchronous version to perform the actual mixing in to the buffer; the client invokes operations on a proxy object that adds audio data to the potentially long inter-thread queue. The asynchronous version involves context switching between threads, but simplifies construction of some applications. The interface exported by the objects is the same in both cases. However, a `Put` operation is provided to add data to the buffer, that may optionally block or only place whatever data will fit in to the buffer. Query operations are provided to query the play out rate of the master and the current queue length of the client.

## 5 Preliminary results

In order to measure the achieved latency, a test application writes directly to a PC parallel port in order to trigger a digital storage oscilloscope, then begins to play audio. The delay between the parallel port trigger and the wave form emerging from the sound card is then clearly visible on the oscilloscope and can be measured to about one part in 500. Given the order of magnitude of the latency results obtained, the delay in writing to the parallel port of less than one microsecond can be neglected.

Experiments were performed on a 150Mhz Pentium machine with 16 megabytes of 60ns EDO RAM and a motherboard audio Sound Blaster 16 compatible sound card. The first port of the Linux kernel 2.0 series sound driver to Nemesis achieved a latency of 8.82ms, with some break up, when configured to a fragment size of 64 bytes at CD quality. It was necessary to run with a fragment size of 1024 bytes to obtain reliable audio, with a latency of 35ms. In both cases, the driver was configured to queue 4 outstanding fragments. With other queue lengths, the latency varied as would be expected, but 35ms was the lowest latency that could be achieved reliably.

The same code running native on Linux, configured with a fragment size of 8192 bytes and 3 fragments queue length, was stable with a latency of 139ms. The device driver can be configured down to a fragment size of 16 bytes with 2 fragments queued, but was unstable (i.e. in excess of ten percent of fragments were corrupted) when configured below 100ms. There is probably room for optimisation of these results on Linux.

When testing the new device driver at 44100Hz, latency was 440$\mu$s. Figure 2 shows latency results for different sample periods. The Sound Blaster hardware contains a hardware FIFO of undocumented length, which these results tend to indicate has an effective length of 18 stereo samples. This seems to indicate a *bus latency* (i.e. delay between application sending and sound card receiving and queuing the data internally) of around 55$\mu$s- i.e. between 2 and 3 samples, but more accurate experimental work is necessary to validate this figure. If the application is set to play any closer to the position of the sound card than this, then the first sample or more is lost. This was observed by having the application produce a sine wave peaking at 25% of maximum volume, but with the first sample being a transient to 75% of maximum volume. The transient is clearly visible on an oscilloscope trace when it is present.

An early, preliminary test of an aubuf application receiving ATM AAL5 CD quality audio from a Fore Systems AVA 200 across two Fore Systems ASX 200 switches, with clock

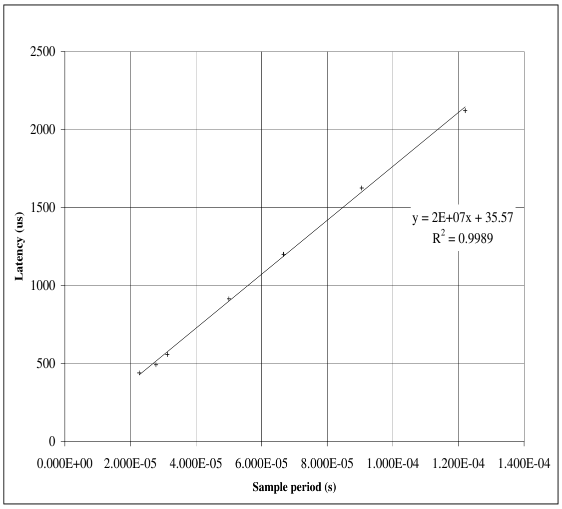


**Figure 2. aubuf driver latency**

| | Time ($\mu$s) |
|---:|---|
| Duration of one sample | 22.7 |
| Est. bus latency of aubuf driver | 36 |
| Est. latency of card FIFO at CD quality | 404 |
| Measured latency using aubufs | 440 |
| Audio across network to aubuf | 4300 |
| Linux driver on Nemesis | 35000 |
| Linux driver on Linux | 139000 |

**Table 1. Timings for CD quality audio**

rate matching, achieved a latency of 4.3ms from a distinctive waveform entering the AVA 200 to the waveform leaving the sound card. (This result was taken before optimisation and several system improvements). Table 1 summarises these work in progress results.

## 6 Future work

Work is currently in progress to examine achievable end to end latencies using this scheme in various forms of network audio. It is anticipated that very low latencies can be achieved at the expense of forcing the operating system to context switch to the very low latency application frequently. It will be interesting to measure both the effective latencies achievable and the impact in overall system performance of executing some low latency audio clients. Lip-sync between an audio stream and a video stream from the same source shall also be examined, as well as synchronisation between different audio streams (for instance for multiple device digital audio hard disk recording and playback). Experiments are also planned to measure latency in audio processing (i.e. recording and playback in the same system simultaneously) and the effect on latency of software signal processing in path. The device driver and client library could be ported to other systems providing accurate timing and CPU guarantees and to other kinds of sound card. More work on measuring and detecting and correcting clock drift is planned. Aubufs may be useful for internal purposes within applications- they may prove a useful in a software synthesiser, for instance. Finally, this device driver abstraction suggests a different design for sound cards, which should mean that protection, mixing and timing can be provided in hardware. For such a piece of hardware, the device driver would only be responsible for initialisation and channel creation.

## 7 Conclusion

Several clients have been ported to use playback aubufs, including an MPEG audio layer 3 decoder, a software MIDI synthesiser, a computer game, a guitar software digital signal processor and an ATM network audio player. Of these, synthesiser, game and signal processor benefitted from using aubufs particularly, by eliminating a stage of software mixing and a copy they previously had to perform. Aubufs provide a useful abstraction for flexible, efficient and sub-sample accurate audio processing for operating systems where CPU guarantees can be made to applications. They provide low latency audio services and are well suited to flexible recording as well as playback on standard hardware, because audio packetisation delays are eliminated.

## References

[1] Microsoft Developer Network; Direct X 5 Software Development Kit.
[2] SunOS 5.5.1 `audio` man page, section 7.
[3] G. Coulson and G. S. Blair. Meeting the real-time synchronisation requirements of multimedia in open distributed processing. *Distributed Systems Engineering Journal*, 1994.
[4] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.
[5] T. D. C. Little and A. Ghafoor. Synchronisation properties and storage models for multimedia objects. *IEEE Journal on Selected Areas In Communications*, 8(3):413–427, April 1990.
[6] C. Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas In Communications*, 8(3):391–400, April 1990.
[7] M. Papathomas, G. S. Blair, G. Coulson, and P. Robin. Addressing the real-time synchronisation requiements of multimedia in an object-orientated framework. *Multimedia Computing and Networking*, 1995.