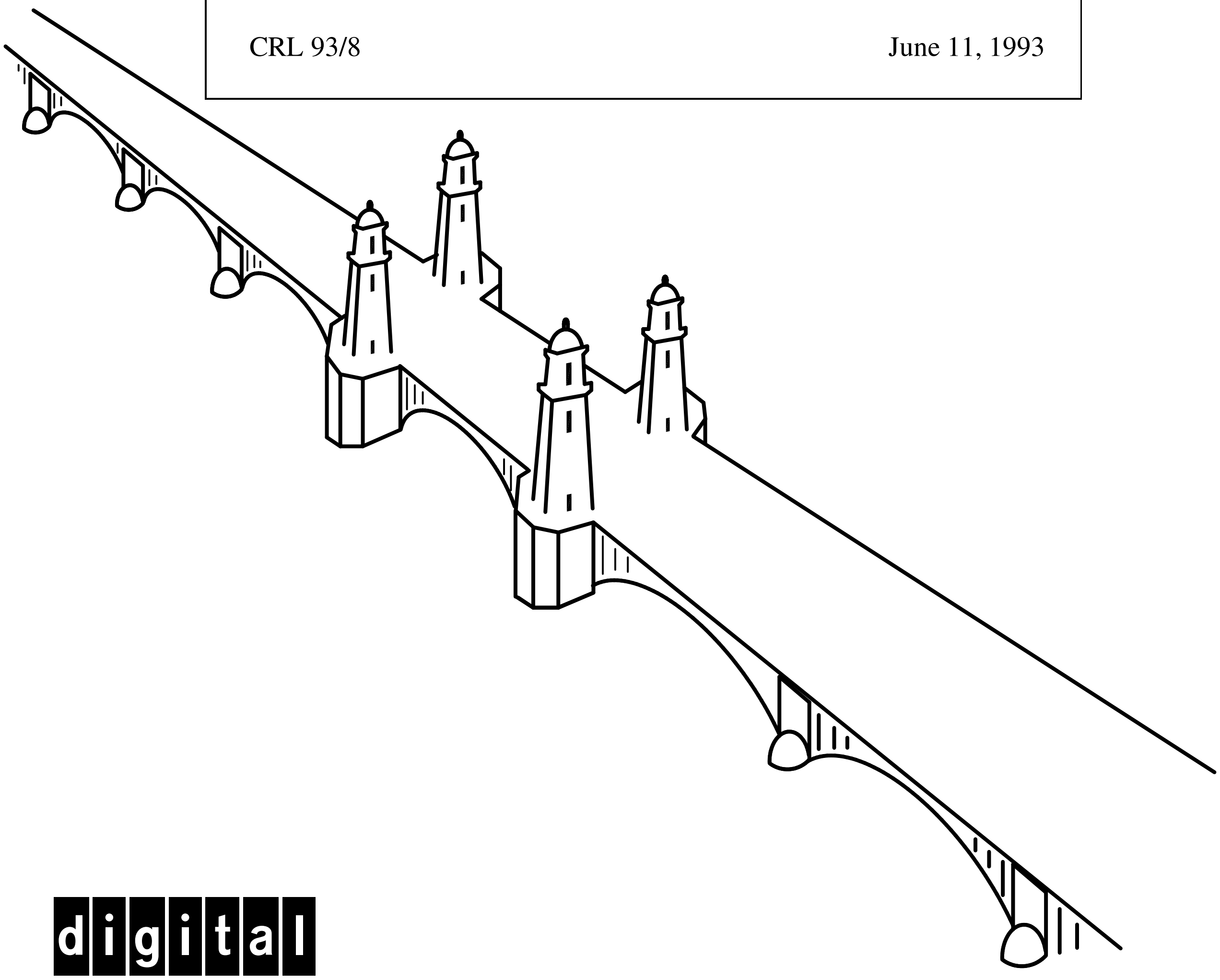
AudioFile: A Network-Transparent System for Distributed Audio Applications

Thomas M. Levergood, Andrew C. Payne,
James Gettys, G. Winfield Treese, and Lawrence C. Stewart

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/8 June 11, 1993



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

AudioFile: A Network-Transparent System for Distributed Audio Applications

Thomas M. Levergood, Andrew C. Payne,
James Gettys, G. Winfield Treese[1], and Lawrence C. Stewart[2]

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/8 June 11, 1993

Abstract

AudioFile is a portable, device-independent, network-transparent system for computer audio systems. Similar to the X Window System, it provides an abstract interface with a simple network protocol to support a variety of audio hardware and multiple simultaneous clients. This report describes our approach to digital audio, the AudioFile protocol, the client library, the audio server, and some client applications. It also discusses the performance of the system and our experience with using standard network protocols for audio. A source code distribution is available for anonymous FTP.

Keywords: audio, client-server, multimedia, speech

[1] Also with MIT Laboratory for Computer Science.

[2] The authors are listed in random order.

An overview of the material in this report will appear in the *Proceedings of the Summer 1993 USENIX Conference*.

Contents

1 Introduction

Audio hardware is becoming increasingly common on desktop computers, such as workstations, PCs, and Macintoshes. In 1990, the authors began a project at Digital's Cambridge Research Laboratory to explore desktop audio. [1] One of us (Levergood) designed a TURBOchannel [2] I/O module called LoFi, with capabilities for telephony and both low and high-fidelity audio. Once that hardware was available, we began work on software. The result of our efforts is the AudioFile System.

It was clear from the outset that audio on the desktop should have the same flexibility that users have come to expect of the display. Similar to the X Window System[14], AudioFile was designed to allow multiple clients, to support a variety of underlying hardware, and to permit transparent access through the network. Since its original implementation, AudioFile has been used for a variety of applications and experiments with desktop audio. These applications include audio recording, playback, video teleconferencing, answering machines, voice mail, telephone control, speech recognition, and speech synthesis. AudioFile supports multiple audio data types and sample rates, from 8 KHz telephone quality through 48 KHz high-fidelity stereo.

Currently, AudioFile runs on Digital's RISC DECstations under ULTRIX, Digital's Alpha AXP systems under DEC OSF/1 for Alpha AXP, Sun SPARC systems under SunOS, and Silicon Graphics Indigo workstations under IRIX. A source code distribution is available by anonymous FTP over the Internet.

Like the X Window System, AudioFile has four main components:

- The Protocol. The AudioFile System defines a wire protocol that links the server with client applications over a variety of local and network communication channels. The semantics of the protocol commands and responses define what servers are expected to do and what services clients can expect.
- Client Library and API. The AudioFile client library and applications programming interface (API) provide a means for applications to generate protocol requests and to communicate with the server using a procedural instead of a message-passing interface.
- The Server. The AudioFile server contains all code specific to individual

[1] And video, but that is another story.

[2] TURBOchannel is the I/O bus used on Digital's DECstation and Alpha AXP workstations.

devices and operating systems. It mediates access to audio hardware devices and exports the device-independent interface to clients.

- Clients. The AudioFile distribution includes several out-of-the-box applications which make the system immediately usable and which serve as illustrations for more complex applications.

This report begins with a discussion of the design goals and the fundamental principles of AudioFile. We then place this work into the historical context of earlier desktop audio efforts and other more recent audio work. Following a discussion of some of the hardware used for desktop audio, we discuss the network protocol, the client library, and the server implementation in some detail. Then we describe a sampling of applications that use AudioFile, followed by an analysis of AudioFile's performance. We conclude with a brief discussion of plans for future work and explain how to get the software.

1.1 Design Goals

AudioFile was designed with several goals in mind. These include:

- Network transparency. Applications can run on machines scattered throughout the network. This property is desirable for several reasons: an application may be licensed to run only on a specific machine, or it may require computing resources not available on every desktop. Network transparency allows such constrained applications to run anywhere but still interact with the user. Another quite different benefit of network transparency is that it enables applications to use audio on several systems at once. Teleconferencing is such an application; it must communicate with multiple audio servers.

- Device-independence. Applications need not be rewritten to run on new audio hardware. The AudioFile System provides a common abstract interface to the real hardware, insulating applications from the messy details. Furthermore, as we will describe in Section 8, some applications can operate on generic audio, without worrying about details such as sampling rate, number of channels, or encoding.

- Support for multiple simultaneous clients. Applications can run concurrently, sharing access to the actual audio hardware. Two audio applications running on a single computer should behave just like those same applications running on separate computers in the same room. [3]

- Support a wide range of clients. It should be possible to implement applications ranging from audio biff [4] to multiuser teleconferencing. We have chosen to implement a few very general-purpose mechanisms that permit a wide variety of applications, including both aggressively real-time applications and those which are more easygoing.
- Simplicity. Adding simple audio to an application should be easy. Simple play and record clients should require very little code. Complex applications should be possible, but one should not burden simple clients with massive mechanism.
- Quick time to implement. We wanted to start building applications quickly. We chose to leverage as much existing mechanism as we could, and we tried to put as little as possible into the operating system kernel. Although debugging kernel device drivers is possible, it is neither a rewarding nor a time-efficient process.

1.2 Fundamental Principles

In addition to our goals, we designed AudioFile with a few fundamental principles in mind. These include:

- Computers are fast. Modern machines are fast enough to handle a variety of signal processing and real-time problems. There is no need to be frightened by a requirement that the computer look at every audio sample. Unnecessary copies of data are to be avoided, but it is frequently better to copy data than to corrupt the structure of an application. We assume an ADC/DAC model for the audio hardware device and do not depend on intelligent controllers.
- Client control of time. Clients are responsible for specifying the exact timing of recording and playback. This puts a minor additional load on the clients, but greatly simplifies the server and makes the AudioFile System capable of handling applications requiring a wide range of real-time behavior. Explicit

[3] We do think that there is room for something like an "audio window manager" which would impose a policy on multiple applications, but so far we have not found it necessary to implement one. In any case, we think the core system should "provide mechanism, not policy".

[4] biff is a Berkeley UNIX program that notifies a user when new mail arrives.

control of time also makes it very easy to construct applications requiring synchronization of multiple activities.

- No rocket science. AudioFile does not require specialized low-level network protocols or multithreaded environments. These facilities were not necessary to achieve our goals and using special protocols or threads would impede the portability of the system. Consequently, the AudioFile server is single threaded, we use standard TCP/IP, and no operating system support more complex than the select() system call is required. [5]

- Simple applications should be simple. Complicated applications should be possible. In other words, simple applications should not have to pay a price in complexity when they need only simple functionality. [6]

1.3 Implementation

The parts of the implementation of AudioFile that are not specific to audio, such as client/server communications, are based on X11 Release 4. The code was freely available and provided a well-understood communications infrastructure. [7] While we considered several languages for the implementation of AudioFile, starting with the MIT X11R4 source code tipped the scales in favor of the C language.

Of course, traveling this route caused us to carry extra baggage. For example, the original MIT source code is written with C preprocessor commands, mostly used by the client library, that conditionally build code with or without function prototypes. At least one well known vendor of big-endian computers does not support function prototypes with their stock C compiler. Function prototypes have proven to be quite useful for developing large portable systems in C. Unfortunately, "portable" sometimes means "lowest common denominator" — so our code is also cluttered with left-justified chicken scratches [19]. [8]

We should emphasize the fact that AudioFile is *not* an addition to an X Window System server. The AudioFile server is a separate entity which borrowed some

[5] It seems likely that a multithreaded server would permit a slightly cleaner implementation in the server, but we felt the performance and portability risks did not justify it.

[6] This is really important — notice that simplicity is both a goal and an abiding principle. Our thinking is that if one gets the core functionality right, then an explosion of complexity can be avoided.

[7] Why start from a clean sheet of paper? For more information on how to steal code, consult Spencer[15].

[8] The next AudioFile release will *require* a compiler that supports function prototypes.

common source code to build the implementation. Contrary to others, we believe that audio services should be separate from graphics.

2 Audio Abstractions

This section describes the fundamental abstractions used by AudioFile. These provide the view of audio available to clients and guided the design of the protocol, client library, and audio server. We model an audio device as an entity that produces and consumes sampled data at a regular rate known as the sampling frequency. The sample data is one of several predefined types and consists of one or more channels. The actual hardware is based on Analog to Digital (ADC) and Digital to Analog (DAC) converters. The important abstractions discussed here are time, the audio input and output models, and events.

2.1 Time

The concept of audio device time is critical to understanding the design of all AudioFile components. We expose audio device time in the protocol and at the client library API. It is also fundamental to the correct operation of the audio server. All audio recording and playback operations in the AudioFile System are tagged with time values that are directly associated with the relevant audio hardware.

This section discusses our decision to use device time, the time abstraction, and how to calculate with audio device times, then finishes with a brief discussion of alternative approaches.

In multimedia systems, exact timing is necessary to synchronize the different aspects of a presentation or to relate the occurrence of multiple events. AudioFile permits clients to express the precise timing of individual digital audio samples.

Why device time?

There are a remarkable number of clocks in a modern distributed computer system. A simple desktop system might have four different time sources: the time-of-day clock, the interval timer, the display refresh, and the audio clock. Each clock has its own uses: the time-of-day clock might be used to schedule overnight backup, the interval timer might be used to schedule program counter sampling for profiling a program, the display clock might be used to schedule cursor tracking, and the audio clock is used by the audio hardware to schedule the recording and playback of individual digital audio samples.

Each computer system in a network has its own clocks. There are network protocols, such as NTP[8], which keep the time-of-day clocks approximately synchronized, but no existing systems we are aware of keep interval timers, display, or audio clocks synchronized. Each of these many clocks has a nominal rate at which "ticks" occur, and if these rates were exact, then one could easily convert from time as shown by one clock to time as shown by another. Unfortunately, all these clocks vary from their nominal rates, and the exact rates are subject to change with the age of the equipment, temperature, and other environmental factors.

In principle, it is possible to use any clock for audio synchronization. However we wanted to be able to specify audio down to the individual sample, so we chose to use audio device time. When a server supports multiple audio devices, it traffics in device time for each device *separately*. AudioFile does not provide a complete infrastructure for synchronization; rather, it supplies low-level timing information to its clients. Client applications can build conversion mechanisms suitable to their own needs for synchronizing multimedia streams, relating events to the real world, or simultaneously communicating with multiple audio devices.

We envision adding standard client library and server mechanisms for synchronizing multiple clocks and for providing clock conversion services to clients, but we have not yet encountered a compelling need to do so.

The device time abstraction

The underlying implementation of the audio device clock is the oscillator that controls the hardware sample rate. This clock may be directly accessible to the audio server, or only indirectly as the running total of audio samples generated or accepted by the hardware. In either case, the server maintains a representation of the clock in a "time register" for scheduling all audio events for the particular device.

Converting time values

There is no absolute reference value for a device time; the value is set to 0 when the server is initialized and advances thereafter. This is in contrast to the usual computer method of handling time-of-day, in which the binary representation is something like the number of seconds since January 1, 1900. The AudioFile time can not be algorithmically converted to a calendar date and time.

One can establish a correspondence between two clocks. Given clocks A and B, and a pair of values (T_a, T_b) of the two clocks that occurred "at the same time," and the rates of advance of the two clocks R_a and R_b, then given a future value of clock A, say t_a, one can compute the corresponding value t_b according to clock B:

$$t_b = T_b + R_b * ((t_a - T_a)/R_a)$$

Although there is no *exact* relationship between device time and time measured by the system real-time clock or to time shown by the clock on the office wall, because the rates are not known to infinite precision, there is an *approximate* relationship which is good enough to permit reasonably accurate conversions between different clocks. The audio device sampling rate is used to move between time in sample ticks and time in seconds. For example, at 8 KHz, four seconds in the future maps to the current device time plus 32000 ticks. This is sufficiently accurate for most purposes, even though the exact sampling rate might be 7999.96 Hz rather than 8000.00.

Representation

Audio device time is represented by a 32-bit (finite length) unsigned integer that increments once per sample period and wraps on overflow. These time values are specific to a particular audio device.

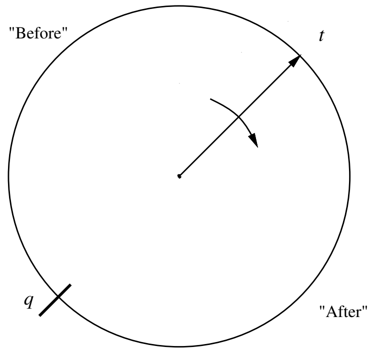


Figure 1: Circular representation of audio time

It is convenient to map this representation of time onto a circle, as shown in Figure 1. In this diagram, the time t is marked with the clock hand that sweeps out clockwise. The circumference of the circle is the range of the time counter, or 2^{32} samples.

Because the 32-bit numbers eventually wrap, one cannot simply compare two values to establish their ordering. Servers and clients often have to make comparisons between two times and decide their relative positions. This is done by dividing all possible time values into the equally sized past and future regions. The division point q (equal to $t + 2^{31}$) is marked on Figure 1. Any time from t clockwise to q is considered to be after t, and any time from q clockwise to t is considered to be before t.

Time comparisons are easy to implement. Given two time values, a and b, compute their 32-bit two's complement difference $a - b$. The most significant bit of the result gives the result of the comparison. If it is set, then b is in the future relative to a. Otherwise, b is in the past. This computation is easily made by casting the difference to a signed data type. The following example is for a device running at 8000 samples per second.

```
if ((int) (b - a) > 0)     /* time b is later than time a.           */
if ((int) (b - a) < 0)     /* time b is earlier than time a.         */
if ((int) (b - a) == 8000) /* time b is one second later than time a */
```

There is a problem, of course, when the difference approaches 2^{31}. A time in the distant past may suddenly switch over to the distant future as time advances. Programs that deal with time must be careful not to make comparisons between widely separated time values. However, this is usually not a problem since even at a 48 KHz sampling rate, 2^{31} samples represents about 12 hours of audio. At 8000 samples per second, this period is about 3 days.

Usually, an audio device supports both input and output. However, AudioFile supports only one time register for each audio device. This means that if the actual hardware uses different sampling rates for input and output, then the server will present distinct unidirectional audio devices for input and output.

Client use of explicit time

Each play and record request carries with it an exact timestamp. The implementation of this abstraction is accomplished by buffering future playback and recent record data in the server. Continuous recording or playback is accomplished by advancing the requested device time for a request by the duration of the previous request.

Explicit control of time provides the mechanism needed for real-time applications. As long as playback requests reach the server before their requested start times, playback will be continuous. A leisurely application will schedule playback for well in the future, while an aggressive real-time application will schedule playback requests for the very near future. The server will buffer requests up to four seconds in the future.[9] Both applications must supply audio at the same rate, but the real-

[9] We use four seconds to be concrete; the precise size of the server buffer is available to clients as

time application must assure a much lower variance in latency. For example, if an application is scheduling audio for one second in the future, any individual block can be delayed for up to one second without disturbing the playback. In contrast, an application scheduling audio for 50 milliseconds in the future has to assure that blocks cannot be delayed for more than 50 milliseconds.

The fact that the server buffers audio for future playback also allows clients to schedule playback asynchronously. This permits single threaded clients to handle audio in addition to their other activities.

Recording is a much easier problem than playback. The server buffers all device input, typically for the past four seconds. Therefore, unless a record request fails to reach the server until four seconds after its requested start time, no data will be lost. Of course an application making real-time use of the record data must make record requests close enough to current time to satisfy its real-time constraints.

Because the server buffers all device input, clients can request recording at times "in the past" and deliver the appearance of instantaneous response. For example, consider an application that displays a "Record" button. There is some delay between the time that the user presses the button and when the record request reaches the audio device. By recording from the recent "past," the application can begin recording at the instant the button was hit. This is a more natural interaction than requiring the user to wait for some indication, such as a visual display or audible beep, that recording has begun.

Alternate designs

In contrast to AudioFile's design, the usual way to handle sequential data such as audio in computers is as a stream, just a sequence of values. The stream is a very simple abstraction, but for audio work, it fails three crucial tests:

- Streams do not permit synchronization. It is usually necessary to employ complex out-of-band mechanisms to find out how much data is buffered in a stream or to find out if a stream is running or blocked. Consequently, an application using a stream mechanism will have difficulty establishing the moment a particular sound will emerge from the loudspeaker or the moment a particular sample was recorded.

- Streams do not solve real-time problems. Streams tend to obscure issues of bandwidth and latency that are critical to real-time applications. The idea is good: the application merely writes audio data into the stream, and the

an attribute of the audio device.

sound will emerge from the speaker without gaps. In fact, the application must still keep up, but it has no way of telling how much margin there is.

- Streams do not deliver any additional simplicity to the application. In the abstract, audio is just an unending sequence of samples and might seem to be a good match for streams, but in practice, applications deal with the data in blocks anyway, so all streams do is force applications and services to continually create and discard the blocking information.

Instead of dealing with streams of audio samples, the fundamental operations in AudioFile are block-oriented, and specify exact times at which the blocks are to be recorded or played. [10]

2.2 Output Model

The output model we use for an audio device is shown in Figure 2. In general, clients can schedule playback at any time from the present to four seconds into the future. Playback data that falls in the past is silently discarded. Playback data that falls in the future is buffered unless it also falls beyond four seconds in the future. Playback requests that fall beyond the four-second buffer are suspended until time advances to within four seconds.

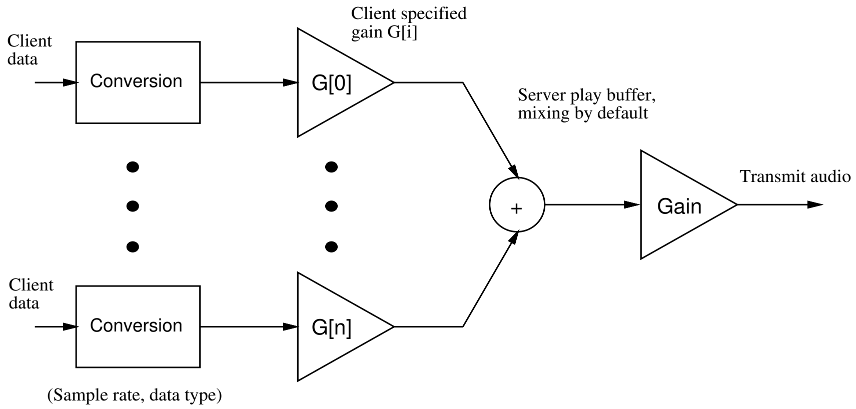


Figure 2: AudioFile server output model

After a playback request is received by the server several stages of processing take place. The sample data will be one of the several sample types supported by the

[10]We may have mentioned this before, but it is important.

abstract device, but possibly different than the data type supported by the actual hardware. The data is first passed through a conversion module that translates the data type received by the server to the data type supported by the audio hardware. Frequently, the client data type is the same as the audio hardware data type and this conversion stage is not necessary. The server support for conversion modules will also be used to handle compressed audio data types. In designing AudioFile we envisioned this module handling sample rate conversion as well, but the design for resampling is not complete. [11]

Once the data is in the form required by the audio hardware, it is adjusted by a client specified gain value before being mixed into a common server buffer. The client gain is stored in an audio context (see Section 7) and defaults to 0 dB. The mixed data stays in the server buffer until its scheduled playback time approaches. As the sample data is drawn from the output buffer and sent to the hardware, a final gain stage is applied. This master gain acts as a volume control for the mixed version of all client data sources. Frequently, this volume control is implemented by the audio hardware.

The server is responsible for ensuring that the samples in the output buffer are sent to the DAC at their corresponding values of the time register. Since time is exposed at the client library API, client applications are able to manipulate data at arbitrary sample boundaries within the server.

The output model specifies that silence is emitted during periods of time in which no client data has been written to the output buffer. This means that applications need not transport "silent" data from client to server. Instead, a client simply advances its playback time accross the silent interval before resuming playback. This mechanism can reduce network bandwidth requirements.

2.3 Input Model

The input model for an audio device is shown in Figure 3. Like the output model, the input model buffers four seconds of sample data. However, the input model is conceptually simpler since there are no dependencies between clients such as those introduced by the mixing stage for playback.

As shown in Figure 3, the recorded data is modified by an input gain which is often implemented by the hardware. The data is placed into a server buffer indexed by

[11] Actually, we still need to add the support of multiple audio sample data types by a single audio device . We were not able to implement this aspect of the AF2R2 design in time for the release of the kit.

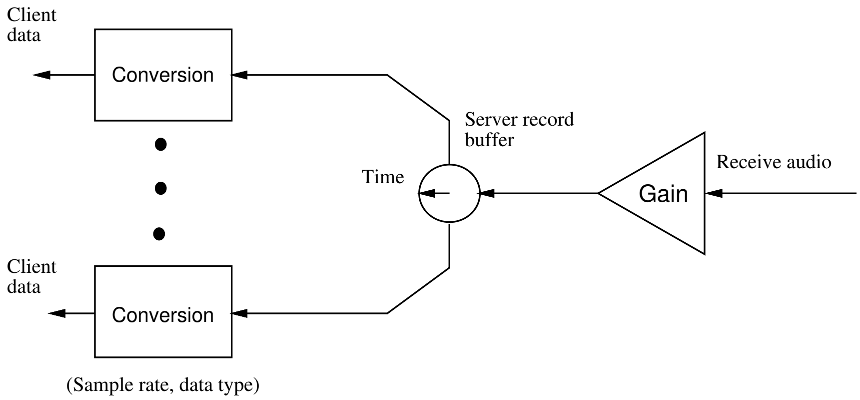


Figure 3: AudioFile server input model

the current value of time.

Clients requesting input data older than four seconds in the past are given silence. Record requests within the past four seconds return the buffered data. Record requests in the future cause the client connection to block until time advances far enough to service the request.

The system also supports a non-blocking record interface. If the client chooses not to block, the server will reply with as much data as it can supply immediately. With either blocking or non-blocking recording, the client application can record consecutive blocks by advancing the requested record time by the duration of the previous block.

The input model also supports a module which converts the native audio hardware data type to a client requested data type.

We did not include a per-client gain modification on recording since the data will always be replayed through a path where gain modification is supported.

2.4 Events

In a client/server system, the server usually waits for the client to ask for service, then responds. An event is the exception: an asynchronous message from server to client. Events may be caused by a device or as a side effect of some client's request. An event is never sent unless a client registers an interest in receiving notification. Clients can register for various classes of events such as a telephone device ringing or a change in a property used for inter-client communications. Protocol events are discussed in Section 5.2; the way the client library handles events is described in Section 6.1.

3 Background

Computer handling of digital audio is not new. The important historical points of reference are those of signal processing, telephone integration, device-independence, and network-transparency.

3.1 Signal Processing

Research groups have used analog-to-digital (A/D) and digital-to-analog (D/A) converters for computer recording and playback of speech and audio for many years. For the most part, this "data acquisition" has been accomplished with expensive and specialized hardware and software intended for the laboratory instrumentation market, rather than for general use. We would categorize these systems as device-dependent and standalone (not networked). However, essentially all existing technology for audio signal processing was developed this way.

3.2 Etherphone

In the early 1980's, the Xerox Palo Alto Research Center built a telephone system in which voice was transmitted over an Ethernet. This system was called Etherphone [16]. Besides its utility as a telephone system, the Etherphone system had capabilities for workstation recording and playback, voice storage, and it was certainly network transparent. Each workstation was associated with a nearby Etherphone, which was a dedicated computer directly connected to the office phone line, local audio devices, and the Ethernet. The Etherphone system was used primarily to explore issues of multimedia documents and telephone integration. Etherphone audio was entirely telephone-quality. In addition, because audio was passed directly from Etherphone to Etherphone, without intervention by more powerful computers, there was little opportunity for signal processing.

3.3 Firefly

In the mid 1980's, the Firefly multiprocessor workstation[18], developed at Digital's Systems Research Center, had simple telephone-quality audio. An audio server on the Firefly buffered the previous four seconds of recorded data and the next four seconds of playback data; it exported a simple remote procedure call (RPC) interface to applications. The Firefly audio system was primarily used for applications such as teleconferencing and multimedia presentations. The Firefly audio capability was primarily used for applications such as teleconferencing and multimedia presentations.

We would categorize this system as network-transparent, but it was still device-dependent. The Firefly audio system pioneered explicit client control of time.

3.4 VOX

In the mid to late 1980's, the MIT Media Lab and the Olivetti Research Lab in Palo Alto collaborated on a project called VOX[4]. VOX was an audio server based on a model in which essentially all audio related functions were included in the server, with the client mainly handling control those functions. The VOX server was responsible both for record and playback functions and for establishing direct connections between disparate devices.

VOX was partly constrained by a view that audio would be primarily sourced and sunk by external devices, possibly with direct connections between them. In addition, the view was that audio was such a real-time compute intensive data type that clients could not manage the load. Instead, all details of audio handling were subsumed into the server.

We would categorize this system as device-independent, but not network-transparent.

3.5 Related Work

Other projects similar to AudioFile were underway at about the same time.

XMedia Tools[3], a Digital product, was somewhat more ambitious than AudioFile, using a more complex protocol and putting more emphasis on implementing applications within the server. In contrast, AudioFile emphasizes simplicity of the protocol and the server, leaving more complicated actions to be performed by clients. As described in Section 10, our experience to date indicates that the resulting performance is quite good.

Terek and Pasquale at UCSD developed an audio conferencing system based on a modified X server [17]. In contrast, we chose not to incorporate audio into the X server. Our approach has several advantages: AudioFile is independent of X and can be used when X is not, server implementors need not understand the intricacies of the X server, and clients do not suffer because of the scheduling decisions that the X server makes in servicing graphics requests and input events. If synchronization between audio and graphics is necessary, it can be performed by the clients or by using the X synchronization extension.

Sonix[12] is a network-transparent sound server developed at Bellcore. It was also inspired by X and is similar to XMedia in design, with "patchcords" to internally connect audio devices or to bypass the Sonix server itself. AudioFile takes a different view of time and emphasizes doing work in clients, rather than manipulating the flow of audio data within the server. Sonix includes minimal support for synchronization.

4 Audio Hardware

This section describes the audio hardware currently supported by AudioFile.

4.1 LoFi

In 1990, as part of the Cambridge Research Lab's overall goals of exploring networked audio and video, one of us (Levergood) designed a TURBOchannel audio module called LoFi[7]. [12] Later, Digital's Multimedia Engineering organization released the design as the product DECaudio. The research "LoFi" and the product "DECaudio" are substantially identical; we will use "LoFi" to refer to this device in the rest of this document.

LoFi supports two 8 KHz telephone quality CODECs, one connecting to a telephone line and one connecting to local audio devices. LoFi also contains a Motorola 56001 DSP chip with 32K 24-bit words of memory shared with the host processor. The 56001 serial port supports a 44.1 KHz stereo DAC and can also be used with

[12] We called it LoFi primarily because it wasn't. LoFi always included high fidelity audio capability, but we chose the name to obscure this fact, because we knew we wouldn't get around to writing the HiFi support software for a while and we wanted to defuse expectations. The high fidelity support software was completed in 1992.

external DSP port devices including stereo A/D and D/A converters operating at sample rates up to 48 KHz.

The telephone interface on LoFi enables applications such as voice mail and remote information access. We see no difficulty in adding AudioFile support for other kinds of telephone interfaces, such as ISDN or PhoneStation [20].

4.2 JVideo

JVideo is a TURBOchannel module developed at Digital Equipment Corporation for experiments in desktop video. Like LoFi, JVideo has a Motorola 56001 DSP processor with shared memory, but JVideo also has stereo ADC and DAC hardware that is capable of variable sample rates. However, JVideo has neither telephony capability nor an external DSP port.

4.3 Integral Workstation Audio Devices

The Personal DECstation series, the Alpha AXP-based DEC 3000/300, 3000/400 and 3000/500, and the Sun SPARCstation-2 all include 8 KHz CODEC devices on their system modules.

4.4 LineServer

The LineServer is an Ethernet peripheral. It is a Motorola 68302 microcomputer system with 128K ROM and 64K RAM, an Ethernet controller, high speed V.35 serial line interface, and an 8 KHz ISDN CODEC. We use LineServer within Digital's research labs for remote Ethernet bridging and IP network routing over both dedicated digital circuits and dial-up ISDN circuits.

The LineServer version of the AudioFile server is interesting because the server runs on a nearby Ethernet host, not on the LineServer itself. The audio server exchanges low-level device-specific network messages with the LineServer.

4.5 SGI Indigo

Recently Guido van Rossum of CWI in the Netherlands contributed AudioFile device support for the Silicon Graphics Indigo workstation. The Indigo supports stereo audio at a variety of sample rates up to 48 KHz.

5 Protocol Description

The AudioFile protocol is designed on the same basic principles as the X Window System protocol. Control and audio data are multiplexed over a single byte-stream connection between client and server. A single connection can carry more than one audio stream. Multiple clients, potentially running on multiple machines of different architectures, can use the same server at the same time.

5.1 Data Transport

AudioFile is intended to be used over almost any transport protocol, though their behavior may affect real-time audio performance (see Section 11 for some performance analysis).

As in X, the AudioFile protocol presumes that the data transport between the client to the server is reliable and does not reorder or duplicate data. Any transport mechanism fulfilling these criteria could be used. AudioFile takes advantage of streaming when possible, though we believe this is less common than in the X case. The current version of AudioFile supports TCP/IP and UNIX-domain sockets.

We believe that the programming model AudioFile presents may reduce the number of audio applications needing low-latency communications, because clients can control exactly when audio will appear or when it was recorded.

5.2 Events

As in X, events have a fixed size. Only five event types are currently defined: four for telephone control and one for interclient communications. Some details of the telephone events are discussed below in Section 5.5.

All device events contain both the audio device time of the device and the clock time of the host of the server. The host clock time may be needed when synchronizing with other media on the same host (for example, video being displayed by the window system).

5.3 Protocol Requests

All protocol requests have a length field (16 bits, expressed in 32-bit quantities), an opcode (one byte), and an optional opcode extension (one byte). The shortest possible request is therefore four bytes long. The length field limits the longest request to 262144 bytes, though in practice AudioFile's longest request is substantially shorter. All data in the requests are kept naturally aligned inside the request header; requests that use additional data are padded to a 32-bit boundary.

There are 37 requests in the AudioFile protocol. Most of these are related to audio, although only two deal with audio data. The remaining requests are used for housekeeping purposes, such as access control, inter-client communications, and extensions (although no extensions are implemented today). In comparison, the X Window System has 119 requests in the core protocol.

At connection setup, the client and server exchange version information and clients provide the server authentication information, exactly as in the X Window System. Table 1 summarizes AudioFile's protocol requests.

5.4 Audio Device Attributes

An abstract audio device has several attributes that are visible to clients. The sampling rate, sample data type, and buffer size have been discussed in previous sections. This and other information is returned for each device at connection setup time. The additional information includes the number of channels for record and playback and whether a channel is connected to a telephone device.

An audio device may have multiple inputs or outputs. For example, some devices may have both line-in and microphone-in connectors which share a single ADC, and line-out and speaker-out connectors driven from a common DAC. The abstract audio device encodes these capabilities of the audio hardware in two quantities indicating the number of inputs and outputs and two masks indicating which inputs and outputs are connected to a telephone line interface.

We intend to modify AudioFile to make the sample data type attribute a prioritized list rather than a single enumerated type. This change would permit the system to have several conversion modules per audio device. These conversion modules can translate one or the enumerated data types to the native audio hardware data type. We will probably extend the same scheme to handle various popular compression methods.

Audio and Events	SelectEvents	Select which events the client wants
	CreateAC	Create an audio context
	ChangeACAttributes	Change the contents of an audio context
	FreeAC	Free an audio context
	PlaySamples	Play samples
	RecordSamples	Record samples
	GetTime	Get the audio device's time
Telephony	QueryPhone	Get telephone state
	EnablePassThrough	Enable telephone passthrough
	DisablePassThrough	Disable telephone passthrough
	HookSwitch	Control hookswitch
	FlashHook	Flash hookswitch
	EnableGainControl	Not for general use
	DisableGainControl	Not for general use
	DialPhone	Obsolete, do not use
I/O Control	SetInputGain	Set input gain
	SetOutputGain	Set output gain (volume)
	QueryInputGain	Find out current input gain
	QueryOutputGain	Find out current output gain
	EnableInput	Enable input
	EnableOutput	Enable output
	DisableInput	Disable input
	DisableOutput	Disable output
Access Control	SetAccessControl	Set access control
	ChangeHosts	Change access control list
	ListHosts	List which hosts are permitted access
Atoms and Properties	InternAtom	Allocate unique ID
	GetAtomName	Get name for ID
	ChangeProperty	Change device property
	DeleteProperty	Remove device property
	GetProperty	Retrieve device property
	ListProperties	List all device properties
Housekeeping	NoOperation	Non-blocking NoOperation
	SyncConnection	Round-trip NoOperation
	QueryExtension	Not yet implemented
	ListExtensions	Not yet implemented
	KillClient	Not yet implemented

Table 1: AudioFile protocol requests

5.5 Telephony

While we have evolved the design of AudioFile and added support for several devices, LoFi is still the only device supported by AudioFile that has an analog telephone line interface.

LoFi's telephone line interface includes a line jack, a set jack, hookswitch relay, ring detection circuitry, loop current detection circuitry, Dual Tone Multi-Frequency (Touch-Tone) decoding circuitry, and output power limiting circuitry. With suitable software this hardware allows applications to originate calls using DTMF or pulse dialing, receive calls, receive DTMF events, monitor the extension phone status (on-hook or off-hook), and source and sink audio to/from the telephone line for applications such as voice mail and remote information access.

While the current protocol includes a DialPhone request, it is not used, because we found it difficult to meet FCC timing requirements for dialing by using our internal tasking system in the server. Instead, the client library implements client side tone dialing by generating appropriate tones and using device time to play them at exactly the right time. We do not support pulse dialing, though the LoFi hardware could in principle do so.

Client applications can learn of state changes in the telephone line interface by monitoring the telephone events. DTMF detect, loop current detect, hookswitch, and ring detect events can be generated by the server each time there is a change in state. DTMF detection can be used to receive information from a remote caller. Loop current detection (PhoneLoop) can be used to indicate if the extension phone is on-hook or off-hook. Loop current detection together with DTMF detection can be used to track numbers dialed manually on the extension phone. Hookswitch events (HookSwitch) can be used to determine if the telephone line interface is on-hook or off-hook. Finally, ring detect events (PhoneRing) can be used to determine whether there is an incoming call.

Other than the support for events and device control, there are no special audio arrangements for supporting telephony in the AudioFile design.

5.6 Audio Contexts

Rather than specifying all parameters for play and record with each request, a client uses an "audio context" (AC) to encapsulate most of these parameters. The AC includes the play gain (relative to the 0 dB point of all clients, independent of user volume control) and preemption flag. The client AC data structure also stores the number of channels, sample type, and byte order. ACs simplify the programming interfaces for play and record considerably.

5.7 GetTime, Play, and Record

GetTime is the protocol request which returns the audio device time. PlaySamples and RecordSamples requests also return the device time as a convenience to the application programmer.

The PlaySamples and RecordSamples protocol requests are almost symmetric. Both pass the start time for the play or record to begin, the number of samples to use, the number of channels in the data, and the data type, with a flag bit to indicate the byte order of the data. An audio context specifies the device, gain parameters, and in the PlaySamples case, a preemption flag that controls mixing. In addition, PlaySamples uses a flag to specify whether the server should suppress the usual time reply, because the client library does not need intermediate replies during a series of contiguous play requests. RecordSamples uses another flag to control whether the server should block the client if not all the requested data can be returned immediately.

At the client library interface, long play and record requests are "chunked" into 8K byte pieces, so that no single request will take very long for the server to process.

5.8 Input and Output Gain, and I/O Control

Each audio device may have multiple input or outputs. AudioFile provides requests to select inputs and to enable or disable outputs. In addition, the gain can be controlled for each input or output device for use as end user volume control.

5.9 Inter-Client Communications

AudioFile adopted from X the same extensible atom type system and property list mechanism to enable clients to communicate. Atoms are short unique integer handles for strings. There are a set of built-in atoms for commonly used types and property names such as sample types, time, and so on. New types or property names can be added by "interning" new strings to create new atoms. Named, typed data (called "properties") can be associated with a device, given a name and type, and stored and retrieved from the server. Table 2 summarizes AudioFile's built-in atoms. Clients can register to be notified by event when properties are changed by other clients.

Primitive types	
ATOM	Unique id for a string
CARDINAL	Unsigned integer
INTEGER	Integer
STRING	String
AC	Audio context ID
DEVICE	Device number
TIME	Time
MASK	Bit vector, often inputs or outputs
TELEPHONE	Telephone device type
COPYRIGHT	Copyright string
FILENAME	Filename string
Encoding types	
SAMPLE_MU255	μ-law
SAMPLE_ALAW	A-law
SAMPLE_LIN16	16-bit linear
SAMPLE_LIN32	32-bit linear
SAMPLE_ADPCM32	ADPCM compressed
SAMPLE_ADPCM24	ADCPM compressed
SAMPLE_CELP1016	CELP compressed
SAMPLE_CELP1015	CELP compressed
Properties	
LAST_NUMBER_DIALED	Type STRING, contains last number dialed

Table 2: AudioFile built-in atoms

For example, the property LAST_NUMBER_DIALED can be used by cooperating applications for storing the last telephone number dialed. It would have type STRING and its name would be LAST_NUMBER_DIALED, with the convention that any client dialing the telephone should update the value of this property. Other clients interested in tracking telephone activity would register for notification of changes. In this way, a directory of recently used numbers could acquire all numbers dialed by all telephone applications.

Clients can use such facilities to coordinate use of resources (like the telephone) and to cooperate among themselves, allowing a collection of small applications to implement complex functions, rather than requiring a single monolithic application.

6 Client Libraries

We have developed two libraries for use by clients of AudioFile. The first is a "core" library that provides the standard interface to an AudioFile server. The second is a utility library that includes common functions required by many clients.

The client libraries perform two functions. The first is as the sole interface to the protocol. These functions include connection management, local maintenance of data structures such as the client-side copy of the audio context and device data, translation of client requests into protocol requests, demultiplexing of the reply/event stream, and buffer management of the communications channel. The second main function of the client libraries is to provide *language bindings* of the requests, events, and functions suitable for a particular client programming environment. We currently supply bindings only for C language and semantics, but other languages could be added.

6.1 Core Library

The core client library is the standard interface for AudioFile clients. Some of its functions provide interfaces to the AudioFile protocol; others provide an interface to the library's internal data structures. Tables 3 and 4 summarize the library functions. The header file AFlib.h contains the necessary definitions and declarations while the library libAF.a contains the implementation.

6.1.1 Connection Management

AFOpenConnection() opens a connection to the audio server. The user can specify which server to use in the following ways: explicit argument on the command line, the AUDIOFILE environment variable, or the DISPLAY[13] environment variable. DISPLAY is used as a convenient fallback, since the user's workstation usually has both audio and graphics systems. AFAudioConnName() returns the name of the connection as used by AFOpenConnection().

AudioFile provides a simple access control scheme based on host network address. The access control functions allow programmers to add or remove hosts from the access list and to enable or disable access control entirely.

[13] DISPLAY is used by the X Window System for specifying a particular graphics display.

6.1.2 Error Handling

Several functions modify the behavior of library functions when errors occur. The default action is to exit the application. AFSetErrorHandler() can be used to specify an application-specific error handler instead. An application can handle system call errors by supplying a new handler with AFSetIOErrorHandler(). AFGetErrorText() translates a protocol error code into a string. This is commonly used to provide useful error messages to the user.

6.1.3 Synchronization

Some of the library functions, such as AFGetTime(), require an immediate response from the server; others, such as AFCreateAC(), do not. In the former case, the library blocks until a reply is received. When a response is not needed right away, the library may delay sending the request to the server and put it on an outgoing request queue. In these cases, the library function will return to the client immediately. Certain operations, including the synchronous functions, flush the outgoing request queue; these are noted below.

Sometimes it is necessary to force synchronous operation for all protocol requests, particularly when debugging. AFSynchronize() is used to enable or disable synchronous operation with the server. If synchronous operation is enabled, every library function that normally generates an asynchronous protocol request calls AFSync() before returning. AFSync() flushes all output to the server and uses a synchronous protocol request to wait for the server's reply. If an application requires a different synchronization procedure, it can specify one using AFSetAfterFunction().

AFFlush() flushes the output buffer. Most client applications will not need to call this, because the output buffer is flushed as needed by calls to AFPending() and AFNextEvent(). Any events generated by the server are put onto the library's event queue.

6.1.4 Events

The library filters events out of the data stream from the server and keeps them on a private queue. This allows events to be interspersed on the audio connection with other traffic from server to client.

Several functions can be used to examine and manipulate the library's event queue. The most important one is AFNextEvent(), which returns the next event in the queue. If the queue is empty, it will flush the output buffer and block until an event arrives. AFEventsQueued() checks the queue for pending events. Depending upon its arguments, it may check only previously read events, unread but available events, or it may flush the output buffer and try to read new events. AFPending() is similar, but it returns only the number of pending events that have been received but not yet processed.

Occasionally a client may wish to block until a specific event occurs. To do this, the client calls AFIfEvent() with a predicate procedure. AFIfEvent() blocks until the predicate returns True for an event in the queue. The matching event is removed from the queue and copied into a client-supplied AEvent structure. If the client does not wish to block when checking for a specific event, it can use AFCheckIfEvent(), which removes a matching event (if there is one) from the queue and copies it into a client supplied AEvent structure. As an alternative, a client can call AFPeekIfEvent(), which is like AFCheckIfEvent(), but it does not remove the event from the queue.

6.1.5 Audio Handling

AFPlaySamples() is used by an application to play back digital audio. The block of samples in the given buffer is played back starting at the specified time.

```
ATime AFPlaySamples(AC ac, ATime startTime, int nbytes, unsigned char *buf)
```

A client can use the startTime parameter of a call to AFPlaySamples() to schedule samples to be played at any time in the near future. The time parameter specifies when the initial sample of the request is to be played. The precise behavior of the server depends upon the requested time:

- Past. If part or all of the request is scheduled for the past , the server discards that part of the request and plays any remaining samples beginning with the current time.
- Near future. If any part of the request is scheduled for the interval between “now” and four seconds in the future, the server copies that part of the request directly to the appropriate location in the playback buffer.
- Beyond near future. If any part of the request is scheduled for the interval beyond four seconds in the future, the server will block the client until the

rest of the request can be safely copied into the playback buffer. This is the only case in which AFPlaySamples() will not immediately return control to the client application.[14]

The client is also allowed to modify any scheduled playback material right up until the moment the samples have been played. If no client request is received for a given time interval, the server plays silence.

AFPlaySamples() returns the current AudioFile device time, as would be returned by a call to AFGetTime(). This is done as a convenience to programmers. We noticed that many applications would alternately call AFPlaySamples() and AFGetTime(). Adding the return value makes application programming easier and reduces client/server communications.

AFRecordSamples() is used by an application to capture sound in digital form. A block of samples beginning at the time specified is filled into the given buffer. AFRecordSamples() also returns the current device time.

```
ATime AFRecordSamples(AC ac, ATime startTime, int nbytes,
                      unsigned char *buf, ABool block)
```

A client may use AFRecordSamples() to request samples from either the past or the future. If block is ABlock, then AFRecordSamples() blocks until all of the requested data is available. If block is ANoBlock, it returns whatever data is immediately available; the returned time can be used to compute how many samples were actually returned. The precise behavior of the server depends upon the requested time:

- Distant past. If part or all of the request is for samples from more than four seconds in the past, that part of the request is filled by samples representing silence. This is data no longer retained by the server.
- Recent past. If part or all of the request is for samples from the interval between four seconds ago and the present, that part of the request is filled by the appropriate samples from the record buffer. If the entire request is in this interval, the call to AFRecordSamples() will return without blocking.

[14] The server should probably return an error indication for the "distant" future, because such requests usually indicate programming errors.

- Future. If part of the request is for samples from the future, AFRecordSamples() will block until the data is available, but will return as soon as the last requested sample becomes available.[15]

6.2 Client Utility Library

The AudioFile distribution also includes a utility library libAFUtil.a, whose contents are described in the header file AFUtils.h.

The utility library provides a number of facilities that are used by several clients. Two kinds of facilities are provided: tables and subroutines. Table 5 summarizes the library tables and Table 6 summarizes the library subroutines.

6.2.1 Utility Tables

The AudioFile System handles a variety of digital audio data formats, particularly μ-law and A-law, the eight-bit-per-sample companded formats used in the US and European telephone industries, respectively. These formats are described by CCITT recommendation G.711. They are similar logarithmically companded formats resembling 8-bit floating point numbers. The μ-law (A-law) format is roughly equivalent to a linearly encoded format of 14 (13) bits. For mixing and gain control, the AudioFile server and some clients need to convert these formats to and from linear encoding. It is possible but time consuming to do this algorithmically; fortunately, it is very easy to do the necessary conversions by table lookup. Conversion from μ-law or A-law to linear requires tables containing 256 16-bit entries. Gain control for a specific gain requires only a 256 byte table. Tables for conversion from linear to μ-law or A-law requires 16,384 bytes.

Another frequent operation is the computation of the signal power of a block of samples. The utlity library provides tables AF_power_uf and AF_power_af to translate μ-law and A-law values to the square of the corresponding linear value.

As discussed further below, table lookup is a very powerful method of generating sine waves or other wave shapes at various frequencies. The library provides integer and floating point sine wave tables, AF_sine_int and AF_sine_float, for this purpose. Finally, the library makes a first attempt to describe various encoding formats that AudioFile supports today or may support in the future. The table AF_sample_sizes is an array of AFSampleTypes structure indexed by AEncodeType, which is an

[15] As with AFPlaySamples(), the notion of future should probably be bounded.

enumerated type describing various digital audio encodings.

```
struct AFSampleTypes {
  unsigned int bits_per_samp;
  unsigned int bytes_per_unit;
  unsigned int samps_per_unit;
  char *name;
};
```

Many encoding types do not have integral numbers of bytes per sample, so AFSampleTypes has two fields for bytes_per_unit and samps_per_unit. Together these fields can describe any fixed-length encoding format. (The field bits_per_samp is only a hint).

6.2.2 Utility Procedures

The AudioFile utility library gathers together a number of useful subroutines. With the exception of AFDialPhone(), these procedures do not directly interact with the AudioFile protocol.

Two procedures, AFMakeGainTableU() and AFMakeGainTableA(), are supplied to compute on-the-fly translation tables for gain modification of μ-law and A-law encoded samples. Applications may find it more convenient to use the precomputed tables (AF_gain_table_u and AF_gain_table_a), but the procedures are provided for those situations calling for gain values outside the range -30 dB to +30 dB or for clients without enough memory to store all 61 tables.

Two procedures, AFTonePair() and AFSingleTone(), are supplied for generating tones or tone pairs. These procedures use the technique of direct digital synthesis, where sample values are produced by stepping through a wave table at a rate proportional to the requested frequency. The requested frequency is divided by the sample rate to produce a phase increment value. The phase increment is added to a phase accumulator, and the fractional value is used to index the wave table.

AFSingleTone() is used to generate a floating point tone into a buffer, with a given peak value. AFSingleTone() accepts an initial phase and returns the final phase, allowing multiple calls to AFSingleTone() to produce a signal that is continuous at block boundaries.

```
double AFSingleTone(double freq, double peak, double phase,
                    float *buffer, int length)
```

AFTonePair() is used to generate a μ-law tone pair into a buffer. The two frequencies are individually specified, with individual power levels relative to the "digital milliwatt", which in turn is 3.16 dB down from digital clipping level. A special parameter, gainramp, controls how the tones will ramp up to full volume and ramp down at the end. This reduces the frequency splatter associated with switching the signal on and off. Two-tone signals are frequently used in telephony, for Touch-Tone, ringback, busy, and dialtone sounds. Table 7 shows some of the telephony related signals represented by tone pairs. The table shows the frequencies in Hertz and power levels in dB relative to the digital milliwatt of the two tones, and the on- and off- times in milliseconds. An off-time of 0 represents a continuous tone.

```
void AFTonePair(double f1, double dBgain1,
                double f2, double dBgain2,
                int gainramp,
                unsigned char *buffer, int length);
```

AoD() stands for "Assert Or Die". A common idiom in programming is to check a condition and exit with an error message if the condition does not hold. AoD() simply captures this idiom into a library procedure. The first argument is a boolean expression. If the expression is true, AoD() returns right away. If the expression is false, the rest of the arguments are interpreted as a format string and arguments for fprintf(stderr,...) after which AoD() calls exit(1).[16]

```
void AoD(int bool, char *errmsg, ...);
```

Finally, AFDialPhone() encapsulates the operations necessary to generate Touch-Tone dialing sequences on a telephone device.

[16] This should be in a more general library, but it isn't.

Connection Management	
AFOpenAudioConn	Open a connection to the audio server
AFCloseAudioConn	Close the audio connection
AFSynchronize	Synchronize with the audio server
AFSetAfterFunction	Set a synchronization function
Audio Handling	
AFGetTime	Get the device time of a device
AFPlaySamples	Play digital audio samples
AFRecordSamples	Record digital audio samples
Audio Contexts	
AFCreateAC	Create a new audio context
AFChangeACAttributes	Modify an audio context
AFFreeAC	Free resources associated with an audio context
Event Handling	
AFEventsQueued	Check for events
AFPending	Returns number of unprocessed events
AFIfEvent	Find and dequeue a particular event (blocking)
AFCheckIfEvent	Find and dequeue a particular event (non-blocking)
AFPeekIfEvent	Find a particular event (blocking)
AFNextEvent	Return the next unprocessed event
AFSelectEvents	Select events of interest
Telephone	
AFCreatePhoneAC	Create an audio context for a telephone device
AFFlashHook	Flash the hookswitch on a telephone device
AFHookSwitch	Set the state of the hookswitch
AFQueryPhone	Returns the state of the hookswitch and loop current

Table 3: AudioFile client library functions

I/O Control	
AFEnableInput	Enable inputs on an audio device
AFDisableInput	Disable inputs on an audio device
AFEnableOutput	Enable outputs on an audio device
AFDisableOutput	Disable outputs on an audio device
AFEnablePassThrough	Connect local audio to the telephone
AFDisablePassThrough	Remove the direct local audio/telephone connection
AFQueryInputGain	Get minimum/maximum input gains for a device
AFQueryOutputGain	Get minimum/maximum output gains for a device
AFSetInputGain	Set the input gain of a device
AFSetOutputGain	Set the output gain of a device
Access Control	
AFAddHost	Add a host to the access list
AFAddHosts	Add a set of hosts to the access list
AFListHosts	Return the host access list
AFRemoveHost	Remove a host from the access list
AFRemoveHosts	Remove a set of hosts from the access list
AFSetAccessControl	Enable or disable access control checking
AFEnableAccessControl	Enable access control checking
AFDisableAccessControl	Disable access control checking
Properties	
AFGetProperty	Manipulate properties
AFListProperties	Get a list of existing properties
AFChangeProperties	Modify a property
AFDeleteProperty	Delete a property
AFInternAtom	Install a new atom name
AFGetAtomName	Fetch the name of an atom
Error Handling	
AFSetErrorHandler	Set the fatal error handler
AFSetIOErrorHandler	Set the system call error handler
AFGetErrorText	Translate error code to a string
Miscellaneous	
AFNoOp	Don't do anything
AFFlush	Flush any queued requests to the server
AFSync	Default synchronization function
AFAudioConnName	Return the name of the audio server

Table 4: Additional AudioFile client library functions

Conversion Tables	
AF_comp_u	13-bit linear to μ-law
AF_comp_a	13-bit linear to A-law
AF_exp_u	μ-law to 13-bit linear
AF_exp_a	A-law to 13-bit linear
AF_cvt_u2s	μ-law to 16-bit linear
AF_cvt_u2a	A-law to 16-bit linear
AF_cvt_u2f	μ-law to floating point
AF_cvt_a2f	A-law to floating point
AF_cvt_u2a	μ-law to A-law
AF_cvt_a2u	A-law to μ-law
Mixing Tables	
AF_mix_u	Mix two μ-law samples
AF_mix_a	Mix two A-law samples
Gain Tables	
AF_gain_table_u	μ-law gain
AF_gain_table_a	A-law gain
Sine Wave Tables	
AF_sine_int	1024 entry 16-bit integer sine wave
AF_sine_float	1024 entry floating point sine wave
Encoding Information Tables	
AF_sample_sizes	Datatype information

Table 5: AudioFile client utility library tables

Gain Control Procedures	
AFMakeGainTableU	Generate a μ-law gain table
AFMakeGainTableA	Generate an A-law gain table
Signal Generation Procedures	
AFTonePair	Generate a two-tone signal
AFSingleTone	Generate a precise sine wave
AFSilence	Generate silence
AFDialPhone	Generate tone dialing signals
Utility Procedures	
AoD	Assertion checking

Table 6: AudioFile client utility library functions

Name	*f1*	*dB1*	*f2*	*dB2*	Time-on	Time-off
Call Progress Tones						
dialtone	350	-13	440	-13	1000	0
ringback	440	-19	480	-19	1000	3000
busy	480	-12	620	-12	500	500
fastbusy	480	-12	620	-12	250	250
DTMF Tones						
1	697	-4	1209	-2	50	50
2	697	-4	1336	-2	50	50
3	697	-4	1477	-2	50	50
4	770	-4	1209	-2	50	50
5	770	-4	1336	-2	50	50
6	770	-4	1477	-2	50	50
7	852	-4	1209	-2	50	50
8	852	-4	1336	-2	50	50
9	852	-4	1477	-2	50	50
$*$	941	-4	1209	-2	50	50
0	941	-4	1336	-2	50	50
#	941	-4	1477	-2	50	50
A	697	-4	1633	-2	50	50
B	770	-4	1633	-2	50	50
C	852	-4	1633	-2	50	50
D	941	-4	1633	-2	50	50

Table 7: Tone pairs for telephony

7 Server Design

The AudioFile server is responsible for managing the audio hardware and presenting abstract device interfaces to clients via the AudioFile protocol. This section discusses some of the important issues in the server's design, the implementation of buffering to provide the audio device abstraction, and some other details of the server's implementation.

7.1 Implementation Considerations

Performance was our primary concern for the implementation of an AudioFile server. We wanted the server to run continuously in the background, so we felt that the quiescent server should present a negligible CPU load. Further, load due to the server with a few clients running should leave most of the CPU available for applications. Otherwise, users would not be inclined to use audio-based applications because they would not get any work done. While server performance was a primary focus, we realized the overall design must be kept in balance so that an efficient server was not compromised by an inefficient client or client library.

We considered using threads to implement the server, but were apprehensive about the performance and portability of existing thread packages. Although the internal structure of the server might be slightly cleaner with threads, we took the safer route and designed the server as a single-threaded process.

The server must be fair in its processing of client requests, in order to meet the real-time constraints of the applications. To satisfy our fairness goal, the server is designed such that one client cannot dominate the processing time within the server and preclude the server from getting work done on the behalf of other clients. The server attempts to achieve fairness by servicing active client connections in a round-robin fashion and by breaking large requests into smaller chunks.

7.2 Buffering

In Section 1, we mentioned the existence of audio device input and output buffers. There are several buffering details that merit further examination. Figure 4 pictorially represents the input and output buffers along a time line centered around *now*, the current value of the time register.

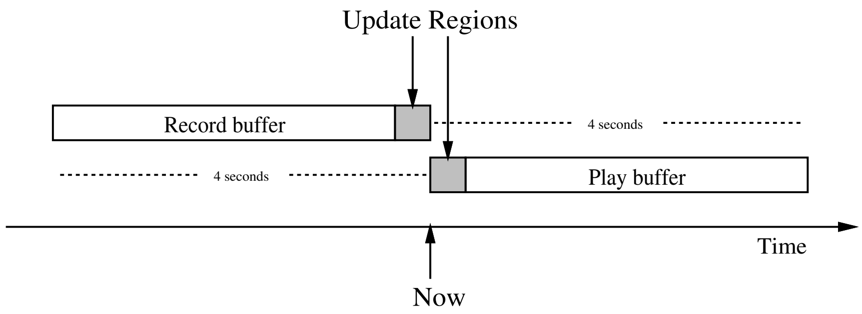


Figure 4: AudioFile server buffering

Play or record requests that correspond to points on the time line that map to unshaded portions of the buffers are handled trivially. Record requests can be serviced from the record buffer and play request data can be simply mixed into the play buffer.

Requests that correspond to the shaded regions of the buffers are treated as special cases by the server. The shaded "update regions" represent where the server buffers may be inconsistent with the audio hardware buffers. If a record request falls into the shaded region, the server performs a record update operation which makes the input record buffer consistent until now. [17] Once the input record buffer has been updated, recent data can be delivered to the client. If a play request falls into the shaded region, the server writes the data through the server buffer into the audio hardware (or low level software) in order to ensure that sample data for the near-future is immediately available to the DAC without intervention by the server.

In these descriptions of how the server buffers data, we have described how client requests cause movement to occur in the input and output buffers. It should be clear that a piece of the buffering picture is missing. There must be a mechanism which periodically moves data between the audio hardware and the server buffers independent of any client request activity. This mechanism is an update task that keeps the hardware buffer consistent, such that the hardware buffer always reflects the server's buffer at the time the hardware consumes an output sample.

Figure 5 illustrates the server record and play buffers before and after the update task executes. At each invocation, the update task moves new record data (since

[17] Or very close to now. It is possible that the record update task can only make the buffer consistent through a time that occurred a few sample ticks in the past depending upon the latency in the audio hardware.

timeRecLastUpdated) from the hardware buffer to the server buffer, and moves the next batch of playback data (starting at the "before" timeNextUpdate) from the server buffer to the hardware buffer.

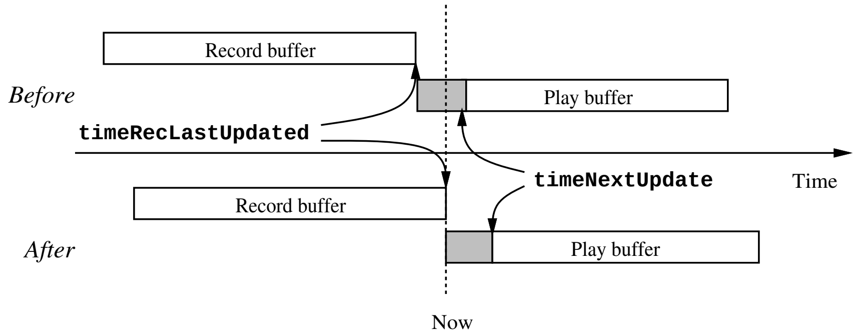


Figure 5: AudioFile periodic update task

If a record request falls after timeRecLastUpdated, the server performs an update before handling the request. If a playback request falls before timeNextUpdate, the server writes the data all the way through to the hardware.

Since the update task generally copies all data (whether any client sample was written to the output buffer or not), the server buffer must be initialized with silence data. In general, the server's buffers are implemented as circular buffers. Therefore, the silence data is written by the update task in the segment (now stale) from the time of the last update until now. This places a constraint on the server's play handler; it must consider the server's play buffer as ending at the device time of the last update plus the server buffer size.

The server was designed to mix output data from multiple clients by default. We believe this is the natural case. However, it is important that a client can preemptively play sample data, such as in an urgent warning message. Clients specify preemption through the audio context used in the play request. If preemption is specified, the play data will overwrite any data already in place. [18]

In cases where hardware buffer accesses are expensive, the server should attempt to minimize the number of hardware accesses for each sample played. At one extreme, the server could perform an access for each client playing at time T. The

[18] We have yet to devise a CPU and memory efficient scheme that supports a stacking order for clients playing data. Imagine that the relative mixing levels are controlled by the client's location on the stack with each level mixing 6 dB lower than the previous level for example. A design that supports this would require unique server buffers for each active client.

minimum is one access for all clients playing audio at time T. For recording, the server always performs one access per sample.

The number of accesses per output sample is approximately one for the periodic update task access as the server's buffer slides into the update interval, plus one for each client playing in the update interval. A large number of applications will play audio by getting the current time and begin playing immediately and quickly move outside the update interval because they run faster than real-time (for example: abiff, background music, audio announcements). These clients enable the server to approach the minimum number of accesses desired. This may not be true for low-latency applications which always schedule playback just ahead of the current time.

7.3 Server Implementation

An AudioFile server is organized like an X server. It includes device independent audio (DIA), device-dependent audio (DDA), and operating system (OS) components. The DIA section is responsible for managing client connections, dispatching on client requests, sending replies and events to clients, and executing the main processing loop. The DDA section is responsible for presenting the abstract interface for each supported device and contains all device-specific code. Finally, the OS section includes all the platform or operating system-specific code. Much of the OS and DIA code is based on X11R4.

The remainder of this section describes some of the functions provided by the DIA, DDA, and OS components as well as presenting some example DDA servers. The discussion is detailed; the interested reader may wish to refer to the source code. Because much of the OS and DIA infrastructure is based on X11R4 code, various documents describing the implementation of the X Window System server [1, 2, 13, 14] may be helpful.

7.3.1 Device-Independent Audio Server

Interoperability

The server contains code to support byte-swapping when communicating with clients on a machine with the opposite byte order. Each protocol request may have a companion swap procedure that interprets the contents of the request and byte-swaps the necessary fields. Play and record requests specify the byte order of the sample data; by default, the AudioFile library uses the byte order of the client unless told otherwise by the application.

Tasks

Instead of using threads, we implemented a simple task mechanism which allows procedures to be scheduled for execution at future times, outside the main flow of control. The task mechanism is used by the server's update mechanism and by the dispatcher to resume execution of partially completed client requests.

A task includes the address of a procedure (proc()), the system time to execute (systime), and closure data (p, time, len, ssize, mask, aDev, ac). [19] Procedures can create a new task, initialize the task, and add the task to the run list.

```
typedef struct tTask {
    struct Task *next;        /* Next task on free list.          */
    ClientPtr    client;      /* Pointer to client struct.        */
    fd_set       fdmask;      /* Save fd mask for processing loop.*/
    pointer      request;     /* Client request information.      */
    ATime        time;        /* ATime at which to process task.  */
    VoidProc     proc;        /* Procedure to call.               */
    pointer      p;           /* Pointer to the task data.        */
    int          len;         /* Amount of data left.             */
    int          ssize;       /* sample size of remaining data    */
    int          mask;        /* request mask: endian-ness        */
    AudioDevicePtr aDev;      /* Pseudo device handle.            */
    ACPtr        ac;          /* Audio context handle.            */
    struct timeval systime;   /* System time (for scheduling).    */
} Task;

/* Exported procedures. */
void        AddTask(VoidProc proc, TaskPtr task, int ms);
TaskPtr     NewTask(void);
```

Here is a code example from the Alofi server that uses the task interface. The update procedure is named codecUpdateTask(). During the server initialization sequence, the DDA creates a new task, attaches an audio device structure to the task structure, and schedules codecUpdateTask() to run MSUPDATE milliseconds from now.

```
    TaskPtr task;
    AudioDevicePtr aDev;

    task = NewTask();
    task->aDev = aDev;
    task->time = 0;
    AddTask(codecUpdateTask, task, MSUPDATE /* 100 */ );
```

[19] This data should be private to proc().

The codecUpdateTask() procedure will be invoked once system time has advanced beyond the task's expiration time. As shown by this example, the update task reschedules itself for execution MSUPDATE milliseconds into the future, causing this procedure to execute periodically. codecUpdateTask() calls codecUpdate() which does the actual work of updating the server buffers.

```
void
codecUpdateTask(TaskPtr oldTask)
{
    TaskPtr        newTask=NewTask();
    AudioDevicePtr aDev=oldTask->aDev;

    *newTask = *oldTask;            /* Task for next time.          */

    /* Get the current device time and update audio device time.    */
    CODEC_UPDATE_TIME(aDev);

    /* Perform the write-back update with silence fill.             */
    codecUpdate(aDev);

    newTask->time = aDev->time0;    /* Mark new task with old time. */
    AddTask(codecUpdateTask, newTask, MSUPDATE /* 100 */);
}
```

Main Loop

At the core of the DIA section is the main control loop. Inside of this loop the procedure WaitForSomething() is called when the server does not have anything to do. WaitForSomething() returns when a client, audio device, or task needs attention. WaitForSomething() relies heavily on the select() system call. select() is called with file descriptors for client connections and open devices, as well as a timeout argument for the next task which needs to execute. When select() returns, the server runs any pending tasks and then handles input events and client requests.

Client requests are processed by the dispatcher. The request type is used to index into a table of protocol request handler procedures. All handlers are implemented by the device-independent part of the server, but audio requests are passed to the device-dependent part. When necessary, the request handler calls into the DDA using interfaces that are defined below.

7.3.2 Device-Independent and Dependent Server Interfaces

This section describes the interfaces shared between the device independent (DIA) and device-dependent audio (DDA) server components. The interfaces include the procedures exported by the DDA and the DIA as well as the shared data structures: AudioDeviceRec and AC.

Exported DDA Interfaces

When the components of an AudioFile server are linked while excluding the DDA library, the linker complains about six unresolved symbols. The six symbols, described in detail below, are InitDevices(), ProcessInputEvents(), ddaGiveUp(), AbortDDA(), ddaUseMsg(), and ddaProcessArgument().

InitDevices() is called during server initialization from dia/main.c.[20] This procedure creates and initializes an AudioDevice structure for each (abstract) audio device supported by the DDA. The DDA can perform any necessary hardware initialization at this time.

The DDA registers file descriptors for open devices that may deliver events with the DIA to be used in generating the arguments to select(). If select() returns because a hardware device becomes ready for I/O, the DIA will call the ProcessInputEvents() procedure within the DDA. The ProcessInputEvents() procedure in the DDA is called from dia/dispatch.c when there are events pending and it is time to process them. The DDA removes pending events from a device driver input queue and then posts them to the DIA's FilterEvents() procedure for further processing before being sent to interested clients.

The ddaGiveUp() procedure is invoked by the dispatcher in dda/main.c if dispatchException masked with DE_TERMINATE is true. [21] The DDA should close any open devices and tear down its state.

The AbortDDA() procedure is invoked from within AbortServer() in os/4.2bsd/utils.c. Any fatal server error will cause AbortServer() to be invoked and subsequently, AbortDDA(). The DDA should close any open devices and tear down its state.

The ddaUseMsg() procedure is invoked from within os/4.2bsd/utils.c if the server command line could not be parsed successfully. The DDA server should use ErrorF() to print formatted error messages indicating the list of DDA-specific server command line switches.

The ddaProcessArgument() procedure is invoked from within os/4.2bsd/utils.c if the command line argument is not understood by the DIA server. The DDA server should check to see if this is a DDA-specific argument. If it is, the DDA should consume this argument and any subsequent related arguments that follow immediately.

[20]Pathnames are relative to the server directory in the AF sub-tree within the AudioFile source kit.
[21]We do not think this can happen.

ddaProcessArgument() should return the number of arguments consumed.

Exported DIA Interfaces

The device-independent audio section of the server exports several procedures to the device-dependent audio section. These procedures are briefly described below.

The DDA creates an audio device by using the MakeDevice() procedure. MakeDevice() returns a pointer to a newly created AudioDeviceRec structure (described below).

The AddEnabledDevice() and FilterEvents() procedures are used by DDA implementations that produce events. The DDA informs the DIA of open devices through AddEnabledDevice(). The DDA hands events to the DIA from the ProcessInputEvents() procedure by calling FilterEvents() with a pointer to an initialized event structure and an audio device number.

The exported interfaces to the task mechanism are the NewTask() and AddTask() procedures. NewTask() is used to allocate a task structure. Once the DDA adds its private data to the structure, it schedules it for execution by passing the task structure to AddTask().

The Xalloc() and Xfree() procedures allocate and free memory.

The DDA uses ErrorF() to output formatted warnings and informational messages. The FatalError() procedure is called to output an informational message and then die.

Audio Device Structure

The AudioDeviceRec structure is used to share information between the DIA and DDA components. This structure encapsulates the information specific to an abstract audio device. For convenience, similar fields within this structure are grouped together and described separately.

The first grouping in the AudioDeviceRec structure includes general fields for dealing with the audio device. For example, index indicates the audio device number and type indicates the type of enumerated audio device. The remaining elements in this group are private to the audio device and are located in this structure for convenience. userProps is used only by the DIA. devPtr and privPtr are only used by the DDA.

```
typedef struct _AudioDevice {
   /* ... */

  int           index;          /* Index of audiodev device.  */
  DevType       type;           /* Codec, Hi-Fi ...           */
```

```
  /* */
  PropertyPtr    userProps;       /* Properties for this device.*/
  /* DDA hangs a physical device structure here                */
  pointer        devPtr;
  /* Audiodev device private information is attached here.     */
  pointer        privPtr;

   /* ... */
} AudioDeviceRec;
```

This next grouping contains the elements to maintain the audio device time. The server's copy of the time register is held in time0 and represents the server's view of current time. dsptime and oldDevTime are used by the server to maintain time0. If the hardware maintains a time register that is narrower than 32 bits, the server uses the difference between two consecutive hardware time register values to update time0. The previous hardware time register is held in oldDevTime. dsptime is the current view of the hardware time register and is only maintained here for implementation convenience. If the hardware time register is a 32-bit register, then time0 can be a copy of that register.

```
  /* Some time information.                                    */
  ATime          time0;           /* Last computed.            */
  ATime          oldDevTime;      /* Old device time for delta. */
  ATime          dsptime;         /* Holds dsp time at update.  */
```

The input and output capabilities are described by the next grouping of AudioDeviceRec structure fields. numberOfInputs and numberOfOutputs indicate the number of input and output connections that can be selected by clients. inputsFromPhone and outputsFromPhone are masks to indicate which of the input and output connections source/sink audio to/from a telephone line interface. These fields contain a binary 1 in the bit position represented by the audio device number if that input or output is connected to the phone.

```
  /* Describe the I/O capability.                                          */
  int            numberOfInputs;  /* Number of input sources               */
  int            numberOfOutputs; /* Number of output destinations         */
  unsigned int   inputsFromPhone; /* Mask of inputs conn. to phone line    */
  unsigned int   outputsToPhone;  /* Mask of outputs conn. to phone line */
```

The play and record features of the audio device are specified by the next grouping of fields. These fields indicate the sampling rate, native audio hardware data type, number of channels, and the buffer size in samples. The buffer size in seconds is computed by dividing the the buffer size by the sampling rate.

```
/* Describe the play buffer type and size.                               */
unsigned int   playSampleFreq;  /* Sampling frequency.                   */
AEncodeType    playBufType;     /* Data type supported.                  */
unsigned int   playNSamplesBuf; /* Length in samples of play buffer. */
unsigned int   playNchannels;   /* Number of channels.                   */

/* Describe the record buffer type and size.                             */
unsigned int   recSampleFreq;   /* Sampling frequency.                   */
AEncodeType    recBufType;      /* Data type supported.                  */
unsigned int   recNSamplesBuf;  /* Length in samples of record buffer*/
unsigned int   recNchannels;    /* Number of channels.                   */
```

The update task uses the next grouping of fields to manage the movement of sample data between the audio hardware and the server buffers. This process is described in the discussion of server buffering in Section 7.2.

```
/* Server Update Information.                                            */
ATime          timeLastUpdated; /* Time of last update.                  */
ATime          timeNextUpdate;  /* Time at start of next update.         */
ATime          timeLastValid;   /* Time of last valid play data          */
ATime          timeRecLastUpdated; /* time of last record update.        */

/* reference counts */
int            recRefCount;     /* Number of open record streams         */

/* Server Buffer Data */
pointer        playBuf;         /* Server's play buffer.                 */
pointer        recBuf;          /* Server's record buffer.               */
```

Lastly, this next grouping of fields contain function pointers that are used by the DIA to invoke device dependent procedures in the DDA usually as a result of client protocol requests. The procedure pointers are further grouped into sets that support time, AC, telephone interface, and device control functions. These procedures are specific to an audio device and do not have client-specific state.

```
/* ATime and Misc.                                                       */
ATime          (*GetTime)();

/* AC                                                                    */
ABool          (* CreateAC)();
```

```
/* Telephone Specific Procedures                              */
int            (*Dial)();
int            (*HookSwitch)();
int            (*FlashHook)();
int            (*HookSwitchState)();
int            (*LoopCurrentState)();
int            (*TLICraftHookSwitchEvent)();

/* Device control procedures.                                 */
void           (*ChangeOutput)();
void           (*ChangeInput)();
void           (*ChangePassThrough)();
int            (*QueryOutputGain)();
int            (*QueryInputGain)();
int            (*SelectOutputGain)();
int            (*SelectInputGain)();
```

Audio Contexts

The context in which a client plays or records audio data is held in the AC structure. The server maintains play and record conversion procedure pointers for each audio context. The procedure pointers are used by the DIA to invoke the appropriate audio context handler in the DDA. For example, play and record requests are handled by context-specific procedures supporting the design of the output and input conversion modules described earlier. Conceptually, the AC encapsulates audio device attributes and handlers for individual clients.

```
typedef struct _ACOps {
    int    (* ConvertPlay)();
    int    (* ConvertRec)();
} ACOps;

typedef struct _AC {
    AudioDevicePtr aDev;
    AEncodeType    playType;
    AEncodeType    recType;
    int            playGain;
    int            recordGain;
    ABool          preempt;      /* Whether it should preempt.     */
    /* ... */
    ACFuncs        *funcs;
    ACOps          *ops;         /* DDA context specific handlers. */
    int            recRef;       /* Record reference count.        */
} AC;
```

Each client's play or record request contains a handle to an audio context maintained within the audio server. The audio context is used to determine the client's sample data type and the output gain (prior to mixing) and preemption mode for play requests. When the audio context structure AC is created, the DDA server initializes the input and output conversion procedure pointers in the contained ACops structure. If the client data type is identical to the hardware data type, then the DDA may choose to bypass the conversion stage. Upon receipt of a play or record client request, the dispatch handler invokes the DDA through the ACops structure permitting the DDA to implement conversion modules on a per AC basis.

7.4 Device-Dependent Server Examples

We have written servers for a variety of systems and audio hardware. These examples range from the simple 8 KHz base-board audio CODEC on Alpha AXP workstations and SPARCstations to the LoFi with two CODECs, a HiFi DAC, and a NeXT compatible DSP port. In addition to these direct connected audio hardware devices, we have implemented a server for a detached audio device named LineServer. This section describes some of the implementation details for these servers.

7.4.1 Alofi

As described in Section 1, the LoFi hardware has two 8 KHz CODECS, a DSP56001 processor, and HiFi hardware. The Alofi server presents five audio devices to clients: two CODEC audio devices and three DSP port audio devices. Each of these audio devices has a separate notion of time.

DSP Firmware

The LoFi module has a DSP processor, with 32K 24-bit words of memory shared between the host workstation and the DSP. The DSP runs a simple program written in DSP56001 assembler language that is loaded by the AudioFile server at startup.

The DSP firmware maintains several important structures in shared memory. Device time counters for the CODEC and HiFi devices are incremented once per sample. The DSP maintains the counters in 24-bit registers in shared memory. The server software updates its view of time by the difference between the previous and current samples of the DSP's counter.

The DSP also maintains input and output buffers for each audio device. The host performs audio I/O by reading and writing these buffers. There are 4 circular buffers for the CODEC devices: a play and record buffer for each of the two CODECs.

Currently, we store one 8-bit sample in each 24-bit word. One optimization would be to double or triple the CODEC buffer sizes by packing multiple samples per word, at the expense of more complex (and slower) firmware. There are also 4 circular buffers for the stereo HiFi device: a play and record buffer for each channel with one 16-bit sample in each 24-bit word.

The buffer sizes are set by the server startup code, before enabling the DSP. Each CODEC buffer contains 1024 samples (about 125 milliseconds at 8 KHz), and each HiFi buffer contains 4096 samples (about 85 milliseconds at 48 KHz). These sizes were chosen to be as large as possible given the size of the DSP static memory and to allow approximately equal buffering time for the CODEC and HiFi devices. [22] The server startup code also sets the initial configuration for the DSP port, such as sample rate and framing details. Currently, there is no way to change the configuration while the server is running.

After the DSP firmware is initialized, it goes into an infinite wait loop. All of the work is done by two interrupt routines: one each for the CODEC and HiFi devices. The CODEC interrupt routine fires at the CODEC rate (8000 Hz) and performs the following functions (for each of the CODEC devices): [23]

- Write play sample from play ring buffer to CODEC registers
- Backfill play buffer with silence.
- Read record buffer from CODEC register to record ring buffer
- Increment device time counter.

The HiFi interrupt fires at twice the HiFi rate, or once per channel. This routine performs the same functions as the CODEC update, except that the update alternates between the left and right channels.

The interrupt routines are optimized for execution speed. At high sampling rates, the overhead of processing interrupts must be minimal. For example, at a 48 KHz sampling rate, the HiFi interrupts occur every 10 microseconds. This gives an upper bound of about 280 DSP cycles (or 140 instructions, not counting time to access memory) between HiFi interrupts. To reduce overhead, the firmware minimizes memory accesses by keeping global values in registers.

[22] Since the current implementation uses the circular addressing modes supported by the DSP56001, the buffer sizes are also constrained because they must be a power of two.

[23] Both CODEC devices can be serviced at the same time since their interrupts are synchronized.

One feature that is not yet implemented is device gain control for the HiFi device. Because many HiFi hardware devices have no mechanism for manipulating the device gain, it must be done in software. Fortunately, such a mechanism would be easy to implement on the DSP chip where integer multiplies are inexpensive.

HiFi Details

The HiFi section of the DDA is similar to the section supporting the 8 KHz CODECs. The differences that do exist are described below.

The HiFi section of the LoFi server supports two modes of operation. The server supports output at 44.1 KHz through the LoFi's built-in stereo DAC. The server also supports external devices, such as the Ariel ProPort, attached to LoFi's external DSP port. External devices may provide input as well as output at a variety of sample rates.

The sample rate and operating mode are selected at server startup and cannot be changed by client applications. Currently, the only sample type supported by the server is 16-bit linear.

In our server, we implemented a single stereo device that represents both the left and right channels. (By convention, left and right samples alternate in the data stream, so a stereo "sample" consists of a 16-bit left sample and a 16-bit right sample). To support mono channel operations, we also implemented two audio devices that represent the separate left and right channels of the stereo device.

In the server, everything is implemented in stereo because it is the most common mode of operation, and it is more efficient to move stereo samples around as a unit than as two independent mono channels. The mono channel devices are built on top of the server's stereo buffers. A mono play request is simply written (or mixed) into the appropriate channel in the stereo buffers, and a record request simply reads from the appropriate channel.

Performance Considerations

To date, we have only optimized the HiFi part of the server. The memory bandwidth and CPU load requirements for supporting the CODECs does not justify the optimizations for that part of the DDA server.

Most of the time in the server is spent moving high-fidelity samples around in the play and record buffers. The server's periodic updates (the routines that move samples between the server's buffers and the hardware) can consume quite a few CPU cycles, especially at high sample rates. We had to spend some time optimizing the update procedures to achieve adequate performance.[24]

The record update only needs to run if there is a client that wants record data. AudioFile has no explicit mechanism for clients to indicate their intent to record, but it is likely that clients that record once are likely to record again. The first record operation performed under a context marks the context as recording. Each device maintains a count of recording contexts: as long as there is one or more, the record update code runs. Note that this optimization breaks clients that start up and immediately want to start recording in the past.

Similarly, the play update should run only if there are samples to play. To accomplish this, the server maintains a variable for each device, timeLastValid, with the time of the last valid playback sample written by any client. The play update code only runs when this variable is in the future relative to the current device time.

A second possible play update optimization has to do with back-filling silence. AudioFile's playback model says that periods with no playback data are filled with silence. Our first implementation achieved this by filling the play buffer with silence immediately after the play data was sent to the device. While this method was easy to implement, it doubles the memory bandwidth requirements to the play buffer. When playing continuous stream (the common case), the samples in the play buffer got written twice; once with silence, and once again with the playback data.

The solution is to fill silence only when absolutely necessary. This can be simply achieved with the timeLastValid variable. If a client play request starts in the future relative to timeLastValid, then the region from timeLastValid to the start of the play request must be silence filled. The play data is then mixed or copied into the server buffer. If a play request is preemptive, the data is copied into the server buffer. Otherwise, samples before timeLastValid are mixed and samples after timeLastValid are copied. In both cases, timeLastValid is updated if necessary to reflect the time of the last valid sample. Note that in the common case of contiguous playback requests, silence filling is never necessary.

Pass-Through

The LoFi hardware is able to directly route audio data between the CODEC devices. This turned out to be a very useful feature that permitted users to communicate through the telephone line interface from the local audio device with very low latency. While this is not a general mechanism, AudioFile supports this feature

[24] We should point out that when audio hardware with DMA support appears, this should be less of a problem.

with a device control primitive that connects the inputs and outputs of two audio devices.

7.4.2 Aaxp and Asparc

The audio servers for the base-board audio on Alpha AXP workstations and SPARCstations use device drivers with similar interfaces. As a result these two servers are nearly identical. They differ from other audio servers in that the DDA does not directly talk to the audio hardware but rather use a kernel device driver.

The base-board audio hardware is an 8 KHz CODEC. The device driver implements read and write entry points for recording and playing audio data. The hardware update procedure in the DDA writes a block of data to the device driver which is then responsible for seeing it is delivered to the CODEC. Similarly, the update procedure reads a block of data from the device driver and stores it in the server buffer.

Because the kernel device drivers do not maintain a time register for the base-board CODECs, the server must maintain an estimated value using the system clock and must occasionally resynchronize the message queue in the device driver.

7.4.3 Als

For the LineServer, an AudioFile server running on a nearby workstation uses a private UDP-based protocol to communicate with the device. The LineServer runs simple firmware that processes incoming packets and moves samples to and from the audio hardware. On the workstation, a periodic update task moves data between the server's buffers and the LineServer's buffers using the private protocol. The server makes every attempt to minimize access to the LineServer, since crossing the network is a relatively expensive operation. Only requests in the update regions require network traffic. For requests that require returning a device time, the server generates an estimate.

An AudioFile server running on the workstation drives the hardware using a special UDP-based protocol between the workstation and the LineServer. There are six packet types, supporting the following functions:

- Play samples
- Record samples

- Read CODEC registers
- Write CODEC registers
- Loopback (for testing)
- Reset

Request and reply packets have the same format, with four header fields: sequence number, audio time, function code, and parameter. Any extra bytes after the header are considered data bytes.

The LineServer only sends packets as replies to requests from the workstation.[25] All requests generate replies which consist of the original command packet header with the time updated to the current LineServer audio device time and any data bytes (if applicable).

The LineServer firmware is very simple. There are two threads of control: a network thread and an update thread. The network thread is a loop that reads request packets, processes them, then sends the reply back to the workstation. The LineServer maintains small (2048 samples, or 1/4 second at 8 KHz) record and playback buffers, and play or record requests write or read samples from these buffers. The CODEC read and write requests manipulate the CODEC registers on the LineServer. The update routines are interrupt driven, and copy play and record samples between the buffers and the CODEC. A loopback request returns the original request packet.

Client play and record requests that can be completely satisfied in the server's buffers are completed without touching the LineServer at all. Only requests that cover the update regions need to go through to the LineServer. For requests that require returning a device time (like play and record), the server generates an estimate of the LineServer time from the time stamp of the last LineServer packet and the local server time.

Other client requests, such as adjust output gain, are converted into the appropriate CODEC read or write command and sent through to the LineServer.

No attempt is made to retry play or record packets (by then, it is probably too late anyway). CODEC read and write requests are retried by the server, if necessary.

[25]There is something to be said for peripherals that speak only when spoken to.

8 AudioFile Clients

The AudioFile System includes two suites of application programs. These applications are called “clients” after the fashion of the X Window System. This is because in a client-server system like AudioFile, the application programs are clients of the facilities provided by the AudioFile server.

The first suite of clients include core applications for recording, playback, telephone control, device control, and access control. These applications have very few dependencies, so they are easily ported to new systems, yet they have enough functionality that they can be used to build useful applications. Table 8 shows the core clients, grouped by their functions of access control, device control, inter-client communications, and audio handling.

Access control	
ahost	AudioFile server access control
Device control	
ahs	Telephone hook switch control
aphone	Telephone dialer
aset	Device control
aevents	Report input events
Inter-client communications	
alsatoms	Display defined atoms
aprop	Display and modify properties
Audio handling	
apass	Record from one AF server and playback on another
aplay	Playback from files or pipes
arecord	Record to files or pipes

Table 8: AudioFile core clients

The AudioFile System distribution also includes a suite of “contributed” applications, shown in Table 9. These applications tend to be more complex or have dependencies on other software packages which are not ubiquitously available. In particular, many of the contributed clients have graphical user interfaces using the Tcl language [10] and Tk toolkit [11] developed by John Ousterhout at the University of California, Berkeley. [26] Section 11.3 explains how to get Tcl and Tk.

[26] We have found Tk to be a very effective toolkit, yet one which is much easier to understand than any of the standard X Window System toolkits. We recommend Tcl and Tk to anyone interested in graphical user interfaces.

Device control	
adial	Tk telephone dialer
axset	Tk version of aset
afxctl	X-based event display and device control
Audio handling	
abiff	Incoming email notification by audio
abob	Tk-based multimedia demonstration
radio	Multicast network audio
xplay	An X-based sound file browser
abrowse	Tk-based sound file browser
Signal processing utilities	
afft	Tk-based real-time spectrogram display
afxpow	X display of audio signal power
autil	Stdio-based signal generators

Table 9: AudioFile contributed clients

In the remainder of this section we describe the audio handling core clients in some detail, in order to illustrate the simplicity of the AudioFile client library API.

8.1 aplay — A Play Client

aplay is the primary client of the AudioFile System. It reads digital audio from a file or from standard input and sends the audio data to the audio server for playback. aplay has several options, which are discussed below.

When used to play back from a file, aplay can serve as the core of a sound-clip browser or voice mail retrieval program. When used to play from the standard input, aplay can serve as the final stage in a signal processing pipeline. For example, the output of our software implementation of the DECtalk speech synthesizer can connect directly to the input of aplay.

At this writing, aplay handles only "raw" sound files. It would be appropriate to extend aplay to handle a variety of popular sound file formats. aplay does not process the file data at all; it simply passes the data to the server. It is the user's responsibility to assure that the data is of an appropriate type for the audio device specified. One interesting benefit of this approach is that aplay is extraordinarily general purpose. It needs no modification to work on any fixed-size encoding or for any number of channels. On the other hand, the user must know the format of the file and choose an appropriate server device in order to play it back.

8.1.1 aplay Options

The command line for aplay looks like this, with optional elements enclosed in square brackets:

aplay [-d <device>] [-t <time>] [-g <gain>] [-f] [-c] [-b] [-l] [<file>]

aplay supports a number of command line options; some of them are summarized here.

-d *device* Specifies which audio device to play the sound file through. If not specified, aplay defaults to the first device that is not connected to the telephone. This is usually correct, because the first non-telephone device is usually connected to the local loudspeaker. In the current implementation, the audio device is what specifies the sample rate, number of channels, and encoding. The manual page for each AudioFile server explains what devices exist.

-t *time* Specifies how far in the future the sound will start to play relative to the current device time. A positive value of *time* will begin playing *time* seconds in the future. If *time* is negative, time seconds of sound data will be thrown away. The default is 0.1 seconds.

Incidentally, if one desires to play only a portion of a sound file, existing utilities such as dd(1) can be used. For example, if the sampling rate were 8000 Hz, the following command would skip the first second of sound, then play the next two seconds, then stop.

```
dd if=sound-file bs=8000 skip=1 count=2 bs=1000 | aplay
```

-g *gain* A gain in dB can be used to attenuate or amplify the sound data prior to mixing in the audio server. This permits relative gain adjustments without changing the server global gain controls and can be used to correct for low or high recording levels in the sound file.

-f This switch turns on flush mode. Normally, aplay will exit several seconds before the last sound is played, because of buffering in the server. This switch forces aplay to wait until the last sound has been played before exiting. This is very useful when writing shell scripts. For example, "aplay **-f** sound-file" followed by "arecord".

For some data types, the sound file may be in big-endian or little-endian format. Normally, aplay assumes the format of the client machine. Two options, **-b** and **-l** explicitly specify a big-endian or little-endian input format. The server converts to its byte order as appropriate.

There are several opportunities for enhancements to aplay. We have already mentioned that it would be desirable for aplay to understand and interpret a variety of popular self-describing sound file formats. In addition, it would be interesting and straightforward to add the capability for aplay to begin playback at a specified absolute time, related to the wall clock, rather than simply a relative time related to the moment aplay begins to execute. This capability would make it possible to synchronize several instances of aplay.

8.1.2 aplay Implementation

In this section, we take a look the code of aplay. The truth is somewhat more complicated than we present here, but not much. Readers who wish all the details should read the sources, which are included in the AudioFile distribution.

```
int flushflag = 0;                  /* set from command line   */

AFAudioConn *aud;                   /* connection to AF server */
AC ac;                              /* audio context           */
AFSetACAttributes attributes;       /* AC attributes record    */
ATime t, act, nact;                 /* Time                    */

aplay()
{
    attributes.play_gain = gain;    /* set from command line   */
    attributes.endian = endian;     /* set from command line   */

    /* open a connection to the audio server specified by
       the AUDIOFILE environment variable */

    AoD ( (aud = AFOpenAudioConn("")) != NULL,
        "%s: can't open connection.\n", argv[0]);
```

FindDefaultDevice() is not reproduced here. Its job is to locate the lowest numbered audio device that is not connected to the telephone. This will usually be the local audio device. (FindDefaultDevice() is not used if the **-d** command line switch is given).

```
device = FindDefaultDevice(aud);

/* set up audio context, possibly setting the gain and endian-ness */

ac = AFCreateAC(aud, device, (ACPlayGain | ACEndian), &attributes);
```

At this writing, important properties of the audio device, such as sampling rate, number of channels, and encoding type occupy fields in the device data structure. There should be standard access procedures or macros for these fields, but we have not yet implemented them.

```
/* extract properties of the device */
srate = ac->device->playSampleFreq;      /* sample rate        */
type = ac->device->playBufType;          /* encoding type      */
channels = ac->device->playNchannels;    /* number of channels */
ssize = AF_sample_sizes[type]/8;         /* bytes per sample   */
ssize *= channels;

/* allocate play buffer */
AoD((buf = malloc(BUFSIZE*ssize)) != NULL,
    "Couldn't allocate play buffer\n");

/* pre-read the first buffer-full from the file */
if((nbytes = read(fd, buf, BUFSIZE*ssize)) <= 0)
    exit(0);
```

It is not logically necessary to pre-read the first file block, but doing so avoids putting the latency of the file read between the call to AFGetTime() and the first call to AFPlaySamples().

The following section is the inner loop of aplay. It establishes the current playback time on the server, and schedules the exact server time for playback of the first block of audio. Thereafter, it schedules each successive block to play directly on the heels of the the previous block, so that playback will be uninterrupted and continuous. After each call to AFPlaySamples(), the time pointer is simply incremented by the number of samples played.

```
/* establish an initial AF server time */
t = AFGetTime(ac);
```

```
    /* schedule the initial playback for a short time in the future.
       toffset can be set from the command line */
    t = t + (toffset * srate);

    do {
        /* send samples to the server */
        nact = AFPlaySamples(ac, t, nbytes, buf);

        /* figure how many samples we read from the file,
           and schedule the next block to start after this one */
        nsamples = nbytes / ssize;
        t = t + nsamples;

        /* At this point, the buffers in the AF server hold
           the samples from time nact to time t */

    } while ( (nbytes = read(fd, buf, BUFSIZE*ssize)) > 0);
```

At this point, we've finished reading the file, and sent all the data to the server, but a lot of it has not yet been played out. If the command line specified **-f**, we now wait until the server finishes the playback. sleep(1) is a sloppy way to do this, but it is easy to program.

```
    if (flushflag) {
        while (((int) AFGetTime(ac)) - ((int) t)) < 0)
                sleep(1);
    }
    /* we're done!  Just abandon the connection to the server, it will
       be cleaned up automatically. */
}
```

The only substantive code omitted above is the mechanism which allows aplay to respond immediately to a control-C or interrupt signal. Without special handling, the signal would cause aplay to exit immediately, but the buffered audio in the server would continue to play for several seconds. For control-C to cause aplay to immediately halt, special handling is necessary. aplay sets up a signal handler for SIGINT, that sets a flag. Each time around the main playback loop, aplay checks the flag and if it is set, aplay breaks out of the loop, as though it had reached end-of-file. If the interrupt flag is set on loop exit, then aplay does the following:

```
    if (int_flag)
    {
      /* fill the playback buffer with ``silence'' according to
         the encoding type */
      AFSilence(ac->device->playBufType, buf, BUFSIZE*ssize);

      /* turn on preemptive playback */
```

```
        attributes.preempt = 1;
        AFChangeACAttributes(ac, ACPreemption, &attributes);

        /* erase the buffered audio still held in the server, by
           writing preemptive silence over top of it.  This needs to
           be done for the time interval between time nact, ''now'' and
           time t, the time farthest in the future for which the
           server is holding buffered audio. */

        while (nact < t)
          {
            act = AFPlaySamples(ac, nact, nsamples*ssize, buf);
            nact += nsamples;
          }
      }
```

This code fragment is interesting because it illustrates how explicit client control of time allows aplay to take full advantage of all the buffering capacity of the server during normal operation — insulating aplay from most real-time issues, yet still allows it to stop "on a dime" when necessary, by erasing the remaining buffered audio. [27]

8.1.3 Flow Control

Note that there is no explicit code in aplay for flow control. aplay merely reads from its input with fread() and writes to the audio server with AFPlaySamples(). There is a fundamental, but unwritten, assumption in aplay that the file system is fast enough to supply audio data faster than it is required by the server. Assuming that this is so, aplay will copy data from the input to the AudioFile server at a speed limited only by file system performance and the performance of the transport protocol to the server. The audio data will be buffered in the server, until aplay gets about four seconds ahead of real-time. At that point, the server connection (AFPlaySamples()) will block, providing flow control. Once the server buffers are full, successive calls to AFPlaySamples() will return at intervals given by the block size. This mechanism of providing flow control means that the file I/O side of aplay could block for as long as four seconds before there would be a break in the smooth playback of audio. Server buffering also gives rise to the need for the **-f** flush flag and the interrupt code in aplay. The flush flag causes aplay to wait until the audio is all played out before exiting, while the interrupt code in aplay

[27] Actually, this is still not quite right. If aplay is running concurrently with other clients, the preemptive playback will erase all the other clients' sound as well as that buffered by this instance of aplay.

actually erases the "future" audio buffered in the server, so that playback halts as soon as aplay is interrupted.

8.2 arecord — An Record Client

arecord and aplay are complementary programs. arecord reads samples from the audio server and writes the data to file, or to standard output if a file is not specified. The sampling rate, encoding format, and number of channels are all specified indirectly by the AudioFile server device. arecord always connects to the server specified in the AUDIOFILE environment variable.

8.2.1 arecord options

The command line for arecord is as follows, with optional elements enclosed in square brackets:

arecord [-d <device>] [-l <length>] [-t <time>] [-silentlevel <level (dB)>]
[-silenttime <time>] [-printpower] [-b] [-l] [<file>]

arecord has a number of command line switches which improve its flexibility. The following paragraphs describe these options in more detail.

-d *device* Specifies from which audio device to record. If not specified, arecord defaults to the first device that is not connected to the telephone. The sample rate and recording format are specified indirectly by the device selection.

arecord offers three methods of halting the record. It will record indefinitely if no option is specified. It can record for a specific length of time, via the **-l** switch, or it can record until the input appears to be silent (via the **-silentlevel** and **-silenttime** switches).

-l *length* length of sound data to record, specified in seconds.

-silentlevel *level* level (in dBm) below which the sound is deemed to be silent. The default value is -60. The 0 dBm reference level is the "digital milliwatt".

-silenttime *time* time (in seconds) of silence which will terminate the recording. The default value is 3.0.

If either **-silentlevel** or **-silenttime** is set, then arecord will terminate recording after so many seconds of "silence".

-printpower Print input power level in dBm on the standard error output every block (8 times a second). This is a debugging option, but it may be useful in figuring out proper values for the **-silentlevel** option. arecord | apower will accomplish a similar function.

-t *time* time can be used to adjust the audio device time at which the the arecord client begins to record the sound data. A positive value of time will begin recording time seconds in the future. If time is negative, sound data will be returned from time seconds in the past. Generally, the AudioFile server is always recording, and keeps the past four seconds in buffers. Thus arecord -t -2 will start recording two seconds earlier than the time arecord begins to execute. If the time offset is too early (beyond the server buffering capacity), silence will be returned. The defaults is 0.125 seconds.

file arecord writes data to *file* in the current working directory. If the file name is not specified, then arecord writes the audio data to standard output.

Like aplay, arecord defaults to the byte order of the client machine. The **-b** and **-l** options will explicitly set a big-endian or little-endian output format.

8.2.2 arecord implementation

This section discusses the implementation of arecord. The initialization of the server connection are essentially the same as described for aplay, so those details are not shown below. The reader is encouraged to study the full source code for arecord, which is distributed with AudioFile.

```
arecord()
{
    /* open a connection to the audio server and device specified. */
       . . .

    /* If the user specifies the number of seconds to record, convert
       the length into a number of samples to record. */
    if(length >= 0) nsamples = srate * length;
```

```
    /* establish an initial AF server time, and schedule the initial
       record request according to toffset.  toffset can be set on the
       command line */
    t = AFGetTime(ac) + (toffset * srate);

    while (nsamples > 0) {

        /* If we are recording the last block of a timed record, then
           the request will be shorter than BUFSIZE */
        int nb = (nsamples > BUFSIZE) ? BUFSIZE : nsamples;

        /* Record nb samples at server time t */
        AFRecordSamples(ac, t, nb * ssize, buf, ABlock);

        /* advance the time pointer by the size of the current block */
        t += nb;

        /* decrement the samples to go (for a timed record) */
        nsamples -= nb;

        /* write the samples on the output stream */
        fwrite(buf, ssize, nb, f);
        fflush(f);
```

The fflush() operation is not strictly necessary, but if arecord is used in a pipeline leading into some real-time application, then we do not want to introduce any excess latency.

If recording is to be terminated by silence, then the following code is active. The program waits for a run of blocks of total length silent_time, each of which has a power level below silent_level.

```
        if (silent_level_flag)
        {
            /* compute the power of the block */
            pow = power(buf, nb);

            if (pow < silent_level)
                silent_run += (nb / ((double) srate));

            /* break will exit the record loop */
            if (silent_run >= silent_time) break;
            if (pow > silent_level) silent_run = 0.0;
        }
    }
}
```

8.2.3 Flow Control

In arecord, flow control is provided by the server. In normal operation, each call to AFRecordSamples() requests a block beginning slightly in the past and extending into the future. The server blocks the call until the requested segment is completely recorded, and the call returns to the client slightly after the time corresponding to the end of the block.

One very interesting property of the AudioFile System is that the server is always listening. It is possible for AFRecordSamples() to request data from the recent past (typically within four seconds). In this case, the call is fulfilled from the server buffers and returns to the client right away. This capability permits voice applications to omit the usual beep that means it is OK to start talking. Instead, the user can invoke arecord with a small negative offset, and recording will start "before" arecord begins execution.

8.3 apass — Copy From One Server to Another

apass is an AudioFile client which records from a device attached to one AudioFile server and, after a small delay, plays back on a device attached to another server.

One of our primary goals in the development of the AudioFile System was to enable experiments in teleconferencing. apass is not a teleconferencing application, but it addresses some of the fundamental problems of network teleconferencing: communications with multiple audio servers, management of end-to-end delay, and management of multiple clock domains.

It is possible to record from one audio server and to route the audio to an output device on the same or on a different audio server by piping arecord into aplay. (If you have an AudioFile environment, try this). However, this is not a satisfactory solution for several reasons:

- In teleconferencing, it is important to have tight control over the end-to-end delay of the audio connection. If, for example, the round trip delay is over about 300 milliseconds, then humans begin to have difficulty with conversational dynamics. apass sets up a strict delay budget, accounting for the various factors involved.

- In a system with multiple audio devices, frequently the different devices will be controlled by different sampling rate clocks. Even though both clocks nominally run at the same rate, the physical implementations are subject to slight frequency errors. For example, crystal oscillators have tolerances of perhaps 100 parts per million and they vary slightly with temperature. If the transmitting end samples faster than the receiving end, then the excess samples will accumulate in buffers in between. This accumulation will manifest itself as gradually increasing end-to-end delay. If the transmit clock is slower than the receive clock, the buffers will run dry and the playback sound will be broken up. apass tracks the transmit and receive clock rates and resynchronizes as necessary.

apass operates by reading blocks of samples from the transmit server, one after another in real-time, and scheduling their playback on the receive server. The overall delay between input and output is made up of three components:

- Packetization delay. Since apass deals in blocks of samples, the last sample of a block must be recorded at the transmit end before the first sample can be played back. Thus the size of the block sets a minimal value of the end-to-end delay. This component of the overall delay is called the packetization delay and is constant.
- Transport delay. The blocks of samples are sent from the transmitting server to apass, and from apass to the receiving server. The associated transmission delay, plus all software overhead and rescheduling delays make up the transport delay. This component of the overall delay is variable.
- Anti-jitter delay. apass inserts extra delay at the receiving AudioFile server by scheduling playback for a point in the near future, rather than as soon as possible. This is possible because AudioFile permits explicit control over playback time. The delay at the receiver serves to absorb variation in the transport delay, provided that the variation in transport delay is not larger than the anti-jitter delay.

Any additional end-to-end delay specified by the user is allocated to additional anti-jitter delay.

8.3.1 apass Options

Some command line options to apass serve mundane purposes — the specification of input and output AudioFile servers and the input and output devices. The more interesting switches allow the user to specify the end-to-end delay, buffer size, and anti-jitter delays.

The command line for apass is as follows, with optional elements enclosed in square brackets.

apass [-ia <input-AF-server>] [-oa <output-AF-server>] [-id <input-device>] [-od <output-device>] [-delay <seconds>] [-aj <anti-jitter-seconds>] [-buffering <buffering-seconds>] [-gain <dB-gain>] [-log] [-f <parameter-file>]

If no options are given, apass will "loop back" the first non-telephone audio device connected to the server specified by the AUDIOFILE environment variable.

The various options are discussed below:

-ia *server* Specifies which audio server to record the sound from. Defaults to the value of the AUDIOFILE environment variable.

-oa *server* Specifies which audio server to play the sound to. Defaults to the value of the AUDIOFILE environment variable.

-id *device* Specifies which audio device on the input server to record the sound from. Defaults to the first device that is not connected to the telephone, which is often the local microphone device.

-od *device* Specifies which audio device on the output server to play the sound to. Defaults to the first device that is not connected to the telephone, which is often the local speaker device.

-delay *seconds* Sets the record to playback delay. The default value is 0.3 seconds. This delay is made up of three components: packetization, transport, and anti-jitter. The minimum value of this parameter is buffering+aj and the maximum is 3.0 seconds.

-aj *seconds* Sets the tolerance for clock drift between the input and the output. If the input to output delay drifts from its nominal value by more than this amount, the delay will be resynchronized, probably resulting in an audible blip. The default value is 0.1 seconds. Legal values are 0 to 1 second.

-buffering *seconds* This parameter sets the amount of audio read from the input and written to the output as a single operation. It sets a minimum value for delay. The default value is 0.2 seconds. Legal values are 0.1 to 0.5 seconds.

-gain *dB-gain* Controls the playback gain. The default value is 0 dB. Legal values are from -30 to +30 dB.

-log If set, apass will print a message on standard output whenever it is necessary to resynchronize clocks between input and output and whenever the record side of the program takes longer than 400 milliseconds.

-f *file* Whenever a SIGUSR1 is received, apass will read *file* to acquire parameters. The parameter file should contain one or more lines. Each value should have a keyword and a value. Legal keywords are delay, buffering, aj, and gain.

A typical parameter file might contain:

```
delay 0.3
buffering 0.2
aj 0.1
gain 0.0
```

The **-f** option allows another process to control apass. For example, a Tk program or EMACS keybindings could alter the behavior of apass. This permits a multi-process but single-threaded environment to act like a multi-threaded environment. This feature permits the user to experiment with different delay configurations without restarting the application.

8.3.2 apass Implementation

This section describes the inner loop of the apass application. Many details and error checks are omitted. The interested reader can refer to the apass.c source module, which is included in the AudioFile distribution.

```
int delay_in_samples;         /* nominal delay except packetization */
int delay_upper_limit;        /* nominal delay + aj                 */
int delay_lower_limit;        /* nominal delay - aj                 */
float delay = 0.2;            /* seconds delay from input to output */
float aj = 0.1;               /* anti-jitter tolerance              */
float time_bufsize = 0.1;     /* data buffer, measured in seconds   */
```

```
int samples_bufsize;          /* data buffer, measured in samples   */

#define SLIPHIST 4
int slip, sliphist[SLIPHIST], nextslip;  /* recent delay values */

apass()
{
    /* open connections to the from and to audio servers */
    faud = AFOpenAudioConn(faf);
    taud = AFOpenAudioConn(taf);

    /* set up audio contexts, find sample size and sample rate */
    fac = AFCreateAC(faud, fdevice, ACRecordGain, &attributes);
    fsrate = fac->device->playSampleFreq;
    fssize = sample_sizes[fac->device->playBufType] *
      fac->device->playNchannels;

    tac = AFCreateAC(taud, tdevice, ACPlayGain, &attributes);

    /* establish a value for the delay from record to playback */
    delay_in_samples = fsrate * delay_in_seconds;

    /* get starting times for the two servers */
    ft = AFGetTime(fac);

    /* playback will start delay_in_samples in the future */
    tt = AFGetTime(tac) + delay_in_samples;

    for (;;) {
        /* record samples from the source server */
        factt = AFRecordSamples(fac, ft, samples_bufsize*fssize, buf, ABlock);
        /* play them back on the sink server */
        tactt = AFPlaySamples(tac, tt, samples_bufsize*fssize, buf);
```

Note that AFRecordSamples() and AFPlaySamples() accept the parameters ft (from-time) and tt (to-time) respectively, and return factt (from-actual-time) and tactt (to-actual-time). In apass, factt will be approximately equal to ft+samples_bufsize, because the pacing flow control of apass is provided by the source AudioFile server. The full implementation checks for this, but that test is not included in this abbreviated version.

Time tt should be about delay_in_samples in the future relative to time tact (now). The exact value of this difference is an instantaneous estimate of the current end-to-end delay minus packetization and transport delays. apass averages four consecutive values of this delay in order to compute "slip". apass then checks to see if slip is within a range specified by a nominal delay plus or minus the anti-jitter specification. If the receive clock is faster than the transmit clock, slip will eventually drift below the lower end of the range. If the receive clock is slower than the transmit clock, then slip will eventually drift above the upper end of the allowable range. In either case, tt is reset to exactly the nominal delay, resynchronizing the connection.

```
        /* Record the delay in the circular history buffer. */
        sliphist[nextslip++] = tt - tactt;
        if (nextslip >= SLIPHIST) nextslip = 0;

        /* compute an average of the recent delays */
        slip = 0;
        for (i = 0; i < SLIPHIST; i += 1) slip += sliphist[i];
        slip /= SLIPHIST;

        /* if the actual delay has drifted outside of the allowable
           region, then resynchronize the connection */
        if ((slip < delay_lower_limit) || (slip >= delay_upper_limit))
            tt = tactt + delay_in_samples;

        /* finally, update the start time of the next block */
        ft += samples_bufsize;
        tt += samples_bufsize;
    }
}
```

8.3.3 Discussion

apass uses the simplest imaginable algorithm for handling clock drift. It simply resynchronizes the connection whenever the delay leaves a tolerance band. There is much room for more complicated algorithms — for example, the connection could be resynchronized whenever the audio is quiet, or apass could use digital signal processing to interpolate the digital audio at the receive sample rate. A simple enhancement would be for apass to perform some simple averaging of the waveform at the point of resynchronization. This would tend to reduce the audible blip caused by a waveform discontinuity.

The reader should note that time values from the two audio servers cannot be directly compared, because they have different initial values and slightly different rates. Instead, one must compare differences. A calculation like $(\mathsf{ft2} - \mathsf{ft1})/(\mathsf{tt2} - \mathsf{tt1})$ estimates the ratio between the two clock rates, provided that time "2" and time "1" are sampled at the same time according to a third clock. apass avoids such calculations by instead allowing the overall flow control to be set by the transmitting audio server. Then apass judges the necessity of resynchronization by tracking the buffering available at the receiving server.

8.4 Telephone Control

AudioFile supplies a number of core clients for control of a telephone connection:

- ahs provides hookswitch control. "ahs off" will take the telephone off-hook, either answering a call or beginning the process of placing a call. "ahs on" places the telephone back on hook, terminating a call.
- aevents is a general-purpose application for printing events generated by the AudioFile server. Most AudioFile events are generated by telephone devices. aevents also has options to count rings.
- aphone is a core client which dials the telephone. aphone uses the AFDialPhone() library procedure to digitally synthesize the DTMF tones generated by pushbutton telephones.

8.5 Miscellaneous Clients

Besides the clients discussed so far, AudioFile is distributed with a few miscellaneous clients:

- aset is a general-purpose device control application
- ahost allows the user to add or delete hosts from the list of machines that are allowed to make connections to the server. This provides a rudimentary form of privacy control and security.
- alsatoms displays a list of atoms defined by the server.
- aprop displays properties attached to AudioFile devices and can track changes to those properties.

8.6 A Trivial Answering Machine

The core clients can be used to construct interesting applications. This section shows a very simple answering machine application as a shell script connecting various core client applications. This is a particularly simple example, because the sequence of actions for an answering machine are fixed.

The general idea is to use the core clients in a strict sequence, to wait for the phone to ring, to answer it, to play the outgoing message, to record the incoming message, and to hang up the phone.

```
#!/bin/sh
#
#  loop forever
#
    while true; do
#
# wait for the phone to ring three times
#
    aevents -ringcount 3.0
#
# answer the phone using LoFi
#
    ahs off
#
# play the outgoing message, then a beep from the effects library
#
    aplay -f -d 0 outgoing_message.snd
    aplay -f -d 0 beep.snd
#
# record up to 30 seconds, or until the caller stops talking
#
    arecord -silentlevel -35.0 \
        -d 0 -silenttime 4.0 -l 30.0 -t -1.0 >>messages.snd
#
# play a thank-you message, then hang up the phone
#
    aplay -f -d 0 thanks.snd
    ahs on
#
# add a date stamp to the message file using a text to speech
#   synthesizer (not part of AudioFile ...)
#
    date | tts >>messages.snd
    mail `whoami` -s "New voice mail received" </dev/null
#
# done!  Go back and get the next message
#
    done
#
```

9 Contributed Clients

The AudioFile contributed clients include a wide variety of things, and users are encouraged to browse the manual pages in the AudioFile distribution to see what is there. A sampling of the contributed clients are discussed below.

9.1 abob — A Tk Demonstration

abob is a very simple multimedia application, combining audio with scanned image and on-line help facilities. abob has a Tk user interface which allows the user to play a prerecorded sound clip and to display an image of the performer. abob also includes a device selection menu, gain control, and help screens. Figure 6 shows the main abob window together with a tear-off menu for device selection. The Tk toolkit makes this sort of application very easy to implement.

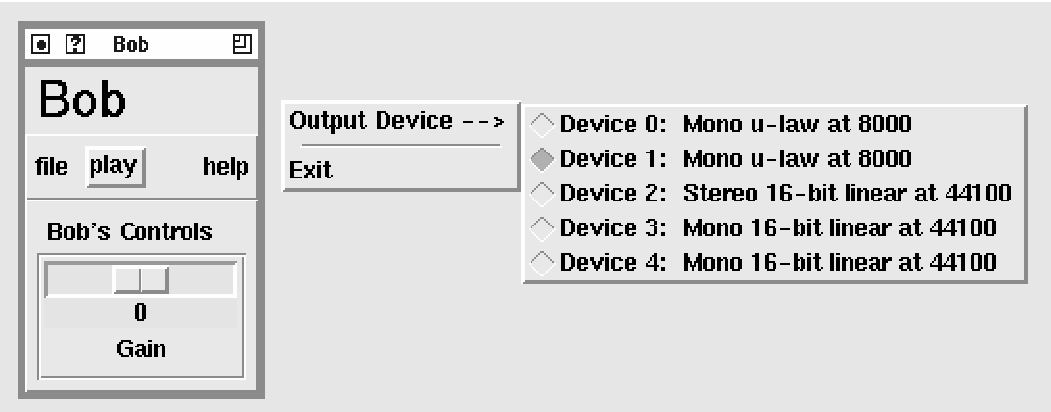


Figure 6: abob client

9.2 adial — A Screen-based Telephone Dialer

adial is a simple Tk based telephone dialer. It uses the Tcl exec command to run the standard core client programs ahs, aset, and aphone to control the hookswitch, talking path, and dialing.

adial will dial a number entered in the entry field. If a name is entered instead of a number, adial will attempt to translate the name to a number through a personal phone directory stored in the file .phonelist in your home directory.

If the entry field of adial is blank, adial uses a Tk facility to retrieve the current X Window System primary selection. In this way, adial can be used to establish a telephone connection to anyone whose name or number is anywhere on the screen.

Figure 7: adial client

9.3 Device Control

axset and afxctl are interfaces to the device control capabilities of an AudioFile server, based on the Tk and Xt toolkits, respectively. axset also illustrates the power of the Tk toolkit, because it automatically adds controls for whatever audio devices exist on the given server. The preferred application for device control is axset, but afxctl has an event history log while axset does not.

9.4 xpow — Display Signal Power

xpow, shown in Figure 8, is a simple client that displays a calibrated line chart of signal power. It can be used to help judge proper recording level.

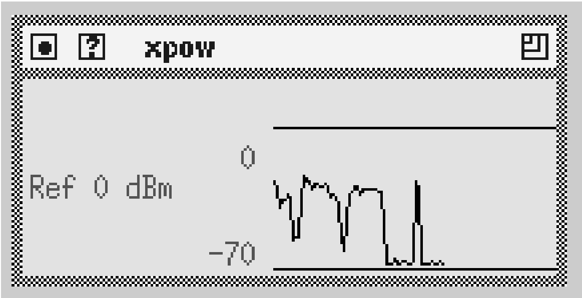


Figure 8: xpow client

9.5 afft — A Real-time Spectrogram Displayer

afft accepts audio data from one of several sources, executes a running Fourier transform on the data, and displays the transform result. The display is updated continuously in either "waterfall" or "spectrogram" format. Figure 9 shows afft in waterfall mode.

afft is initially configured through command line switches. Many of the parameters can be changed while the program is running via the graphical user interface, as described below.

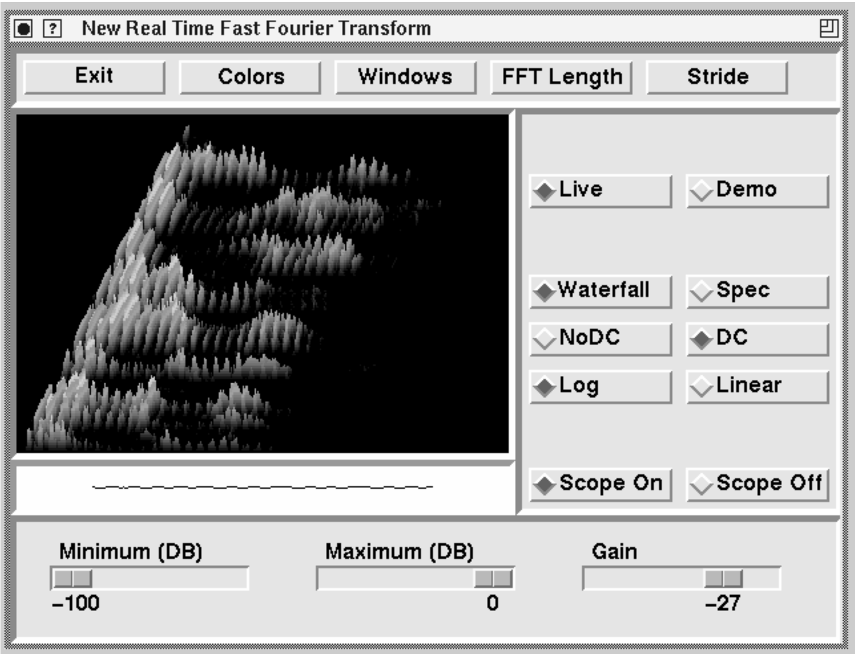


Figure 9: afft client

The core of afft is a C program which reads audio samples from a file, from the standard input, or from a server in real-time. afft windows the data using a selectable window function, then performs a Fourier transform on the data. The resulting spectrogram is presented in an X Window System display in either the waterfall or spectrogram format. An optional on-screen oscilloscope display shows the actual waveform.

The C core of afft is surrounded by a Tk-based graphical interface, which allows the user to alter a number of parameters:

- Display colors. The amplitude of the spectral information can be presented in gray scale, blue through orange, or the traditional blue through red "spectral" colors.

- Window functions. The data can be windowed by Hamming, Hanning, or triangular windows, or the windowing can be disabled.
- FFT length. The tradeoff between time and frequency resolution can be altered by changing the transform block size in power of two steps from 64 to 512 samples.
- FFT stride. The poor effects of a large transform block on time resolution can be ameliorated by overlapping the transforms of adjacent blocks. The overlap can be adjusted from 64 to 512 samples.
- Log scale. The display can be presented in linear or logarithmic form.
- Display. The user is given the choice of waterfall or spectrogram displays.
- Live vs. demo. A built-in swept frequency sine wave is displayed when afft is put into "demo" mode.
- Sliders. Slider controls govern the display gain, and permit the user to compress the power levels shown by the display.

The command line for afft is shown below. Full details are in the manual page.

afft [-color] [-d <device>] [-file <file>] [-gain <gainvalue>] [-length <fftlength>] [-log] [-min <minvalue>] [-max <maxvalue>] [-nodc] [-noscope] [-nowindow] [-realtime] [-sine] [-spec] [-stride <stridelength>]

Most of the command line switches are also available at runtime, but *-file* and *-realtime* must be specified on the command line.

The source of the audio that is displayed depends on the command line as follows: If the *-file* switch is given, then the audio source is a file of μ-law samples. Otherwise, if the *-sine* switch is given, the audio source is a "canned" sine wave that sweeps up and down the frequency spectrum. If neither *-file* or *-sine* are given, then afft takes its audio data from the audio server local input.

When input is taken from a file, afft will continuously loop through the file from beginning to end, rewind the file, and repeat. If file is given as "-", input is taken from standard input. Since it is not possible to rewind standard input, afft will terminate on end-of-file in this case.

If afft can connect to the audio server, then the file is played through the audio output device in synchronization with the fft display.

If the *-realtime* switch is given, afft attempts to stay synchronized with the audio server in real-time. If afft cannot get enough CPU cycles to keep up with the incoming audio stream, it may fall behind enough that it no longer captures valid audio data. If the switch is not given, afft will discard audio samples in order to keep up with real-time.

9.6 Miscellaneous Contributed Clients

Some of the other contributed clients in the AudioFile distribution include:

- abiff is the audio analog to the Berkeley UNIX application biff. abiff uses our DECtalk text to speech synthesizer to announce the from and subject field of arriving electronic mail.

- radio is a network unidirectional multicast system. An application at the transmitting end, radio_mcast, transmits audio using Ethernet multicast. Many users can then run the receiving program, radio_recv, to listen in to a multipoint broadcast. We have used these programs, for example, to relay radio broadcasts into regions of our building where ordinary radio reception is poor.

- abrowse and xplay are Tk and Xt toolkit, respectively, applications for browsing and playing directories of audio sound files.

- apower and atone are standard I/O-based signal processing utilities. apower calculates μ-law signal power relative either to the CCITT "digital milliwatt" or to a sine wave 3.16 dB below the digital clipping level. atone is a μ-law signal generator that will create a specified frequency and power level sine wave. "atone | aplay" is a useful technique for setting playback levels.

9.7 Other AudioFile Applications

There are already a number of applications which use AudioFile but are not distributed with it. We mention some interesting examples here.

9.7.1 Speech Synthesis

We built a software-only version of the DECtalk text-to-speech synthesizer[5] which generates output via the AudioFile system.

The DECtalk synthesizer is a three stage process: letter to sound translation, phonemic synthesizer, and the vocal tract model. Of these, the vocal tract model, which generates the output digital waveform, consumes about 95% of the synthesizer's CPU time. However, the entire system uses only about 37% of the processing power of the DECstation 5000 Model 200, which is based on a 25 MHz R3000 MIPS CPU.

9.7.2 DECspin

DECspin is a network audio and video teleconferencing product produced by the Digital Equipment Corporation. DECspin uses AudioFile to provide audio teleconferencing facilities. An audio-only version of DECspin is generally available for public FTP.

9.7.3 ARGOSEE

A group at Digital's Systems Research Center in Palo Alto, California is exploring teleconferencing and collaborative work, using, among other things, AudioFile.

9.7.4 VAT

A team at the University of California, led by Van Jacobson, has built a network teleconferencing application using IP multicast protocols. VAT can use AudioFile for its audio I/O.

10 Performance Results

In this section, we present some performance results for our implementation of the AudioFile System. First, we measure the time to complete client library operations. Next, we measure the CPU load for recording and playback. Finally, we discuss our experience using TCP as the transport protocol.

10.1 Server and Client Performance

We measured latencies and performance of our AudioFile implementation by the timing various client library functions. We tested with two types of systems (MIPS and Alpha) under six local and networked configurations:

`alpha`	Alpha local client & server
`alpha/mips`	Alpha client, MIPS server
`alpha/alpha`	Alpha networked client & server
`mips`	MIPS local client & server
`mips/mips`	MIPS networked client & server
`mips/alpha`	MIPS client, Alpha server

The testing environment was as follows:

- All testing was done with the LoFi server, Alofi, with a CODEC (8 KHz) device.
- All MIPS systems were DECstation 5000/200s running ULTRIX 4.3. All Alpha systems were DECstation 3000/400s running DEC OSF/1 for Alpha AXP V1.2.
- All network testing took place on a lightly loaded Ethernet (10 Mbit/sec).
- Unless stated otherwise, all functions were timed by measuring the time to complete 1000 iterations, then computing the average time per iteration.

10.1.1 Basic Latency

The library function AFGetTime() is a good baseline case for measuring the time to process AudioFile functions because it incurs minimal processing on the server and client side. [28] Figure 10 shows the time required for a call to AFGetTime() for the different configurations.

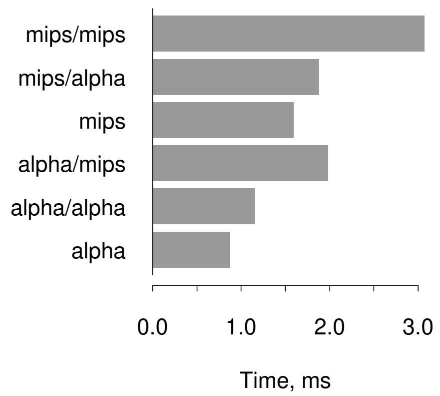


Figure 10: AFGetTime() function timings

This example shows the latency through the operating system network code and over the wire. Most of this overhead is spent in the operating system and network driver: the actual network latency is negligible. The AFGetTime() function causes an 8 byte request packet to be sent to the server and an 8-byte reply to be returned. Adding the TCP, IP, and Ethernet overheads results in 66-byte request and reply packets. At 10 Megabits/second, these packets spend less than 50 microseconds on the wire.

10.1.2 Play and Record

The AudioFile library functions that move data have latencies that depend on the length of the data. Figure 11 shows the time required to process various length AFRecordSamples() requests on the different system configurations. The record requests were scheduled to hit entirely in the server's record buffer (and not block).

[28] In this version of AudioFile, The no-op function AFNoOp() does not incur a full client-server exchange.

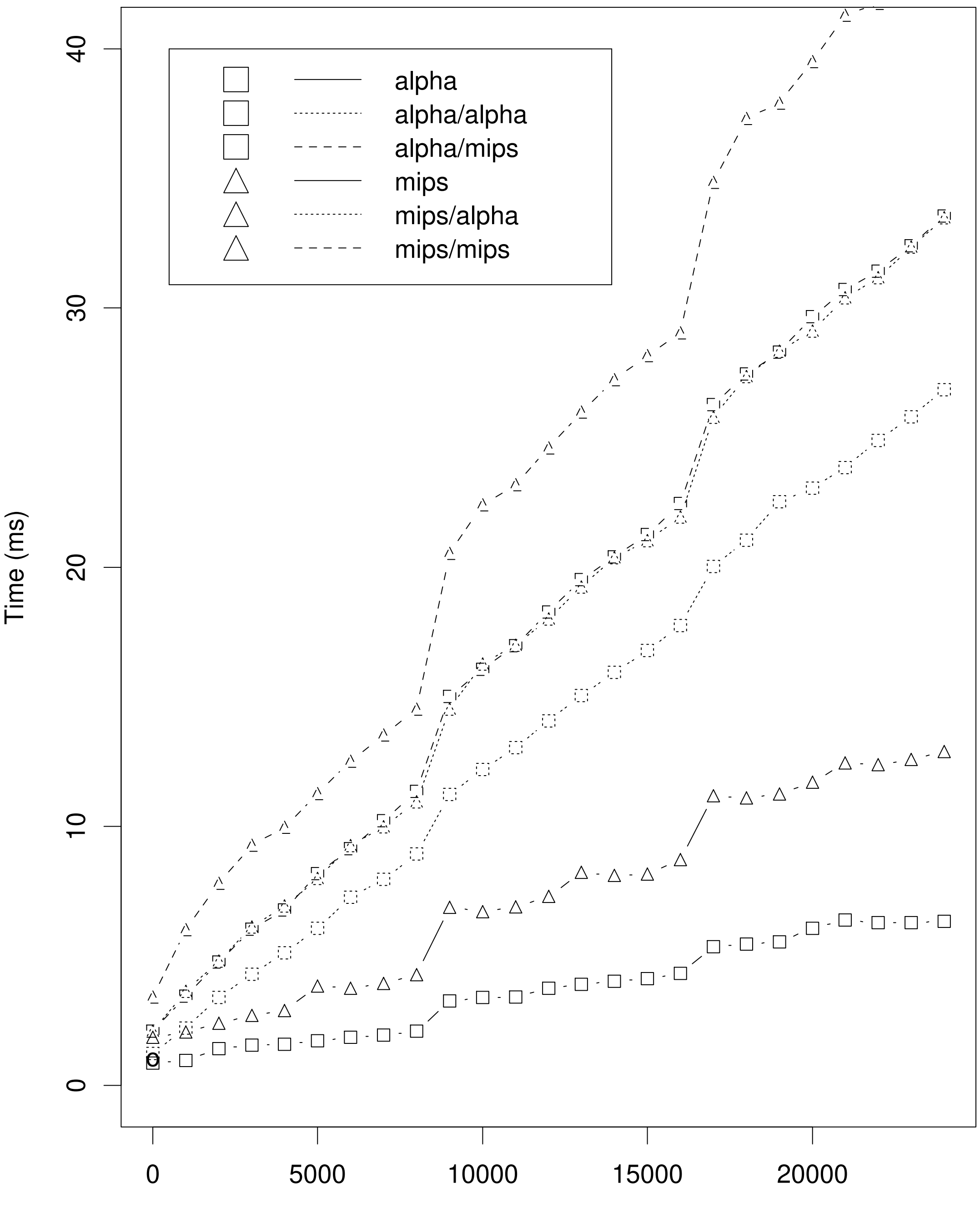


Figure 11: AFRecordSamples() timings

A record request packet is 20 bytes long, and the reply packet is 32 bytes plus the returned samples.

The timing for short requests represents the basic overhead and correlate with the base times for AFGetTime(). The jumps at approximately 8K bytes are due to "chunking" performed in the client library. Requests longer than 8K bytes (not samples) are broken into 8K byte request chunks to better control interactions with the transport protocol heuristics and to simplify the server implementation. Each request completes synchronously—the client library waits for the reply before sending the next chunk. A 16K byte request therefore takes the same time as two independent 8K byte requests.

Is this overhead significant? For high sample rates, it might be. For 16-bit 48 KHz stereo data, an 8K byte request is about 42 milliseconds worth of samples. In the slowest configuration (`mips/mips`), handling an 8K byte request takes about 12 milliseconds, of which 2.5 milliseconds is the basic overhead. This means that our slowest machines are only about three times faster than they need to be. This example assumes that the throughput for the HiFi device (as measured in bytes per second) is the same as the CODEC device's throughput.

The slopes of the lines in Figure 11 give a measure of basic throughput. Table 10 shows the throughput for the six different configurations.

Configuration (client/server)	Throughput (K bytes/sec)
alpha	4400
alpha/alpha	980
alpha/mips	760
mips	2200
mips/alpha	770
mips/mips	580

Table 10: Record throughput

10.1.3 Preempt Play vs Mix Play

The AudioFile play request is very similar to the record case: the request packet is a 20 bytes plus the play data, and the reply packet is 8 bytes. However, the play request can be processed in one of two modes: Mix or Preempt, which may have performance implications for the server. A preemptive play request is usually the fastest, since the data is just copied into the server's play buffers. A mixing play request requires some processing to be done by the server as the new play samples are mixed with the existing samples.

Figure 12 shows the time to complete preemptive play operations of various lengths. In an early implementation, the play performance was nearly identical to record performance because it used the same chunking algorithm. However, we realized that many replies in the play case were unnecessary. We modified the play protocol request to let the client specify if there should be a server reply for the request. We then modified the play chunking code to request (and wait for) the server reply for only the final chunk.

The resulting play timing is a nearly linear function of play request size. The slight jumps at 8K byte multiples are due to the (minimal) request overhead.

Figure 13 shows the timings for mixing play operations, and the cost of mixing by the server is clearly evident. A mixing play operation is always slower than a preemptive play. Table 11 summarizes the throughput for mixing and preemptive play requests.

Configuration	Throughput (K bytes/sec)	
(client/server)	Mixing	Preempt
alpha	2500	5500
alpha/alpha	1000	1100
alpha/mips	660	940
mips	1100	2500
mips/alpha	950	1000
mips/mips	650	830

Table 11: Play throughput

10.1.4 Open Loop Record/Play

The timings of various AudioFile operations have implications for applications that process audio in real-time. Simple applications, such as playing a file, do not really care how long the operations take to complete, as long as the throughput exceeds the audio data rate. However, other applications (such as a conferencing application where audio streams are set up between participants) depend on minimizing the time needed to handle samples.

To illustrate some of the fundamental limits, we coded a loopback test that reads

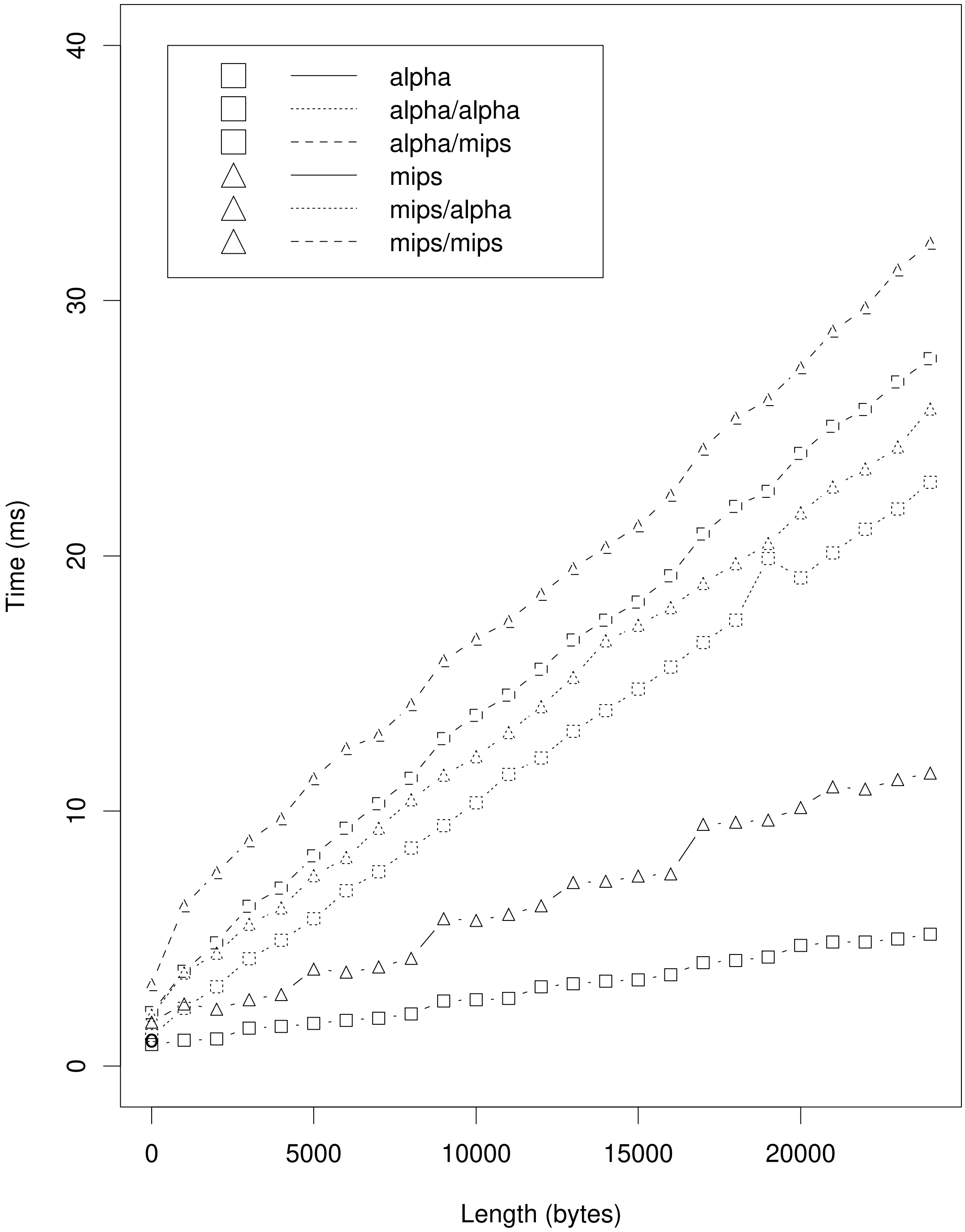


Figure 12: Preemptive AFPlaySamples() timings

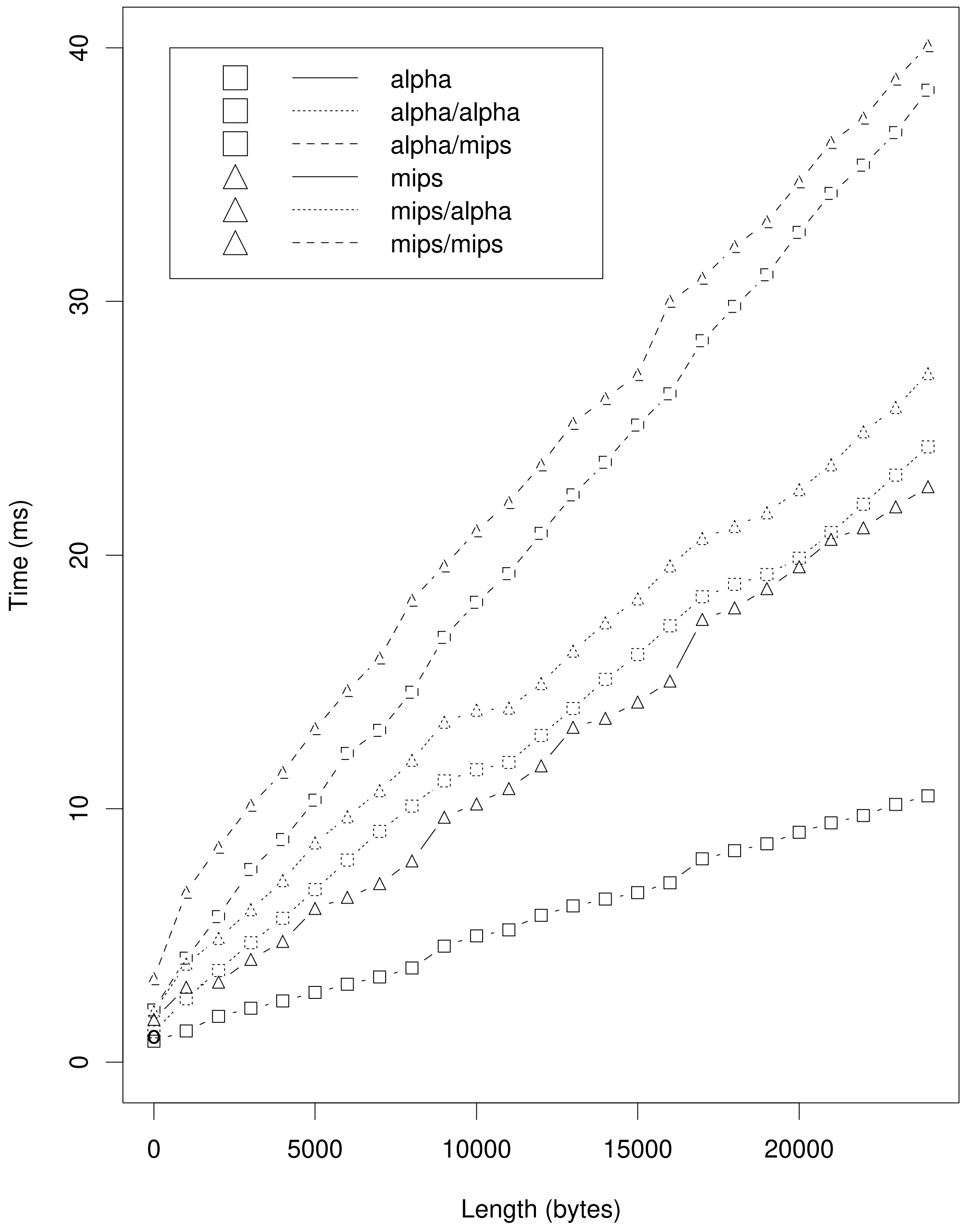


Figure 13: Mixing AFPlaySamples() timings

samples from a device and then writes them back as quickly as possible. The test uses a non-blocking record function that returns only what samples are available. The algorithm is shown in this code fragment:

```
for(;;) {
     now = AFRecordSamples(ac, next, 8000, buffer, ANoBlock);
     length = now - next;
     AFPlaySamples(ac, next+4000, length, buf);
     next = now;
}
```

The rate at which this loop iterates is governed entirely by the AudioFile overhead, and represents a limit for handling real-time audio. The average times to complete one iteration are shown in table 12.

Configuration (client/server)	Time (ms)
alpha	0.87
alpha/alpha	1.27
alpha/mips	2.17
mips	1.93
mips/alpha	2.15
mips/mips	3.45

Table 12: Loopback timing

AudioFile's overhead establishes a minimum latency for real-time applications. However, we believe that AudioFile will be adequate for all but the most demanding real-time requirements. In a networked configuration AudioFile's overhead will be dominated by the network delays. The latency for a link across North America has a minimum 15 millisecond propagation time, not including transmission and routing time.

10.2 CPU Usage

In this section, we investigate the CPU usage for playback and record operations. The test were configured as before, with a local configuration (the client and server running on the same machine using UNIX domain sockets).

The tests consisted of playing and recording 100 seconds of audio at two sample rates and types: 8 KHz μ-law, and 44.1 KHz CD quality stereo. Table 13 summarizes the server and client CPU usage for the two cases. Both user and system times (in seconds) are given. The total time can also be viewed as a percentage load.

		Server		Client		
		User	Sys	User	Sys	Total
Alpha DEC 3000/400	8 KHz playback	0.6	0.5	0.0	0.1	1.2
	8 KHz record	0.3	0.3	0.3	0.3	1.0
	44.1 KHz playback	10.0	7.4	0.1	1.6	19.2
	44.1 KHz record	12.1	3.1	2.3	4.9	22.5
MIPS DEC 5000/200	8 KHz playback	0.6	0.4	0.0	0.4	1.4
	8 KHz record	0.4	0.6	0.0	0.6	1.6
	44.1 KHz playback	5.5	6.5	0.3	9.8	22.1
	44.1 KHz record	13.5	11.9	4.0	10.8	40.2

Table 13: CPU usage

Much of the server time is spent moving samples to and from the audio hardware. The LoFi does not have DMA and must be accessed with programmed I/O.

Table 14 summarizes the times to perform read and write operations to LoFi's shared memory. These timings were obtained by measuring the time to complete ten millionoperations, then computing the average time per operation. These timings were collected with LoFi's DSP disabled. With the DSP enabled, contention for the shared memory could increase these access times by up to 70%.

Reads are expensive because the processor stalls until the read data returns. Writes are fast because they are buffered; they run at nearly the full speed of the option module.

One optimization we have not yet pursued is taking advantage of the Alpha's 64-bit operations. An Alpha-specific version of the server could nearly double the bandwidth to the TURBOchannel by performing 64-bit reads and writes.

System	Operation	Time per op. (ms)
Alpha	read	0.89
	write	0.23
MIPS	read	0.66
	write	0.25

Table 14: Read and write timings for LoFi

10.3 Data Transport

AudioFile can be used over almost any transport protocol, though the details of the protocol may affect real-time audio performance. This section discusses our experience using TCP as the transport layer.

Although applications such as apass may exercise tight control over timing, most do not have strong real-time requirements. TCP is usually sufficient for these applications because the delay caused by retransmission of lost packets is small compared to the buffering of unplayed samples. On the other hand, applications like teleconferencing do require timely delivery of the audio data.

We found that a naively implemented teleconferencing application displayed serious problems when used over a transcontinental TCP link. We observed frequent and lengthy dropouts in the audio stream, which were especially likely with bidirectional data streams. These stem from packet losses caused by a phenomenon known as "ACK-compression" [9, 21], a subtle consequence of the use of window-based flow control. The duration of each dropout is exacerbated by TCP's slow-start algorithm [6], which comes into play when packets are dropped by the network.

ACK-compression occurs when the spacing between acknowledgments is changed by delays in the routers. This can cause cause TCP to send large bursts of packets, which overrun the buffers in a router, causing packets to be dropped. Unfortunately, the TCP slow-start algorithm converts these losses into lengthy recovery periods during which data flows more slowly. On a connection such as a long-haul T1 circuit, it can take several seconds to restore full throughput.

TCP is arguably the wrong transport protocol for applications such as teleconferencing, since it tries to guarantee ordered packet delivery without any concern for packet delay. Many applications instead need guarantees on bandwidth and latency, but they may be prepared to accept some lost data. Networks and protocols that provide such guarantees are active areas of research. To manage these issues, all of the teleconferencing applications mentioned in Section 8 are split among sites, using special protocols over long-haul paths, and only communicate locally with AudioFile servers.

11 Summary

The AudioFile System provides device-independent, network-transparent audio services. With AudioFile, multiple audio applications can run simultaneously, sharing access to the actual audio hardware. Network transparency means that application programs can run on machines scattered throughout the network. Because AudioFile permits applications to be device-independent, applications need not be rewritten to work with new audio hardware.

Development of AudioFile began in 1990 at Digital Equipment Corporation's Cambridge Research Laboratory. In February, 1993, we released a version for public use. It supports both low and high-fidelity audio using a variety of audio devices, and runs on several different computer architectures.

11.1 Areas for Further Work

It is remarkably difficult to get something as big as AudioFile completely right. We are very pleased with the basic design decisions we made, but we do have a list of items which, if fixed or implemented, would make AudioFile still more useful.

- It should be possible for a device to support multiple sample rates and it should be possible to support dynamic changes in sample rate. On the other hand, it would seem like a mistake to add a lot of mechanism to handle the confusion that would result when multiple clients which want to run simultaneously but require different sample rates. In the long run, real-time sample rate conversion may be the answer.

- Audio devices should have an ordered list of supported data formats, so that clients can match against it and so that a device preference for one format over another can be communicated.

- The protocol and library should offer improved support for synchronization and conversion between clocks, including clock prediction routines and the simultaneous reporting of all device clocks. This would aid aggressive applications or those requiring synchronization with other media on the same host.

- The various audio channels supported by a server are assigned integer device numbers arbitrarily. There should be a symbolic way to refer to "the local loudspeaker" or "the telephone". Device sample rate and data types are also useful ways to select devices.

- AudioFile clients which deal with disk files only know about uninterpreted byte streams. There should be support for the various popular sound file formats which automatically specify their data types.
- We would like to add the capability for the server to play and record compressed data types. There is a possibility that the current play and record protocol interfaces are not adequate for complex compressed formats which might include variable rate codes and unusual blocksizes.
- The client library should provide support for sample rate conversion.
- Right now we have separate servers for each hardware device on a particular machine. A single server should be able to support all configured devicesa and should be able to support multiple devices of the same kind as well. This is not a design issue, just implementation.
- We should be able to use real-time services provided by the OS to good advantage. DEC OSF/1, for example, supports the POSIX real-time library.
- We should improve the error handling in the AudioFile libraries, before the present inadequate error handling support becomes too widespread.
- We have built the infrastructure for inter-client communications using properties stored in the server, but the out-of-the-box clients do not yet use the mechanism.
- We need to include support for additional transport protocols which meet the needs of audio services.

11.2 Conclusions

We believe that AudioFile has done well in meeting our design objectives: network transparency, device independence, simultaneous clients, simplicity, and and ease of implementation. We also believe we our experience to date has validated our principles:

- Client control of time. AudioFile permits both real-time and non real-time audio applications using the same primitives. It is much easier to learn one way to do something than to learn two ways.

- No rocket science. Our decision to build on top of standard communications protocols and not to use threads, have improved the portability of the system. It is also arguable that AudioFile performs so well precisely because of its minimalist underpinnings.
- Simplicity. Simple play and record clients require very little code. Indeed, many applications can be constructed using independent AudioFile clients organized by shell scripts.
- Computers are fast. We did not let fear of per-sample processing get in our way. Our slowest implementation platform supports audio mixing playback at telephone quality with less than 2% of the machine per connection.

11.3 How to Get AudioFile

The AudioFile distribution is located at FTP site crl.dec.com (Internet 192.58.206.2) in /pub/DEC/AF. The kit is contained in a compressed tar file named AF2R2.tar.Z. Use anonymous FTP to retrieve the file.

```
% ftp crl.dec.com
   ...
ftp> cd /pub/DEC/AF
ftp> binary
ftp> get AF2R2.tar.Z
```

The kit is shipped as a compressed tar file. To unpack the kit,

```
% cd <audio_root>
% zcat AF2R2.tar.Z | tar xpBf -
```

We also provide a sample kit of stereo sound bites: AF2R2-other.tar.Z. These high-fidelity files should work with Hi-Fi capable AudioFile servers, such as the LoFi and the SGI Indigo versions.

Other files available in this same directory are the release notes, copyright notice, and a README file. Read these first!
We have set up an Internet mailing list for discussions of AudioFile:

```
af@crl.dec.com
```

Send a message to af-request@crl.dec.com to be added to this list.

The Tcl and Tk distributions may be obtained from many FTP sites on the Internet, including sprite.berkeley.edu (128.32.150.27) and gatekeeper.pa.dec.com.

11.4 Acknowledgments

Many people have contributed to AudioFile. We would like to thank Ricky Palmer and Larry Palmer for the SPARCstation DDA code. They, along with Lance Berc and Dave Wecker, persevered as early users of AudioFile. Lance Berc's experiments with long-distance teleconferencing taught us much about the network issues. Jeff Mogul offered valuable assistance on understanding these issues. Guido van Rossum contributed the DDA code for the Silicon Graphics Indigo only two weeks after we released the first public distribution. Dick Beane provided useful comments on drafts of this paper. We would also like to thank Victor Vyssotsky and Mark R. Brown for putting up with us.

Author Information

Lawrence C. Stewart joined the Cambridge Research Lab (CRL) in 1989 after 5 years at Digital's Systems Research Center. His interests include speech, audio, and multiprocessors. He was one of the designers of the first Alpha AXP computer system. Before joining Digital, Larry was at Xerox PARC. He received an S.B. from MIT in 1976, and M.S. and Ph.D. degrees from Stanford in 1977 and 1981, all in Electrical Engineering.

G. Winfield Treese joined CRL in 1988 after working at MIT on Project Athena. Win's interests are in networks and distributed systems. He received an S.B. in Mathematics from MIT in 1986 and an S.M. in Computer Science from Harvard University in 1992. He is now pursuing a Ph.D. at MIT in the area of computer networks.

James Gettys joined CRL in 1989. Jim's focus at CRL is multimedia audio and video systems. Before joining CRL, Jim spent two years at the Systems Research Center. Before that, he was a Digital engineer and visiting scientist at MIT working on Project Athena. He is one of the two principal designers and developers of the X Window System. Jim received an S.B. from MIT in 1978.

Andrew C. Payne joined CRL in 1992 after receiving a B.S. in Electrical Engineering from Cornell University. As a co-op at Digital in 1990, he helped build the first Alpha AXP chip. Andy's interests include signal processing, speech, and user interfaces.

Thomas M. Levergood joined CRL in 1990. Tom's focus is on speech and audio-related research. He is also involved in Alpha AXP system and software projects, including an experimental evaluation of split user/supervisor cache memories. He received his B.S. in Electrical Engineering in 1984 and M.S. in Electrical Engineering in 1993, both from Worcester Polytechnic Institute.

All of the authors can be reached at: Digital Equipment Corporation, Cambridge Research Lab, One Kendall Square, Bldg. 650, Cambridge, MA 02139, or by e-mail as {stewart, treese, jg, payne, tml}@crl.dec.com.

12 Glossary

AC AC is an abbreviation for audio context. See *Audio context.*

Access control list An AudioFile server maintains a list of hosts from which client program can be run. By default, only programs on the local host and hosts specified in an initial list read by the server can use the server. This access control list can be changed by clients on the local host. Some server implementations can also implement other authorization mechanisms in addition to or in place of this mechanism. The action of this mechanism can be conditional based on the authorization protocol name and data received by the server at connection setup.

ADC Analog to digital converter. A hardware device that converts analog signals into digital form.

Atom An atom is a unique ID corresponding to a string name. Atoms are used to identify properties and types.

Audio context Various information for audio input and output is stored in an audio context (AC), such as the sample type, number of channels, playback gain, record gain, and so on. An audio context can only be used with the audio device on which it was created.

Audio device An audio device is the abstraction AudioFile uses to describe the underlying audio hardware's ADC and DAC. Attributes of an audio device include audio data type, sampling rate, and server buffer size. A server may support multiple audio devices. Devices may support input, output, or both.

Byte order For audio data, the client defines the byte order and the server swaps bytes as necessary.

Client A client is an application program that connects to the audio server by some interprocess communication (IPC) path, such as a TCP connection or a shared memory buffer.

CODEC Contraction of CODer and DECoder. As used in this paper, a CODEC is an 8 KHz sampling device with integral ADC and DAC and anti-aliasing filtering. The CODEC is generally used in telecommunications applications and supports the CCITT G.711 μ-law and A-law encoding specifications.

Connection The IPC path between the server and client program is known as a connection. A client program typically (but not necessarily) has one connection to the server over which requests and events are sent.

DAC Digital to analog converter. A hardware device that converts digital signals to analog form.

dB An abbreviation for decibel, which is a measure of relative power level. It is equal to 10 times the log of a power ratio, so that a signal twice as powerful as another is said to be 3 dB louder. One decibel is approximately the minimum perceptible change in loudness.

DECaudio See *LoFi*

DSP Digital Signal Processor (or Processing).

DSP port The DSP port encapsulates two flexible serial interfaces first found on the DSP56001 and used by NeXT on their first workstation. This interface has become an industry standard. Third party vendors such as Ariel and Applied Speech Technologies make boxes that connect to the DSP port and support flexible ADC and DAC sampling rates.

DTMF Dual Tone Multi-Frequency, also known as Touch-Tone. This is the in-band signaling method for dialing push-button telephones that use 16 tone pairs constructed from two groups of four frequencies. In addition to its dialing function, DTMF generation and decoding is frequently used to control voice response systems.

Event Clients are informed of information asynchronously by means of events. Events can be either generated from devices or generated as side effects of client requests. Events are grouped into types. The server never sends an event to a client unless the client has specifically asked to be informed of that type of event. Event timestamps are reported relative to an audio device.

Event mask Events are requested relative to an audio device. The set of event types a client requests is described by the event mask.

Extension Named extensions to the core protocol can be defined to extend the system. Extensions can add both new requests and new kinds of events.

LineServer The LineServer is an Ethernet peripheral. It is a Motorola 68302 microcomputer system with 128K ROM and 64K RAM, an Ethernet controller, high speed V.35 serial line interface, and an 8 KHz ISDN codec. We use LineServer within Digital's research labs for remote Ethernet bridging and IP network routing over both dedicated digital circuits and dial-up ISDN circuits.

LoFi LoFi is a TURBOchannel option module that contains two CODECs, a DSP56001 signal processor with static RAM, a 44.1 KHz stereo DAC, a DSP port interface, and analog and digital (ISDN) telephone line interfaces. This module is available from DEC as PN AV01B-AA. (LoFi is the research prototype version of DECaudio).

Property Devices can have associated properties that consist of a name, a type, a data format, and some data. The protocol places no interpretation on properties. They are intended as a general-purpose naming mechanism for clients. For example, clients might use properties to share information such as the last telephone number dialed.

Property list The property list of a device is the list of properties that have been defined for that device. See also *Property*.

Reply Information requested by a client program using the AudioFile protocol is sent back to the client with a reply. Both events and replies are multiplexed on the same connection. Some requests do not generate replies.

Request A command to the server is called a request. It is a single block of data sent over a connection.

Sample rate The frequency at which and audio signal is sampled; usually given in Hertz, or samples per second. Popular rates are 8 KHz, for telephone quality audio and 44.1 KHz for compact disc quality audio.

Sample type The encoding format of the sample data supported by an audio device in the server. Popular encodings are 16-bit linear PCM and μ-law.

Server The server provides the basic audio mechanism. It handles connections from clients, multiplexes multiple requests onto the audio devices, and demultiplexes input back to the appropriate clients.

Tcl The Tool Command Language. Tcl is a small, interpreted, application-independent command language. See also *Tk*.

Time Audio device time is represented by a 32-bit (finite length) unsigned integer that increments once per sample period of an audio device and wraps on overflow. The audio device sampling rate is used to move between time in sample ticks and time in seconds. As an example at 8 KHz, four seconds in the future maps to the current value in the time register plus 32000 ticks.

Tk An X toolkit which is an extension to Tcl. See also *Tcl*.

References

[1] Susan Angebranndt, Raymond Drewry, Philip Karlton, and Todd Newman et. al. Definition of the porting layer for the X v11 sampler server, 1990. Located in the doc/Server directory in the MIT X distribution.

[2] Susan Angebranndt, Raymond Drewry, Philip Karlton, and Todd Newman et. al. Strategies for porting the X v11 sample server, 1990. Located in the doc/Server directory in the MIT X distribution.

[3] Susan Angebranndt, Richard L. Hyde, Daphne Huetu Luong, Nagendra Siravara, and Chris Schmandt. Integrating audio and telephony in a distributed workstation environment. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.

[4] B. Arons, C. Binding, K. Lantz, and C. Schmandt. The VOX audio server. In *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, 1989.

[5] Edward Bruckert, Martin Minow, and Walter Tetschner. Three-tiered software and VLSI aid developmental system to read text aloud. *Electronics*, Apr. 21, 1983.

[6] Van Jacobson. Congestion avoidance and control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, August 1988.

[7] Thomas M. Levergood. LoFi: A TURBOchannel audio module. CRL Technical Report 93/9, Digital Equipment Corporation, Cambridge Research Lab, 1993.

[8] D. L. Mills. Network time protocol (NTP). Internet RFC 958, Network Information Center, September 1985.

[9] Jeffrey C. Mogul. Observing TCP dynamics in real networks. In *Proc. SIGCOMM '92 Symposium on Communications Architectures and Protocols*, Baltimore, MD, August 1992.

[10] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter Conference*, January 1990.

[11] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Proceedings of the USENIX Winter Conference*, January 1991.

[12] Steven J. Rohall. Sonix: A network-transparent sound server. In *Proceedings of the Xhibition 92 Conference*, June 1992.

[13] David S. H. Rosenthal and Adam R. de Boor et al. Godzilla's guide to porting the X V11 sample server, 1990. Located in the doc/Server directory in the MIT X distribution.

[14] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, Bedford, MA, 3rd edition, 1991.

[15] Henry Spencer. How to steal code -or- inventing the wheel only once. In *Proceedings of the USENIX Winter Conference*, pages 335–346. USENIX, February 1988.

[16] D. C. Swinehart, L. C. Stewart, and S. M. Ornstein. Adding voice to an office computer network. In *Proceedings of GlobeCom 1983*, November 1983.

[17] Robert Terek and Joseph Pasquale. Experiences with audio conferencing using the X window system, UNIX, and TCP/IP. In *Proceedings of the USENIX Summer Conference*. USENIX, June 1991.

[18] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988.

[19] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conf.*, pages 41–51, London, July 1990.

[20] Stephen A. Uhler. PhoneStation, moving the telephone onto the virtual desktop. In *Proceedings of the USENIX Winter Conference*. USENIX, January 1993.

[21] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proc. SIGCOMM '91 Symposium on Communications Architectures and Protocols*, pages 133–147, Zurich, September 1991.

Index