# Ak: An Audio Toolkit for Tcl/Tk

*Andrew C. Payne*

*Digital Equipment Corporation*
*Cambridge Research Lab*

**Abstract**

Ak is an audio extension for Tcl built on top of the AudioFile System. Ak provides mechanisms for file playback, recording, telephone control, and synchronization.

## 1 Introduction

AudioFile is a network-transparent, device independent audio system [1]. AudioFile is modeled after the X Window System: servers run on systems with audio hardware, and a library of client routines provide a programming interface for applications. The library includes routines to play and record audio samples, to manipulate devices (gain controls, input/output enables), and to control telephone devices. AudioFile also implements events, such as ring, hookswitch, and tone detection.

This paper describes work in progress on Ak, a Tcl audio extension that is built on top of AudioFile. Ak is designed to be simple, yet general and flexible. Simple applications should be easy to write, but sophisticated applications should still be possible. Ak provides the basic functions, such as playback and recording, as well as a convenient framework for implementing new functions. Ak also provides a powerful mechanism for clients that need synchronization.

Ak is similar to the PhoneScript Tcl extension [2], but Ak is designed to be much more general purpose and not restricted to just telephone applications. The remainder of this paper describes Ak's programming interface and implementation.

## 2 Basic Operations

Ak has three basic abstractions: server connections, device contexts, and requests. Server connections are to AudioFile servers on other machines (or the local machine). Device contexts represent a particular device on a server, with a set of context attributes (playback and record gain, sample type, little/big endian data). Requests are operations, such as file playback or recording, that are executed in some device context. This section describes the Tcl commands that implement these abstractions.

### 2.1 A Word About Time

First, it is important to mention the role time plays in AudioFile and Ak. Each device maintains a clock: an integer counter that increments once per sample period. Clients are responsible for explicitly specifying the time of record and playback requests. By specifying the timing appropriately, clients can generate continuous playback and record streams. Consider a playback application, for example, that reads blocks of 100 samples from a file and sends them to the audio device. If the application plays a block of samples at time $t$, the next block would start at time $t + 100$. A record application would schedule requests in a similar fashion to obtain a contiguous stream of samples.

Exposing time to clients greatly simplifies many synchronization problems. Since the clients can control exactly when the sound is going to emerge, they can implement whatever level of synchronization needed. Clients can read the times and synchronize the audio clock with other clock domains, such as the X server or other audio devices.

AudioFile's model of device time is carried through to Ak: all requests are scheduled in the device time for the audio device. Times are represented using convenient string descriptors. For example, the string "**`now`**" represents the current time. Offset modifiers are permitted, with units in samples or time intervals. For example, "**`now +5s`**" refers to a time 5 seconds from the current time. Ak automatically converts the time offsets into sample counts, based on the device's sampling rate.

### 2.2 Connections, Contexts, and Requests

The **`audioserver`** command opens server connections. It takes two arguments: the name of the server connection and the hostname of the server. For example,

```
audioserver main "north-fork:0"
```

opens a connection named **main** to the server on the machine **north-fork**. The name of the server connection is registered as a new Tcl command, which is used to manipulate the connection.

Once the connection is established, device contexts may be created using the server command. For example, this command:

```
main context room-device -device 1
```

creates a context named **room-device** for audio device 1 on the server specified by **main**. The context name is registered with the Tcl interpreter as a new command.

The context command has a number of options to query and manipulate the device. There are options to retrieve the sample rate, sample type, input and output configurations, and other information about the device. For telephone devices, there are commands to manipulate the telephone hookswitch and dial the phone. This example takes the telephone off hook and dials for help:

```
phone-dev hook off
phone-dev dial "911"
```

Requests are created in device contexts. Request types include playback, record, tone generation, pass-through (sending audio between two servers), and action (a command scheduled to execute at some future time). Requests are created using the context **create** command, which returns a unique *request-id* that can be used to manipulate the request after it is created. Here is an example of a play file request:

```
set req [room-device create play \
  "hello.au" -start {now +5s} \
  -stop {now +6s} -offset {+10s}]
```

This example schedules a request to play one second of audio from the file "**hello.au**", starting in 5 seconds, from an offset 10 seconds into the file. The returned request id is stored in the variable **req**. The play request has a variety of options, including a way for commands to be executed at the beginning, the end, and on regular tick intervals.

This is an example of an action request, which schedules a command to be executed 5 seconds from now:

```
room-device create action \
  "puts stdout BANG!" -at {now +5s}
```

Note that actions are scheduled using the audio device's clock. This provides a simple, yet powerful, mechanism for synchronizing to audio. For example, actions could be used to update a screen animation in sync with playback audio.

Once created, requests can be manipulated with the **reqconfig** command. For example, this command halts the previous play request "mid-stream" by changing the stop time:

```
room-dev reqconfig $req -stop {now}
```

Meaningless configuration requests, such as modifying the start time for a request that has already started, generate an error.

### 2.3 Events

AudioFile devices may generate events that are sent to interested clients. Ak allows commands to be bound to events in a device context. This example picks up the phone whenever it rings (**phone-dev** is a context for a telephone device):

```
phone-dev bind <RingStart> \
  "phone-dev hook off"
```

Ak performs substitutions on the bound command before it is executed. For example, for the **<DTMFStart>** event (generated when a Touch-Tone$^{TM}$ is detected), the string '**%d**' is substituted with the keyed digit.

## 3 Implementation

Currently, Ak consists of about 2500 lines of C. About 2000 lines are for general routines and common code. The remaining 500 lines implement the play request. The implementations of the other request types are in progress.

Ak's request types are implemented in a flexible manner, much like Tk's canvas item types. A type table contains an entry for each request type, with information including pointers to procedures to implement the basic request operations: create, configure, and delete. Adding new types is straightforward.

Ak hides many of the details of the underlying implementation. For example, most AudioFile servers buffer only about four seconds of audio. Therefore, a large play request must be broken into smaller chunks that get written to the server at regular intervals. Similarly, a large record request would have to read samples from the server at regular intervals.

To simplify the implementation, Ak provides a

scheduler that allows procedures to be executed at arbitrary audio times. For example, a playback request might schedule an update task to run every 1000 samples. Having the scheduler run in audio time hides details like the sample rate and clock drift. The scheduler uses a priority queue, based on device audio time. For each device context, an update task reads the next item from the queue, executes it, and reschedules the next update using Tk's timer routines.[1] The update task fires at least once a minute, even if there are no tasks to execute, to compensate for any drift between the system clock and the audio clock.

## 4 Conclusion

Hopefully, Ak will do for audio what Tk has done for X. Together, Ak and Tk provide a powerful and flexible system for developing multimedia applications. Ak is being used to implement a full-featured tape deck, multimedia presentations and tutorials, and a telephone inquiry system.

There is still a lot of work ahead. The major tasks include:

- Finish implementing request types
- A tagging mechanism for requests (which will be useful for manipulating whole sets of requests at once)
- More applications!

A beta-version should be finished this summer, with a source kit available by anonymous FTP.

## References


[1] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart. AudioFile: A network-transparent system for distributed audio applications. In *USENIX Summer 1993 Conference Proceedings*. USENIX, 1993. To appear.

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


## Author Information


Andrew Payne is a research associate at Digital's Cambridge Research Lab in Cambridge, Massachusetts. His interests include signal processing, speech, and user interfaces. He can be reached via e-mail at:

**payne@crl.dec.com**


[1] This is the only major Tk subroutine used by Ak. Other Tk routines are used for signaling background errors and parsing arguments. With suitable replacements for these routines, Ak could be made independent of Tk.