An Asynchronous Audio Server

Chris Schmandt and Jordan M. Slott

Speech Research Group, Media Laboratory
Massachusetts Institute of Technology
Building E15, 20 Ames Street, Cambridge, MA 02139

The authors may be found at the following address:

Chris Schmandt (geek@media-lab.mit.edu)
E15-356
20 Ames Street
Cambridge, MA 02139

Jordan M. Slott (hordack@media-lab.mit.edu
E15-344
20 Ames Street
Cambridge, MA 02139

Chris Schmandt is a Principal Research Scientist and Head of the Speech Research Group at the MIT Media Laboratory.

Jordan M. Slott is an Undergraduate student in the Department of Electrical Engineering and Computer Science at MIT.

ISBN 0-000-000000-0

Contents

Figures

Tables

CHAPTER 1 Introduction

The document describes the Audio Server project done by the Speech Research Group at the M.I.T. Media Laboratory. The primary goal of the document is to provide a reference for application developers who incorporate audio into their application and want to use the Audio Server environment as a framework. As a consequence, the majority of the chapters contained within will be an application programmatic interface (API) specification--a well-defined guideline on taking advantage of the Audio Server's capabilities.

Before an interface specification is provided, it is useful if the application developer understands the reasons behind the Audio Server project as well as it's design principle and goals. This introduction will provide this background, and should be read thoroughly. In many respects, once an application developer understands these concepts, the Audio Server API is straightforward and intuitive.

Requirements for an Audio Environment

The Speech Group is composed of a dozen or so Sun SPARCstations (ranging from SPARCstation 1's through SPARCstation 10's) on a local area network. They are each file served from a central server. Each SPARCstation is equipped standard with an audio "device." On the older machines, only 8 Khz, U-law data is available; multiple sampling rates and encodings are available on the newer machines.

Based on ~~Through~~ the work performed by the group, the following requirements for ~~on the~~ audio capabilities of a workstation were deemed critical:

- Standardized and simple API for ~~to~~ playing or to play and recording digital audio data.
- Multiple applications can access the audio device at once. That is, several clients can request to play and record audio at once, with resource management and arbitration by the server.
- Asynchronous operation and notification of events and the ability to delegate real-time operations to the server.
- Access, via the local-area network, to the audio resources of a remote workstation, i.e. network transparent.
- Standardization of audio file management.
- Availability of various real-time, pre- and post- audio processing capabilities, such as time-scale and gain modification, DTMF detection, and silence detection.

The first stated requirement is true of any environment used by ~~which~~ multiple application developers ~~use~~. A simple audio API allows for the quick incorporation of audio into an application. A standardized interface allows changes and improvements to the audio environment to propagate to each client transparently. In an informal sense, all of the audio-capable applications developed by the Speech Group will have a similar/interface consistent ~~"look alike"~~ in terms of the audio interface.

for a member of the speech group

Some typical audio clients running on a workstation are: a graphical voice-mail player, a recording vu-meter, a digital sound editor, a speech-recognition based window focus application, and an hourly chimes application. These applications should all run on a single workstation and share the audio resources in some well defined manner. For example, a vu-meter displays the level of the recording, but only should do so when another client is currently recording. [The speech-recognition system should always be listening for user commands.] not necessarily Also, other applications should be able to record at the same time an application is performing speech recognition. yes A digital sound editor both plays and records audio data. An hourly

why? it works as a stand-alone app for monitoring/checking audio levels chimes application plays an audio file periodically, and should be able to co-exist and operate normally if, say, the voice-mail player is currently playing a voice message.

Audio is inherently asynchronous. Audio data can be played and recorded while other activities are occurring, such as graphical user interface input. An application which uses an asychronous audio management scheme does not have to sit in a tight loop while processing audio data; it is free to go about other activities. The reverse is true as well--if the audio management operates asynchronously from the application, it is not subject to synchronous delays inside the application.

In an audio rich environment, various centralized services may exist, including text-to-speech conversion, speech recognition, and teleservices. These services may be limited to a particular workstation because of the computing resources or ized special hardware available only on that workstation. In this situation, these centralized services may need access to audio from another workstation. A common example is a speech recognition server running on a remote machine operating on speech from a user's local microphone. Another example is remote listening to a radio tuner connected to a different workstation. It is these reasons why audio must be available over the local-area network.

The final two requirements remove some of the programming burden on applications into a more centralized location. The numerous audio-capable applications on a workstation should be able to share audio files. The ability to time-scale audio data or perform DTMF (touch-tone) detection ~~on audio data mainly~~ provides convenience and functionality to all clients.

The Existing Sun Audio Interface

basic or minimal Although all Sun SPARCstations come with an audio codec as standard equipment, they have only ~~a crude~~ interface to audio. The interface to the Sun audio "device" is a special file on each workstation's local disk called ~~"~~**/dev/audio**~~"~~. For file management, a demo library, **libaudio**, ~~is provided which~~ provides basic routines for reading and writing audio files using a particular file format.

Opening a Connection to the Audio Device

Clients may open **/dev/audio** much like any other file. Using the standard UNIX file operations, the **open()** system call returns a file descriptor. The particular characteristic about the Sun audio device is that only a single client may open the audio device for playing or recording. All other clients wishing to open an already opened device will be unable to do so.

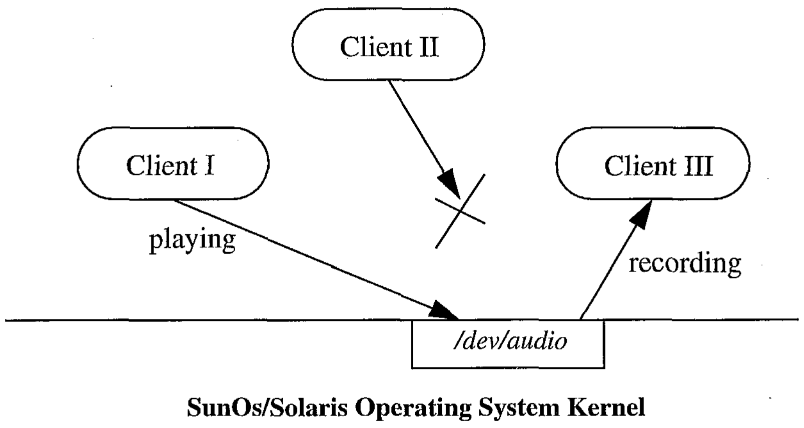

FIGURE 1. **Client~~s~~ interaction with the Sun audio device.**

The Figure above shows a typical Sun workstation. The audio device is considered part of the operating system kernel, which is separated by a line in the diagram from the user's process space. Client I has **/dev/audio** opened for writing (i.e. playing) and Client III has **/dev/audio** opened for reading (i.e. recording). Client II attempts to open the audio device for playing, but is unable to do so. As a result, Client II must therefore manually attempt to re-open the audio device at a later time.

Herein lies one of the major problems with the Sun audio interface. The types of applications which need to run in ~~an~~ the environment ~~such as the~~ developed by Speech Research Group could not survive using the Sun audio interface alone. Multiple applications which require record audio data at the same would be unable to do so. Each application would fight for the total resources (i.e. total in terms of playing and recording separately) of the workstation. A misbehaving audio-capable application might

monopolizing the audio resource
never release control of the audio device, ~~transforming the workstation into a audio-starved environment rather than an audio-rich one.~~ for only a single app.

Another limitation of the Sun audio device is the lack of remote accessibility. Since **/dev/audio** is a device on the local file system, only a client running on the local workstation may **open**() it. Therefore, audio data may not be shared in any easy way across the network. This limits a user to running only those audio-capable applications which are able to run on the local workstation, and do not rely on a remote audio-based server.

Reading ~~Data From~~ and Writing Data To the Audio Device

Once a client has successfully opened the audio device, it plays audio data out the workstation speak~~ing~~er by writing the digital audio data to the device. Similarly, a client can record digital audio data by reading from the device. This can be accomplished via the standard UNIX **read**() and **write**() system calls. If the client wishes to play audio data from a disk file, for example, it ~~must manually~~ manage the reading of the data from disk and writing to the device. This interface allows the precise control and acknowledgment of how much data was sent to the device and visa versa. which?

e Sun for reading/writing
Th~~is~~ interface suffers from two flaws. First, clients which use this interface to play and record must perform their own memory management. This often entails allocating the proper amount of memory and keeping track of the number of bytes either read or written. Not only can this method be prone to bugs, but it is also sloppy and should not be the client's responsibility.

the issue of
The second flaw is ~~a matter is~~ synchronous operation versus asynchronous operation. More advanced applications make use of audio while engaging in other activities simultaneously. Each client must sit in a loop reading/writing audio data; it is difficult to side-track to other tasks without causing the resulting audio to become choppy. Some multi-threaded libraries have appeared which solve this synchronosity problem. However, writing multi-threaded code is difficult to debug and lacks the ability to operate over a network.

Audio Device Progress and Audio Processing

Because the Sun audio interface is inherently synchronous and audio inherently asynchronous, several audio operations become awkward. For example, a synchronous call is needed to find out how much audio data has actually been played. If an application wants to know when a particular landmark amount has been played, it must continuously query the device. Since an application might be busy doing other tasks, it may miss this landmark. Also, to know when the audio device has truly finished playing a sound, the client must again query the device for this information. A more natural interaction would be for the client to be asynchronously notified of the finish of a playing. awk

The Sun audio interface provides no audio processing. It does, however, allow~~s~~ a client to record ~~in~~ at a desired sampling rate and encoding, if the workstation's hardware is capable. All other audio processing such as time-scale modification and silence detection must be performed by the client.

An Audio Server Solution

In consideration on the requirements of an audio environment and the limitations of the existing Sun audio interface, the Speech Research Group developed an asynchronous Audio Server.[1] The figure below diagrams the basic function ~~on~~ of the Audio Server.

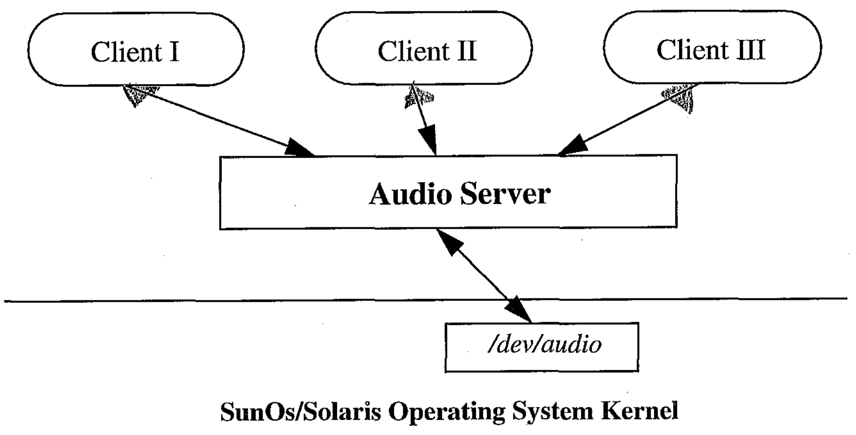

FIGURE 2. The Audio Server Environment.

maybe you should reference previous papers for people who want more detail

1. The Audio Server project is an ongoing one. It began roughly in 1983 and is still under development in the fall of 1994.

In the Audio Server environment, individual clients no longer communicate with the Sun audio device (**/dev/audio**) directly. Each client communicates via a custom IPC (Interprocess Communication) protocol. The Audio Server is responsible for the management of the audio device, opening and closing the device, and reading and writing from the device. This IPC mechanism allows clients to communicate with an Audio Server running on a remote workstation. The Audio Server is unaware of whether a client is running locally or remotely.

A Multi-Client Audio Server

The Audio Server can accept an unlimited number of clients. Although a given Audio Server can have any number of clients, its performance as the number of clients gets large (i.e., greater than 10 active clients) may decrease. Which client(s) are permitted to play or record audio data from a device at a particular time is determined by a rule-based mechanism inside the Audio Server. The ~~whole~~ process of deciding which Audio Server clients are permitted to perform a task is called audio resource management. This will be discussed extensively in Chapter x.

An Asynchronous Audio Server

In accordance with the requirements of the Speech Group, the Audio Server is asynchronous in its operation. From the perspective of a client, it is free to go about other tasks while the Audio Server is playing or recording audio data. A client may receive notification of changes updates in the Audio Server's state, for example, when playing an audio file has concluded. A client is also told when a client-caused error occurs inside the server (i.e. invalid file name, invalid parameter, etc.)

or just "receive updates of the Audio..."

Get rid of

CHAPTER 2

A Sample Audio Server Client

The purpose of this chapter is to introduce the Audio Server API by providing a simple example, **play.c**. **Play.c** requests that the Audio Server plays an audio file located on disk. It also defines several event callbacks for notification of the progress of the play request.

This example code is meant to be a skeleton for more advanced Audio Server clients. More advanced features of the Audio Server are described in Chapter x.

The example code covers the following topics:

- Audio Server include files
- Establishing a Connection to the Audio Server
- Defining and Registering Asynchronous Event Callbacks
- Requesting the Audio Server to Play an audio file.
- Entering the Audio Server main loop

Audio Server Include Files

The following include files must be present in every Audio Server client:

TABLE 1. **Audio Server Include Files**

```
#include <s_service.h>
#include <s_app.h>
#include <sparc_sound.h>
#include <bsm.h>
```

The file **s_service.h** has definitions pertaining to all of the available Audio Server request functions, while the file **s_app.h** has definitions pertaining to the available Audio Server event callback functions. **sparc_sound.h** contains miscellaneous constant defniitions and **bsm.h** contains definitions for the Audio Server RPC mechanism.

Establishing a Connection to the Audio Server

Before a client can issue requests and receive events from the Audio Server, it must open a connection to it. This is typically done once in a client towards the beginning. Opening a connection to the Audio Server does not tie up any resources or interfere with other clients who wish to open a connection to the same Audio Server.

TABLE 2. **Opening a Connection to the Audio Server**

```
#define AUDIO_SERVICE "s_server"
#define AUDIO_HOST ""

int audiofd;
if ((audiofd = s_open_server(AUDIO_SERVICE, AUDIO_HOST)) < 0) {
 fprintf(stderr, "Fatal Error: Cannot Open a Connection to Audio Server.\n");
 exit(-1);
}
```

The function **s_open_server**() has two arguments: a service name and a host. In mostly all cases, the service name is "s_server." [See Chapter x about the Audio Server Status Service for more information] The host argument is the name of a valid machine to which connect. This functionality allows a client to access the audio resources of a remote workstation. In the case of **play.c**, the host is set to the empty string (""), which requests a connection to the Audio Server on the client's host.

The return value of **s_open_server**() is an integer file descriptor. A return value of -1 indicates that a connection to the specified audio server has failed. This file descriptor is most often the first argument to all future Audio Server calls. In some advanced clients, connection to audio servers on different hosts are kept open at one time. These file descriptors are used to differentiate to which Audio Server each request is sent.

Defining And Registering Asynchronous Callbacks

The Audio Server can notify clients about various events results associated with their requests. The client play.c asks to be notified when the playing of the audio file begins and when the playing ends. Since all requests to the Audio Server are asynchronous, these event callbacks provide valuable information to the client as to the success of its requests. Also, these events allow the client to provide real-time information to the user about Audio Server activity.

TABLE 3. Defining Asynchronous Event Callbacks

```
play_begin(int audiofd, char *clientdata, int handle)
{
 fprintf(stderr, "Playing has begun on connection=%d, client data=%s, handle=%d\n",
audiofd, clientdata, handle);
}

play_done(int audiofd, char *clientdata, int termination, int handle, int position)
{
 fprintf(stderr, "Playing has finished on connection=%s, client data=%s, termina-
tion=%d, handle=%d, position=%d\n", audiofd, clientdata, termination, handle, posi-
tion);
}
```

Note the definitions of **play_begin()** and **play_done()**. The first argument of each of these callbacks is the connection file descriptor, the same descriptor returned by **s_open_server()**. The second argument is always user-supplied client data of the type **char** *. This client data is supplied in the callback register function, as described below. The remaining arguments to each of the event callbacks are described in detail in Chapter x.

TABLE 4. Registering Asynchronous Event Callbacks

```
s_register_callback(S_PLAY_BEGIN_EV, play_begin, (char *)NULL);
s_register_callback(S_PLAY_DONE_EV, play_done, (char *)NULL);
```

The Audio Server reqeust **s_register_callback()** allows clients to register a defined function to be called when a certain event inside the Audio Server occurs. **s_register_callback()** is one of the few Audio Server request which does not require the connection file descriptor as an argument.

The first argument to **s_register_callback()** is the name of the event for which the function is registered. In play.c, the event name for when a play starts is S_PLAY_-

BEGIN_EV, and S_PLAY_DONE_EV is the event name for when a play request finishes. A complete description of all Audio Server event names can be found in Chapter x. The second argument is the function pointer of the desired callback and the third argument is the user-supplied client data. This client data, as described earlier, is passed to the callback function as its second argument.

Requesting the Audio Server to Play an Audio File

The **s_play()** request is a convenience function, asking the Audio Server to play a specified audio file:

TABLE 5. **The s_play() convienence function**

s_play(audiofd, "/sound/hordack/hello_world.snd");

The **s_play()** request takes two arguments: the first is the connection fd, and the second is the path name of a valid Sun audio file. This function actually translates to several audio server requests. In more advanced Audio Server clients, these separate requests may be used instead of the one convenience function:

TABLE 6. **Playing an Audio File**

s_name(audiofd, "/sound/hordack/hello_word.snd");
s_start(audiofd, 0);
s_prepare_play(audiofd);
s_continue_play(audiofd);

Entering the Audio Server Main Loop

Since all Audio Server clients are asynchronous, each must invoke an event main loop--much like the X Window System event loop. If the Audio Server main loop is not called, the client will never be notified of any Audio Server events.

TABLE 7. **The Main Loop**

s_main_loop();

The function **s_main_loop()** has no arguments nor return values. Unless an unexpected fatal error ocurrs, **s_main_loop()** should never return.

Complete Code Listing for Play.c

The following is a complete code listing for play.c:

TABLE 8. Complete Code Listing for play.c

```
#include <s_service.h>
#include <s_app.h>
#include <sparc_sound.h>
#include <bsm.h>

void play_begin(int audiofd, char *clientdata, int handle)
{
 fprintf(stderr, “Playing has begun.\n”);
}

void play_done(int audiofd, char *clientdata, int termination, int handle, int position)
{
 fprintf(stderr, “Playing has finished.\n”);
}

main()
{
 int audiofd;
 if ((audiofd = s_open_server(AUDIO_SERVICE, AUDIO_HOST)) < 0) {
 fprintf(stderr, “Fatal Error: Cannot Connect to Audio Server.\n”);
 exit(-1);
 }

 s_register_callback(S_PLAY_BEGIN_EV, play_begin, (char *)NULL);
 s_register_callback(S_PLAY_DONE_EV, play_done, (char *)NULL);
 s_play(audiofd, “/sound/hordack/hello_world.snd”);
 s_main_loop()
}
```

CHAPTER 3

Audio Server Events

The following is a list of all of the events the Audio Server may generate and the declaration of their respective callbacks. To register a callback for an event, use the **s_register_callback**() request.

Events	**Description**
S_PLAY_BEGIN_EV	A play has begun.
S_PLAY_DONE_EV	A play has finished
S_PLAY_BUF_EV	The Audio Server needs more play data (when playing from buffers).
S_RECORD_BEGIN_EV	A record has begun.
S_RECORD_DONE_EV	A record has finished.
S_RECORD_BUF_EV	New record data is available to the client.
S_PREEMPTED_EV	A client request has been interrupted by another client.
S_CONTINUE_EV	A client request has resumed after being interrupted by another client request.
S_INTEREST_UNABLE_EV	A client data interest is interrupted because of a mismatching audio format (to go away!)

Events	Description
S_INTEREST_ABLE_EV	A client's data interest is resumed after being interrupted because of a mismatching audio format (to go away soon!)
S_INTEREST_DATA_EV	Record data, requested by a client as an interest, is available.
S_INTEREST_DATA_BEGIN_EV	A registered data interest has begun.
S_INTEREST_DATA_DONE_EV	A registered data interest has ended.
S_DTMF_EV	A touch tone has been detected.
S_ENERGY_EV	An energy calculation is available to the client.
S_INTEREST_EVENT_BEGIN_EV	A registered interest activity has begun.
S_INTEREST_EVENT_DONE_EV	A registered interest activity has ended.

```
void s_play_begin_ev(int sfd, char *clientData, int handle);
void s_play_done_ev(int sfd, char *clientData, int done, int handle, int position);
void s_play_buf_ev(int sfd, char *clientData);
void s_record_begin_ev(int sfd, char *clientData, int handle);
void s_record_done_ev(int sfd, char *clientData, int done, int handle, int position);
void s_record_buf_ev(int sfd, char *clientData, bsm_bytes *buffer);
void s_preempted_ev(int sfd, char *clientData, int cause);
void s_continue_ev(int sfd, char *clientData, int cause);
void s_interest_unable_ev(int sfd, char *clientData);
void s_interest_able_ev(int sfd, char *clientData);
void s_interest_data_ev(int sfd, char *clientData, bsm_bytes *buffer);
void s_interest_data_begin_ev(int sfd, char *clientData, int interest);
void s_interest_data_done_ev(int sfd, char *clientData, int interest);
void s_dtmf_ev(int sfd, char *clientData, int tone);
void s_energy_ev(int sfd, char *clientData, int channel, int packetsize, int avgmag, int
peak, int rmsenergy);
s_interest_event_begin_ev(int sfd, char *clientData, int interest);
s_interest_event_done_ev(int sfd, char *clientData, int interest);
```

CHAPTER 4

Fundamental Concepts of Playing and Recording

This chapter introduces the general ideas involved when a client wants to play or record digital audio data. There is much in common between playing and recording, and this chapter exists to provide an explanation of these common ideas and concepts once. Later, in the following two Chapters, the specifics of playing and recording are discussed in turn.

This chapter covers the following material:

- Conceptual process of playing and recording
- Setting a new play or record device.
- Buffered operations versus file operations.
- API requests for file and buffer operations.
- Obtaining status on a play or a record.

Conceptual Model Behind Playing and Recording

Before using the Audio Server to play or record sound, it is important to understand the conceptual model on which the Audio Server performs these tasks. The Audio Server API is designed around this model. Understanding the model allows the application developer to better understand the API.

Playing and recording are essentially complementary processes. The main difference is which way the digital audio flows. In playing, audio data originates from a source and is played out an output (write) audio device. In recording, audio data originates from a read device and flows to a destination. The figure below illustrates this process:

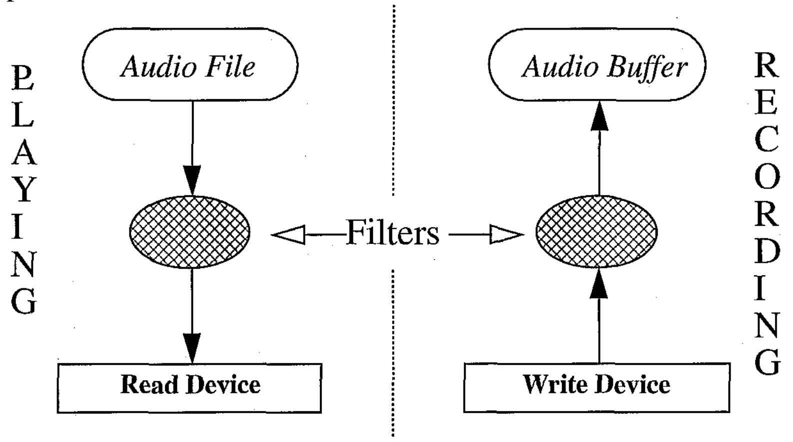

FIGURE 3. **Model for Playing and Recording**

Files and Devices

All data flows from a *device* to a *file*. Note, however, that these classifications are historic mostly, and their true meaning is much broader than specifically a device and specifically a file. Typically allowable *devices* are audio device, named pipes, memory buffers, and even disk files. *Files* can also be named pipes and memory buffers as well. The Audio Server avoids labelling these as sources as syncs in general.

Digital Audio Filters

When digital audio is being played or recorded, it may pass through one or many audio filters. Audio filters may perform any or all of the following: modify the audio data, observer characteristics from the audio data, or alter the current activity based on the audio data.

In the case of playing digital audio, two common filters server as examples. First, time-scale modification of the data either slows down or speeds up the playing of the audio data. Gain modification either softens or loudens the digital audio. For recording, two possible filters are DTMF (touch-tone) detection, which calculates the telephony key pressed when it hears a touch tone, and silence detection, which determines when a pause occurs in the audio data. The two recording examples also interrupt recordings under certain conditions. The available filters in the Audio Server are explained further in Chapters x and x.

Setting an Alternative Play and Record Device

In most instances, applications use the default audio device on the workstation: the microphone and built-in speaker. Some other applications require a more advanced usage of the audio server and might need to use a different play or record audio device. For example, the voice mail application asks the audio server to play and record from the telephone device. Other applications might want the digital audio to be read/written to a named pipe. This sub-section describes how to set an alternative play and record device.

The Audio Server maintains the notion that the play and record device are each separate. Each is opened and closed independently of one another. Also, each can be changes independently of the other. This means that although the recording device was set to a named pipe, the playing device can still be the workstation's speaker.

Also, a current disadvantage of the Audio Server is that when a client changes a device, *all* clients use that new device. This is the wrong behavior because a client can have its device changed without its knowledge. In the future, the Audio Server, will implement a per-client device control policy, where changing the device in one client does not affect another client.

Setting the Device at Audio Server Start-Up Time

A simple way to change the read or write device in the audio server is by a command line argument when the Audio Server is initially started.

There are three command line arguments: '-a <device name>' which sets both the and reading and writing device. This is used, for example, when you want to use another audio-like device on the workstation. One example is an ISDN telephone line voice channel, i.e. '-a /dev/isdn/b0'. To set only the reading device (and not alter the writing device), use the 'r <device name>' option, and to set only the writing device, use the '-w <device name>' option. The table below summarizes the command line options available to modify the audio devices used:

TABLE 9. **Command Line Arguments**

Command Line Argument	**Description**
-a <device name>	Sets both the reading and writing device.
-r <device name>	Sets the reading device.
-w <device name>	Sets the writing device.

Setting the Device within the Client Application

A client may set change the Audio Server device via the request **s_set_device()**. The **s_set_device()** request takes three arguments. The first is the usual connection fd. The second is the name of the new read device while the third is the name of the new write device. Note that at the present, there are no separate commands for the read and write device. The table below summarizes the **s_set_device()** request:

TABLE 10. **The s_set_device() Request**

Request	**Description**
void s_set_device(int fd, char *read, char *write);	Sets the new read device to **read** and the new write device to **write**.

File vs. Buffer Operations

The Audio Server allows clients to play from or record to disk files or memory buffers. In Figure 3 above, the playing is being done from a file, while the recording is being done into a memory buffer. There are two distinct sets of requests within the API specific to whether the operation (i.e. playing or recording) is being done with files or with buffers.

What is a File?

A file, obviously, is a collection of bytes on some disk. There are several variations of files, which are enumerated here. In the future, the term *file* may refer to all of these: character (normal) file, block (device) file, and a named pipe (fifo). Audio and other audio-like devices are block files.

API Request for File Operations

The sub-section is meant to only briefly introduce the API associated with file operations. These requests will be discussed in more detail in the following chapters.

The primary request for all file operation is the **s_name()** request. This request informs the Audio Server of the name of the file which is to be played from or recorded to.

The **s_name()** request is followed by the **s_prepare_play()/s_continue_play()** and the **s_prepare_record()/s_continue_record()** command to begin the actual action.

What is a Buffer?

A buffer is an allocated portion of the workstation's volatile memory which stores digital audio data. Applications use buffered operations if they wish to handle the actual audio data. This is often useful when the application sends the audio data to a remote host. During playing, applications supply the Audio Server with memory buffers via requests. They are notified when the Audio Server needs more data via an asynchronous callback. During recording, when audio data become available, the Audio Server sends this data to the application via asynchronous callbacks.

API Requests for Buffer Operations

The following requests will be described in greater detail in future chapters.

To record into a memory buffer, an application invokes the **s_async_record_buf()** request. [To start actually recording, it must also call **s_continue_record()**. This is a strange thing to do, and probably will be changed in the near future.] Digital audio is made available to the client via the asynchronous event S_RECORD_BUF_EV.

To play from a memory buffer, an application calls **s_async_play_buf()**. When the Audio Server need more audio data, it sends the asynchronous event S_PLAY_-BUF_EV. To send additional audio data to the server, the application subsequently calls **s_async_play_new_buf()**. [Note: in the future they may be only a single **s_async_play_buf()** request serving both purposes.]

Obtaining Play/Record Status

There are two API requests which allow the application to obtain information about the activity currently associated with that client synchronously. There are **s_done()** and **s_where_is_sound()**.

The **s_done()** request returns an integer code representing the current state of the activity. Often, when an activity completes, the **s_done()** command reports to the client the reason for the termination of the most recent activity. Although **s_done()** returns a number of codes, it returns a value equivalent to a boolean FALSE if an activity is currently running and TRUE is it has completed. A table describing the **s_done()** request and all of its return values is found at the end of this section.

The **s_where_is_sound()** request returns the amount of audio data that has been either played or recorded. This amount is in milliseconds. If there is currently no activity going on, **s_where_is_sound()** returns 0.

The table below summarizes these two requests:

TABLE 11. Requests for Obtaining Play/Record Status

Request	Description
int s_done(int sfd);	Returns the status of the current activities.
Return Codes:	
S_NOT_DONE	The activity is currently ongoing.
S_INTR	The activity was interrupted by the client.
S_TONE	The activity was halted by the presence of a touch tone.
S_MAXD	The maximum duration was exceeded.
S_NO_SND	Pause exceeded initial pause limit
S_PAUSE	Pause exceeded final pause limit
S_P_EOF	An EOF was reached on a file
S_BUFFER_FULL	Buffer full recording.
S_BUFFER_EMPTY	Buffer empty playing.
int s_where_is_sound(int sfd);	Returns the number of milliseconds that have been played/recorded. Returns 0 if no activity exists.

CHAPTER 5

Playing Digital Audio

In Chapter 2, the example client **play.c** used the Audio Server request **s_play()** to play an entire audio file out the workstation speaker. In some instances, this is all the functionality need, but in many more cases, a client needs greater control of what, where, and how a sound is played. This Chapter covers the entire functionality for playing digital audio data using the Audio Server, including the following topics:

- Setting an alternative playing device
- Changing the default Sun audio device output port
- Specifying the name of a file to play from.
- Setting the starting and stopping times of the play
- Starting and Halting a play
- Perform pre-processing on the audio data: time-scale modification and gain modification.

Changing the Playing Device

As discused in Chapter 4, the Audio Server maintains a single playing device for all clients connected to the Audio Server. A client may change the global playing (i.e. writing) device by either of two methods. The playing device may be set upon

Audio Server start up time via command line options. Also, a client may invoke the **s_set_device**() request once connected to the Audio Server. This is discussed in more detail in the previous chapter.

Changing the Default Audio Port

The Sun default audio device, */dev/audio*, has two different output ports. Audio can either be played out of the speaker port. The speaker port is connected to an internal speaker within the workstation. The headphone port is an RCA jack located at the back of the workstation, into which any amplifying device may be placed. Most commonly, this port is used for headphones.

The **s_output_port**() request changes which output port to use. Note that this change is server wide and affect all clients. Also, the speaker versus headphone port setting is only relevant if the playing device is the Sun default audio device. The **s_output_port**() request has two arguments: the Audio Server connection file descriptor, and which port to use. A client specifies which port to use by either of the two predefined constants, AUDIO_SPEAKER and AUDIO_HEADPHONE.

The **s_output_port**() request is summarized in the table below:

TABLE 12. **The s_output_port() request**

Request	**Description**
void s_output_port(int fd, int port);	Sets the output port for the default audio playing device. The argument **port** may either be AUDI-O_SPEAKER or AUDIO_HEADPHONE.

Playing from Files versus Playing from Buffers

As described in Chapter 4, playing audio data can take two forms, playing from files and playing from memory buffers. The following sections describe, in depth, the API associated with each.

Playing from Files

The following sub-section describes the API assocaited with playing from files. It describes how to specify which file to play from, how to set the starting and stopping locations for playing, and how to halt the playing immediately.

Setting the File Name

When playing from files, it is neccessary to tell the audio server from what file you want to get the audio data. The **s_name()** requests inform the audio server of this file name. The most recent **s_name()** request issued by the client takes precedence. As mentioned in Chapter 3, if a full path name is not provided, the Audio Server assumes the given path name to exist under the current working sound directory.

The first example sets the name of the file to play as "/home/hordack/foo" while the second sets the name of the file to "/sound/hordack/foo", assuming the current working sound directory is "/sound/hordack." Note that the first example is an

TABLE 13. **Various s_name() commands.**

s_name(sfd, "/home/hordack/foo");
s_name(sfd, "foo");

absoulte path name which the second is not. The **s_name()** request has no return values, any error on the specified file is reported when the play is actually begun.

Audio files may be of various formats, i.e. different encodings and sampling rates. This information is stored in the standard Sun Audio File header description. The Audio Server automatically detects the format of the data stored in the file and reconfigures the playing audio device automatically. Those audio files without Sun audio file headers, i.e. a file only containing audio data, will be assumed tocontain 8 KHz, Ulaw encoded data.

Setting the Starting and Stopping Times of the Play

Without further information, the audio server assumes the client wishes to play the audio file in its entirety. Clients, however, may only want to play a certain chunk of the audio data within the file. The s_start() and s_stop() requests allow the client to provide the Audio Server with this information.

The **s_start()** and **s_stop()** requests each have two arguments. The second argument is an offset into the file of type long, given in milliseconds (t_{start} and t_{stop}).

The Audio Server translates both of these values into byte count offsets internally. The *time values given to* **s_start**() *and* **s_stop**() *are constrained to be greater than or* equal to zero (0) milliseconds and less than or equal to the total millisecond length of the file. (To find out the length of a file in milliseconds, see the **s_length**() request). The Audio Server places the additional contratins that t_{start} must be less than or equal to t_{stop}.

A client may specify t_{start} and/or t_{stop} or neither. The default value for t_{start} is 0 milliseconds and the default value for t_{stop} is the length of the file, in milliseconds. The following several examples illustrated the many uses of **s_start**() and **s_stop**().

TABLE 14. **Examples of s_start() and s_stop()**

```
/* Plays the first 5 seconds of the file */
s_stop(sfd, 5000);

/* Plays the last 5 seconds of the file */
length = s_length(sfd, fname);
s_start(sfd, length - 5000);

/* Plays from the third millisecond to the seventh millisecond */
s_start(sfd, 3);
s_stop(sfd, 7);
```

Starting and Halting a Play

Once a client provided a file name and optional starting and st oping information, it may start and stop the play. To start playing, a client first issues a **s_prepare_play**() request and then an **s_continue_play**() request. The **s_prepare_play**() request asks the Audio Server to ready itself to begin the playing. The purpose is to allow the Audio Server some time to get ready before it is asked to play in order to reduce the latency between the time the Audio Server is asked to begin playing and when it actually can. While the **s_prepare_play**() request is still required, its usefull is diminished in modern workstation which have quick file systems and cached memory.

The **s_prepare_play**() and **s_continue_play**() take a single argument, the connection file descriptor and have no return values. The only constraint is that an **s_continue_play**() must be immediately preceded (i.e. no other Audio Server requests) by an **s_prepare_play()** request. The **s_continue_play()** requests generates an S_PLAY_BEGIN_EV event if the play has begun successfully.

TABLE 15. **Examples of s_prepare_play() and s_continue_play()**

s_prepare_play(sfd);
s_continue_play(sfd);

The client may halt the playing at any time using the **s_halt_play()** command. This command executes immediately, flushing an existing audio data inside the audio device which has not been played. This requests generates an S_PLAY_DONE_EV event. Note that this does not pause the playing, but stops it entirely. If a client wants to play again, it must reissue the **s_prepare_play()** and **s_continue_play()** requests.

The **s_halt_play()** request is called as follows:

TABLE 16. **The s_halt_play() request**

s_halt_play(sfd);

Putting Playing from Files All Together

The following several examples show the sequence of commands used to play varous files. For a complete example of a client which plays from files, see **play.c** in *Chapter 2: A Sample Audio Server Client.*

TABLE 17. **Playing from Files**

/* Plays "/home/hordack/foo" from the first second to the fourth second */
s_name(sfd, "/home/hordack/foo");
s_start(sfd, 1000);
s_stop(sfd, 4000);
s_prepare_play(sfd);
s_continue_play(sfd);
/* Plays "/sound/geek/calendar/monday" in its entirert */
s_name(sfd, "calendar/monday");
s_prepare_play(sfd);
s_continue_play(sfd);

Using The s_play() Convenience Request

The **s_play()** request is a quick way to play an audio file. The **s_play()** request has two arguments. Its second argument is the name of the audio file. The **s_play()** requests expands into the following three requests issued consecutively:

TABLE 18. **The s_play() macro**

```
/* The s_play macro */
s_play(sfd, fname);


/* ... and its equivalent */
s_name(sfd, fname);
s_start(sfd, 0);
s_prepare_play(sfd);
s_continue_play(sfd);
```

Summary of Requests for Playing from Files

The chart below summarizes the usage of the requests used for playing digital audio data from disk files.

TABLE 19. **Requests for Playing From Files**

Requests	Description
void s_name(int sfd, char *fname);	Sets the name of the file to play.
void s_start(int sfd, int tstart);	Provides a starting time, in milliseconds, into the file.
void s_stop(int sfd, int tstop);	Provides a stopping time, in milliseconds, into the file.
void s_prepare_play(int sfd);	Prepares the Audio Server to play a file.
void s_continue_play(int sfd);	Requeusts that the Audio Server begin playing a file.
void s_halt_play(int sfd);	Immediately stops the current play.
void s_play(int sfd, char *fname);	Plays an audio file from its beginning.

Playing Audio Data From Buffers

In some cases, the source of the audio data to be played by a client does not come from a disk file, but rather the client itself. It may wish to do its own file management or generate the audio data synthetically. The Audio Server names this "playing from buffers." The Audio Server API provides two requests for playing from buffers, s_async_play_buf() and s_async_play_new_buf().

The s_async_play_buf() and s_async_play_new_buf() requests

The **s_async_play_buf()** is used to first begin playing. The **s_async_play_new_buf()** is called to provide the Audio Server with additional audio data once the playing is underway. Both requests take two arguments. Their second arguments are pointers for a structure of type **bsm_bytes.**

Using the bsm_bytes structure

Clients sends audio data to the Audio Server over the interprocess communication protocol using the **bsm_bytes** structure. The definition of the **bsm_bytes** structure (found in **bsm.h**, which must be included in all clients) is:

TABLE 20. The bsm_bytes structure definition

```
typedef struct {
int maxLen;
int currLen;
char *data;;
} bsm_bytes;
```

All three fields must be set by the client before sending it to the Audio Server. The *maxLen* field tells the RPC mechanism the absolute maximum about of bytes which can be apart of the structure. The *currLen* field is the number of bytes currently stored in the *data* field. The *data* field, subsequently, is a pointer of type **char *** to the audio data.

Receiving S_PLAY_BUF_EV events

The Audio Server informs the client when it needs more audio data via the S_PLAY_BUF_EV events. As with other Audio Server events, the client must register a callback for this event. The code belows gives the prototype definition for the event and its registration.

TABLE 21. The S_PLAY_BUF_EV event

```
void s_play_buf_handler(int sfd, char *clientData);
{
}


s_register_callback(S_PLAY_BUF_EV, s_play_buf_handler, NULL);
```

Typically, a client registers a callback for the S_PLAY_BUF_EV events and issues an initial **s_async_play_buf**() request. Whenever its callback get invoked, it sends the Audio Server more data with an **s_async_play_new_buf**() request. Although a client does not have to wait for this event notification and send a series of **s_async_play_new_buf**() requests to the server, the RPC mechanism might become bogged down because of the many requests it must queue up.

Signalling the End of the Audio Data

When playing audio from files, the Audio Server knows it has reached the end of the file upon receiving an EOF (end-of-file) indication. When playing from buffers, the client must specifically provide this end-of-file notification.

To tell the Audio Server that it has received all of the audio data for a particular play, the client issues an **s_async_play_new_buf**() request, However, the *currLen* field in the **bsm_bytes** structure it is passed must be set to zero (0).

Note that the **s_halt_play**() request may be issued when playing from buffers, however this immediately stops the playing. If an EOF packet is sent to the Audio Server, the remaining audio data is still played.

Setting the Audio Format for Playing from Buffer

[the s_async_play_rate() request to tell the server the format of the data you are giving it!]

The Requests for Playing from Buffers

The following chart describes the requests for playing from buffer.

TABLE 22. Requests for Playing from Buffers

Requests	Description
void s_async_play_buf(int sfd, bsm_bytes *b);	Plays the first packet in a buffered play.
void s_async_play_new_-buf(int sfd, bsm_bytes *b);	Plays subsequent packets in a buffered play.

An Example of Playing from Buffers

The following code is a skeleton of a client which plays audio from buffers.

TABLE 23. Example of Playing From Buffers

```
static char buffer[3000];
void s_play_buf_handler(int sfd, char *clientdata);
{
bsm_bytes b;
b.maxLen = b.currLen = 3000;
b.data = (char *)buffer;
s_async_play_new_buf(sfd, &b);
}
main()
{
int sfd;
bsm_bytes b;
sfd = s_open_server("s_server", "");
s_register_callback(S_PLAY_BUF_EV, s_play_buf_handler, NULL);
b.maxLen = b.currLen = 3000;
b.data = (char *)buffer;
s_async_play_buf(sfd, &b);
s_main_loop();
}
```

Preprocessing the Audio Data

One advantage of digital audio data is that is can be easily modified before it is played. This sections describes the various processing done on audio data before it is played. A client may request that time-scale modification or gain scaling be performed on the data before it is played. Simple, time-scale modification make the sound play faster or slower, and gain scaling make the sound play louder or softer. These processing methods can be done whether from playing from files or from buffers.

Summary of Preprocessing Requests

The chart below provides the usage for all preprocessing requests.

TABLE 24. **Summary of Preprocessing Requests**

Requests	Description
void s_tsms_ratio(int sfd, double r);	Sets the current time-scaling ratio.
void s_get_tsms_Ratio(sfd, double *ratio);	Returns the current time-scaling ration in **ratio.**
void s_play_gain(int sfd, double gain);	Sets the current play gain.
void s_get_play_gain(int sfd, double *gain);	Returns the current play gain in **gain**.

CHAPTER 6

Recording Digital Audio

Primary Recording

TABLE 25. The s_record_duration() Request

Request	Description
s_record_duration(int sfd, long duration);	Sets the desired maximum duration of the record to **duration** milliseconds.
void s_set_record_rate(int sfd, int rate);	Specifies the desired audio format for recording.
int s_get_record_rate(int sfd);	Returns the current recording audio format.

Recording to Files

Summary of Requests for Recording to Files

TABLE 26. **Requests for Recording to Files**

Requests	Description
void s_name(int sfd, char *fname);	Sets the name of the file to record to.
void s_prepare_record(int sfd);	Prepares the Audio Server to record to a file.
void s_continue_record(int sfd);	Requests that the Audio Server begin recording.
void s_halt_record(int sfd);	Immediately stops the current record.
void s_record(int sfd, char *fname);	Records to an audio file.

Recording to Buffers

Summary of Requests for Recording to Buffers

TABLE 27. **Requests for Recording to Buffers**

Requests	Description
void s_async_record_buf(int sfd);	Prepares the Audio Server to record into buffers.
void s_continue_record(int sfd);	Requests that the Audio Server begins recording.

Receiving Energy Events

Summary of Requests for Energy Events

TABLE 28. **Request for Energy Events**

Requests	Description
void s_set_energy_interval(int sfd, int time);	Tell the Audio Server to send energy events every **time** milliseconds.
int s_get_energy_interval(int sfd);	Returns the current energy interval.

Postprocessing the Audio Data

Summary of Postprocessing Requests

TABLE 29. **Postprocessing Requests**

Requests	Description
void s_record_gain(int sfd, double val);	Sets the current record volume to **val**.
void s_get_record_gain(int sfd, double *val);	Returns the current record volume.
void s_pause_initial(int sfd, int time);	Sets the maximum initial pause to **time** milliseconds.
void s_pause_final(int sfd, int time);	Sets the maximum final pause to **time** milliseconds.
void s_pause_detect(int sfd, int set);	Sets pause detection and record termination on if **set** equals 1, otherwise off.
void s_set_dtmf_detect(int sfd, int set);	Turns touch-tone detect on if **set** equals 1, otherwise off.
void s_set_dtmf_termination(-int sfd, int set);	Terminates the recording when a touch-tone is heard, if **set** equal 1.

CHAPTER 7

Multi-Client Operation and Audio Device Resource Management

Throughout the past couple of chapters, an assumption was made: only a single client would be either playing or recording at one time. In truth, this was only a simplifying assumption. This assumption is now removed--this chapter discusses how the Audio Server mediates between multiple clients wishes to play and/or record at a single time. The ability of an Audio Server to acknowledge requests from a number of separate clients is a main argument of an Audio Server versus a client-side only library.

Audio Device Resource Management

Central to the ability of handling requests from multiple clients, is how to multiplex or prevent clients from using the audio device. The series of rules which govern how the audio device is appropriated is called *audio resource management.* Here, the single audio device is a resource which is in demand by more than one client. These rules may decide that a particular client may play audio while another has to wait. It also decides when client may receive recorded audio data and when they may not.

Client Priority

Not all clients' requests can be serviced at once; some client may have to wait until the Audio Server completes the requests of other clients. While a first-come, first-served scheme is easy to implement, it may not represent the true ordering of each requests' importance. A first-come, first-served scheme is combined with a *client priority* scheme in order to determine the order in which requests are serviced.

Each client may declare itself either a low, medium, or high priority client. While a mis-behaving client may declare itself as the highest priority, the Audio Server relies on clients accurately declaring their priorities. The request **s_set_client_priority()** is used to inform the Audio Server of a client's priority. It takes two arguments, the first is the connection file descriptor. the second is the clients priority, chosen from the following: S_PRIORITY_LOW, S_PRIORITY_MEDIUM, S_PRIORITY_HIGH. All clients who do not declare a priority default to S_PRIORITY_MEDIUM. Clients may change their priority at any time while connected to the Audio Server.

With respect to playing, the intention of the high/medium/low priority scheme is clients which need to play audio urgently will declare their priority to be high. Examples of high priority sounds include those which the user must here as soon as possible, such as an incoming phone call indication. Medium priority sounds are those which are normally and most commonly played on the workstation, such as voice mail. Low priority sounds typically originate from background audio applications, such as an audio email notification or an hourly chime.

This three level priority scheme is also available for primary recordings as well (see the previous Chapter). In practice, recording tends to assume more of a bi-level, foreground and background, priority structure. Specifying a client priority when performing recording is uncommon.

Resource Management for the Playing Device

When multiple clients request to play digital audio data, the Audio Server services one play request at a time. It maintains a queue of requests, sorted first by decreasing order of priority, and then sorted on a first-come, first-served basis. The following algorithm outlines the policy the Audio Server uses to determine which request to service:

- When finishing a play request, the Audio Server services the request at the head of the queue. That is, the highest priority request is played next. If there are multiple requests of the same, highest priority, the first one requested in served.
- If a play request is received or higher priority than a current play request underway, the current play request is interrupted and placed on the queue. The new play request is immediately serviced.

In comparison to a first-come, first-served only scheme, the Audio Server allows newer, high priority requests to be serviced immediately. Those clients which have play requests interrupted received an S_PLAY_INTERRUPTED_EV event, and an S_PLAY_CONTINUE_EV event which their request is eventually resumed. When a play is interrupted, the Audio Server attempts to restart the playing a couple of seconds earlier than it had been interrupted. This makes it easier for the listener to readjust to the previous context when the playing is resumed. Note that this backup feature is only done when playing from *files*. If a client does not wish for its request to resume after an interruption, it can issue an **s_halt_play**() request when it receives the S_PLAY_INTERRUPTED_EV event.

The following example illustrates the interaction among various clients which issue play requests in the sequence below:

- Client A issues a play request at medium priority. The Audio Server services Client A's request.
- Client B issues a play request at medium priority. The Audio Server places Client B's request at the head of the request queue.
- Client C issues a play request at high priority. The Audio Server interrupts Client A and places it at the head of the request queue. Client C's request is serviced.
- Client C's request completes. Client A's request is continued.
- Client D issues a play request at low priority. Its request is placed at the end of the queue.
- Client A's request completes, Client B's request is serviced.
- Client B's request completes, Client D's request is serviced.

Resource Management for the Recording Device

Managing client's record requests is nearly identical to managing play requests. (Note that only primary record requests are managed in this fashion; primary recording was described in the previous chapter. Background recording is dealt with below.) The single difference between playing and recording is that when a record request is interrupted, it is halted rather than just interrupted. In other words, these record request are not placed on a queue and therefore, are not continued later. Instead of receiving an S_INTERUPTED_EV event, the client receives and S_RECORD_DONE_EV event.

Registering Interest In Audio Server Activity

Certain clients may wish to perform duties when other clients use the Audio Server for a specific purpose. For example, a VU Meter only wants to indicate the recording level when a recording is actually taking place. Clients may issue a request which allows them to be notified of various audio server activities, including receiving record data while these activities are underway. This is sometimes known as *background recording*, but it will be referred to as *data* and *event interests*.

When a client registers an event interest, it received events when the activity it is interested in both begins and ends. A client which registers a data interest received record data from the workstation's microphone while the activity is taking place. In particular, for registered event interests, the client receives S_INTEREST_-EVENT_BEGIN_EV and S_INTEREST_END_EV events. For registered data interests, the client receives S_INTEREST_DATA_BEGIN_EV, S_INTEREST_-DATA_END_EV, and S_INTEREST_DATA_EV events.

The **s_register_event_interest**() and **s_register_data_interest**() request allows client to register event and data interests. The prototypes of these requests are found below.

TABLE 30. **Registering Event and Data Interests**

```
void s_register_event_interest(int sfd, int interest);
void s_register_data_interest(int sfd, int bufsize, int intent, int interest);
```

What is this all About?

Clients use recorded audio data for many different purposes. Some example are: storage, dtmf (touch-tone) detection, speech recognition, and a vu meter display. Different clients, however, may want recorded audio data only under certain circumstances. Hence, they have a particular *interest* in what other clients are doing. As mentioned before, a vu meter client is interested in data when other clients are recording audio data for storage.

Suppose a client wants recorded audio data only when another client is performing speech recognition. How does the Audio Server know that a client is performing speech recognition? Each client must declare its *intent*, or what is plans to do with the audio data. Therefore, the Audio Server provides recorded audio data to clients based on their interests. These interests are satisfied by examining the declared intents of other clients which currently have active requests. In other words, the intents of some clients are used to determine if the interests of other clients are satisfied.

Interests

The second argument to **s_register_event_interest()** and the fourth argument to **s_register_data_interest()** is an *interest*. An interest is a mask of one or many Audio Server activities. The client will receive events or data, respectively, if any one of these activities are underway in the Audio Server. The following chart explains the possible interests a client may have:

TABLE 31. **Audio Server Interests**

Interests	Descriptions
S_INTEREST_NONE	Interest when there is no activity underway
S_INTEREST_RECORD	Interest when a client is recording audio data
S_INTEREST_DTMF	Interest when a client is performing touch tone detection on recorded data.
S_INTEREST_ENERGY	Interest when a client is performing energy calculations on recorded data.
S_INTEREST_VUMETER	Interest when a client is acting as a VU Meter on recorded data.
S_INTEREST_RECOGNITION	Interest when a client is performing speech recognition on recorded data.
S_INTEREST_PLAYING	Interest when a client is playing audio data.
S_INTEREST_CONNECTION	Interest when a new client makes a connection (event registration only)
S_INTEREST_LOCKED	Interest when a client locks the audio server (event registration only)
S_INTEREST_ALWAYS	A client is always interested in receiving recorded data.

The client may specify multiple interest by ORing these definition together. For example, if a client wants to register and interest when a client is either recording of performing speech recognition, is provides S_INTEREST_RECORD | S_INTEREST_RECOGNITION as the interest.

Intents

The third argument to the **s_register_data_interest()** request is an *intent.* This intent is a declaration by the client of its purpose for receiving recorded audio data. The possible intents are: S_INTENT_NONE, S_INTENT_RECORD, S_INTENT_DTMF, S_INTENT_ENERGY, S_INTENT_VUMETER, and S_INTENT_RECOGNITION.

Data and Event Interests In Practice

Although the Audio Server allows clients to receive recorded audio data under complicated circumstances based on their interests, only very few of the interests are used in practice. For example, S_INTEREST_RECORD and S_INTEREST_ALWAYS are most commonly used. These client either want recorded data only when other clients are receiving data for recording, or these clients want data all of the time. It is very rare that a client would want to obtain record data when another client is acting as a VU Meter, for example.

Examples

The following are examples of code which make interest requests for various purposes:

TABLE 32. **Examples of Data Interests**

```
/* A VU Meter client only wants to display the record level when another client is
recording */
s_register_data_interest(sfd, 10, S_INTENT_VUMETER, S_INTEREST_RECORD);

/* A client to record all noise in an office all of the time */
s_register_data_interest(sfd, 4000, S_INTENT_RECORD, S_INTEREST_ALWAYS);
```

Unregistering Interests

When a client no longer has an interested in an Audio Server activity, it may unregister that interest with the **s_unregister_event_interest()** and **s_unregister_-data_interest()** requests. These requests take two arguments. The second argument is an interest. A client must unregister the interests in which it had originally registered an interest. It may, however, unregister only some of the interests.

Multi-Client Requests

The following chart summarizes the requests available for multi-client operation purposes:

TABLE 33. **Multi-Client Requests**

Requests	Description
void s_set_client_priority(int sfd, int priority)	Sets the priority of the client to either high, medium, or low.
void s_register_event_interest(int sfd, int interest);	Registers an event interest.
void s_register_data_interest(int sfd, int bufsize, int intent, int interest);	Register a data interest. The client receives data in buffers no larger than **bufsize**.
void s_unregister_event_interest(int sfd, int interest);	Unregisters an event interest.
void s_unregister_data_interest(int sfd, int interest);	Unregisters a data interest.

CHAPTER 8

The Sound File Directory

The example Audio Server client play.c is an example of some of the sound file management handled by the Audio Server. In addition to play from and recording to audio files, the Audio Server provides basic audio file management. This chapter describes the features it provides.

For each client, the Audio Server maintains a "current working sound directory." It is under this directory which the Audio Server searches for all non-absolulte audio file names. This is true for all operations (e.g. play, record) on audio files; this behavior is kept consistent throughout.

The Default Sound Directory

The Audio Server sets the default working sound directory for each client based on the SOUND environment variable, and the user associated with the client. The SOUND environment variable provide a base directory for the sound directory. Each user has a subdirectory off of this base directory. For example, suppose user "hordack" execute a client and asks the Audio Server to play a file named "greeting.snd." Also, suppose the SOUND environment variable was set to "/sound". The Audio Server would search for the file named "/sound/hordack/greeting.snd."

The SOUND environment variable must be set before the Audio Server is run, and in the same environment as the Audio Server. To set the SOUND environment variable, type:

TABLE 34. **Setting the SOUND environment variable**

```
% setenv SOUND /sound
```

When the client issues the **s_open_server**() command to initialize a connection with the Audio Server, the user associated with the client is send to the Audio Server. The client uses the **getpwuid**() UNIX system call to obtain the identify of the owner of the client process. Therefore, a user is simply required to execute the client as him/herself (in addition to setting SOUND properly) in order for the current working sound directory to default correctly.

In some cases, the user of the client might not be the owner of the process. The Audio Server provides a means to clients to manually set the current user. This, in turn, sets the current working sound directory to $SOUND/<user>, where <user> is the specified login name. For example, to change the current working sound directory to "geek", use the following Audio Server request:

TABLE 35. **Changing the current user**

```
s_user(audiofd, "geek");
```

The **s_user**() request has two arguments. The first is the connection fd to the Audio Server and the second is a string of the new login name. **s_user**() may be issued at any time while a connection is actively open to an Audio Server and may be called any number of times. Note, however, that the current working sound directory for the client is changed immediately after the **s_user**() request is invoked. The **s_user**() request is summarized below:

TABLE 36. **Description of the s_user request**

Request	**Description**
s_user(int fd, char *login);	Sets the current user and sound directory

Utility Routines for Managing Directories

Not all users structure the location of their audio files as provided by the default sound directory used by the Audio Server. Therefore, the Audio Server provides some basic sound directory creation, deletion, and management requests. Although the standard UNIX system calls may replace these Audio Server requests, it is recommended to use the latter rather than the former.

The **s_cd**() request changes the current working sound directory. All non-absolute, future file names given to the Audio Server will be assumed to be located under the new sound directory. The **s_cd**() request takes two arguments: a conenction fd, and the name of a new sound directory.

s_mk_dir() and **s_rm_dir**() create and remove directories, respectively. These requests have the same requirements as the UNIX system calls. That is, a client cannot successfully create an existing directory, nor can it remove a non-empty directory. Both requests have two arguments, the second of which is a path name.

s_get_dir() and **s_get_path**() are identical. Both return the name of the current working sound directory. Their first argument is a connection fd to an Audio Server. The second is a pointer to an allocated portion of memory large enough to hold a path name.

The Audio Server provides two simple requests to access the subdirectories under the current working sound directory, **s_find_first_dir**() and **s_find_next_dir**(). To find the first subdirectory, the **s_find_first_dir**() request is used. Subsequent queries use the **s_find_next_dir**() request. The **s_find_first_dir**() request takes three arguments: a connection fd, a search template, and a pointer to allocated memory for a resulting directory. The client is able to filter for certain types of subdirectories with the second argument. This template argument has the same semantics as used in UNIX calls. This means that such wildcard characters as "?" and "*" are permissible. The third argument must be allocated for the longest possible directory. The **s_find_next_dir**() request has two arguments. It uses the template argument provided to the most recent call to **s_find_first_dir**() and, therefore, omits that argument.

A summary of the available directory management routines is found in the chart below:

TABLE 37. Directory managment Requests

Requests	Description
void s_cd(int fd, char *path);	Changes the current working sound directory to **path.**
void s_mk_dir(int fd, char *path);	Makes a new sound directory named **path.**
void s_rm_dir(int fd, char *path);	Removes the sound directory named **path.**
void s_get_dir(int fd, char *dir);	Returns the current working sound directory in **dir.**
void s_get_path(int fd, char *dir);	Returns the current working sound directory in **dir.**.
int s_find_first_dir(int fd, char *template, char *match);	Returns in **match**, the first subdirectory in the current working sound directory. The returned directory name matches the filter **template**. Returns TRUE if a matching directory was found, FALSE otherwise.
int s_find_next_dir(int fd, char *match);	Returns the next subdirectory under the current working sound directory in **match.** The returned directory name matches the filter **template** given to the most recent **s_find_first_dir**() call. Returns TRIE if a matching directory was found, FALSE otherwise.

Utility Routines For Managing Audio Files

The following Audio.Server requests provide simple functionality for managing (e.g. copying, moving) audio files. They provide a convenient and easy interface to the standard UNIX system calls. These requests also reference files in relation to the current working sound directory.

The **s_file_exists**() requests asks the Audio Server to see if a given audio file exists. It takes two arguments, the second of which is a file name. This file name is assumed to be located under the current working sound directory, unless it is given as an absoulte path name. **s_file_exists**() returns TRUE if the file does indeed exist, false otherwise.

The **s_mv**() requests moves one audio file to another. The second argument to **s_mv**() is the source audio file and the third argument is the destination audio file name. Both of these file names are assumed to exist under the current working sound directory if they are not specified as absolute path names.

The **s_rm**() request deletes a specified audio file. The second argument of **s_rm**() is the audio file name which to remove. **s_cp**() copies the audio file name given as the second argument to the audio file name given as the third argument.

The **s_find_first**() and **s_find_next**() requests are identical to the **s_find_first_dir**() and **s_find_next_dir**() requests as described previously. The only difference is that **s_find_first**() and **s_find_next**() operates on files rather than directories. These requests return the file names under the current working sound directory.

The table below summarizes the file management requests:

TABLE 38. **File Management Requests**

Requests	**Description**
int s_file_exists(int fd, char *fname);	Returns TRUE if **fname** exists, otherwise returns FALSE. **fname** is assumed to be located under the current working sound directory if not given as an absoulte path.
void s_mv(int fd, char *from, char *to);	Moves the audio file **from** to the audio file **to**.
void s_rm(int fd, char *fname);	Deletes the audio file **fname**.
voi s_cp(int fd, char *from, char *to);	Copies the audio file **from** to the audio file **to**.

TABLE 38. **File Management Requests**

Requests	Description
int s_find_first(int fd, char *template, char *match);	Returns in **match**, the first file in the current working sound directory. The returned file name matches the filter **template**. Returns TRUE if a matching file was found, FALSE otherwise.
int f_find_next(int fd, char *match);	Returns the next file under the current working sound directory in **match**. The returned file name matches the filter **template** given to the most recent **s_find_first**() call. Returns TRIE if a matching file was found, FALSE otherwise.

```
/*
 *
 * Streams library module. Implements the stream abstraction on top
 * of swindows.
 *
 * Atty Mullins
 * Speech Research Group, MIT Media Lab
 * 1994
 *
 */

/*
 * Standard Includes
 */
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <termios.h>
#include <math.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <swindows.h>
#include <s_app.h>
#include <sparc_sound.h>
#include <db.h>
#include <sndnote.h>
#include <varray.h>
#include <stream.h>
#include <cre_midi.h>

/*
 * Constants
 */
#define SILENCE_BUFFER_LENGTH 8000
#define START_DELAY_TICKS 3
#define TICK_SIZE 20
#define DISPLAY_LENGTH 30
#define UNFOCUS_RATE (float)15.00
#define CHNL 0
#define INSTR 12
#define VELOCITY 63
#define FISH_UPDATE_WINDOW 1200

/*
 * Local Variables
 */
static swindow *sw1, *sw2, *sw3, *sw4;
static stream *s1, *s2, *s3, *s4;
static stream *last_gaze;
static stream *gaze;
static stream *focus;
static bsm_bytes silence_bb;
static unsigned char silence[SILENCE_BUFFER_LENGTH];
static char erase_string[DISPLAY_LENGTH];
static time_t last_tone_time = 0;
static int polhemus_rw;
static int fish_rw;
static int fish_ticks;
static int track_head;
static int output_fd;
static int feedback;
static int last_pos_index = 0;
static float lrh[FISH_TICKS];
static float llh[FISH_TICKS];
static float lra[FISH_TICKS];
static float lla[FISH_TICKS];
static float lrhv[FISH_TICKS];
static float llhv[FISH_TICKS];
static float last_left_head_velocity;
static float last_right_head_velocity;
static time_t last_time = 0;
static time_t base_time = 0;
static int first_time;

static float level_0_left_gain;
static float level_1_left_gain;
static float level_2_left_gain;
static float level_3_left_gain;
static float level_4_left_gain;

static float level_0_right_gain;
static float level_1_right_gain;
static float level_2_right_gain;
static float level_3_right_gain;
static float level_4_right_gain;

static float level_0_middle_gain;
static float level_1_middle_gain;
static float level_2_middle_gain;
static float level_3_middle_gain;
static float level_4_middle_gain;

static float level_0_left_rate;
static float level_1_left_rate;
static float level_2_left_rate;
static float level_3_left_rate;
static float level_4_left_rate;

static float level_0_right_rate;
static float level_1_right_rate;
static float level_2_right_rate;
static float level_3_right_rate;
static float level_4_right_rate;

static float level_0_middle_rate;
static float level_1_middle_rate;
static float level_2_middle_rate;
static float level_3_middle_rate;
static float level_4_middle_rate;

static int  tone_time_interval;
static int  focus_time_window;
static int  tsms_ticks;
static int  gain_delay_ticks;
static int  middle_start_delay;
static int  right_start_delay;
static int  left_start_delay;
static double  tsms_ratio;
static double  max_tsms_ratio;
static double  tsms_rate;

/*
 * Local Function Declarations
 */

/* Play Functions */
static void play_tone(stream *s,
                      char *filename,
                      int delay);
static void play_segment(stream *s,
                          segment *current_segment,
                          long start,
                          long stop);
static void play_segment_on_stream(stream *s,
                                   VARRAY *segments,
                                   int segment_index);
static void play_empty_buf_on_stream(stream *s);

/* Gain Functions */
static void s_delayed_gain(stream *s,
                           float gain,
                           int ticks);
static void decay_gain(stream *s);
static void delay_gain(stream *s);
static void update_gain(stream *s);
static void update_tsms_ratio(stream *s);

/* Callbacks */
static void segment_cb(int fd,
                       char *client_data,
                       int termination);
static void buffer_cb(int fd,
                      char *clientdata,
                      int termination);
static void time_cb(caddr_t *data);

/* Initialization Functions */
static void initialize_swindows();
static void read_streamer_config_file(char *filename);
static void init_streamer_config();
static thread *initialize_thread(char *filename);
static VARRAY *initialize_segments(char *filename,
                                     int *num_segments_ptr);
static stream *initialize_stream(swindow *sw,
                                 thread *t,
                                 float level_0_gain,
                                 float level_1_gain,
                                 float level_2_gain,
                                 float level_3_gain,
                                 float level_4_gain,
                                 float level_0_rate,
                                 float level_1_rate,
                                 float level_2_rate,
                                 float level_3_rate,
                                 float level_4_rate,
                                 int start_delay_ticks);
/* Storage */
static void free_thread(thread *t);
static void free_stream(stream *s);

/* Polhemus Interface */
static int init_polhemus_rw(char *device);
static float *fetch_polhemus(float *location);
static float *update_head_position();
static void update_gaze(float yaw, float pitch);

/* Fish Interface */
static int init_fish_rw(char *device);
static void fetch_fish(float *sensor1,
                       float *sensor2,
                       float *sensor3,
                       float *sensor4);
static void update_fish();

/* Miscellaneous Functions */
static int stagger_start(stream *s);
static long current_pos(stream *s);
static void pause_stream(stream *s);
static void continue_stream(stream *s);
static void play_level(int level, stream *s);

/*
 * Function Definitions
 */

/*
 * Opens the device to read and write to the polhemus.
 */
static int
init_polhemus_rw(char *device)
{
    struct termios temp_termios;
    int fd;

    fd = open( device, O_RDWR, 0);
    tcgetattr(fd, &temp_termios);

    cfsetospeed(temp_termios, B19200);
    cfsetispeed(temp_termios, B19200);

    /* Clear some flags */
    temp_termios.c_oflag &= ~(ONLCR | OPOST);
    temp_termios.c_iflag &= ~(ICRNL | IXOFF | INPCK);
    temp_termios.c_lflag &= ~(ECHO | XCASE  | ISIG | ICANON) ;
    temp_termios.c_cflag &= ~(PARENB | PARODD);

    /* And set others */
    temp_termios.c_cflag |= CS8;

    /* Set and update */
    tcsetattr(fd, TCSANOW, &temp_termios);
    tcflush(fd, TCIOFLUSH);

    /* Don't block */
    fcntl(fd,F_SETFL,O_NDELAY);

    /* Raw binary */
    write(fd,"cFK", 3);

    return fd;
}

/*
 * Fetches one location vector from the polhemus.
 */
static float
*fetch_polhemus(float *location)
{
  int nread, bytes_read, status;
  int reply_length = 47;
  char buffer[128];
  float x, y, z, yaw, pitch, roll;

  /* Request a status record */
  write(polhemus_rw, "P", 1);

  bytes_read = 0;

  /* Process the reply and print it out */
  while(bytes_read < reply_length)
    {
      nread = read(polhemus_rw, &(buffer[bytes_read]),
                    reply_length - bytes_read);
      if(nread > 0)
        bytes_read += nread;
    }
  /* Process the input into floats */
  sscanf(buffer, "%d%f%f%f%f%f%f", &status, location, (location + 1),
         (location + 2), (location + 3), (location + 4), (location + 5));

  return location;
}

/*
 * Determines gaze (head pointing) from polhemus and updates focus
 * accordingly.
 */
static void
update_gaze(float yaw,
            float pitch)
{

  /* Update the gaze */
  if(pitch < -10.00)
    {
      gaze = (stream *)NULL;
      s_unfocus();
    }
  else
    {
      if((yaw <= 90.00) && (yaw > 15.00))
        gaze = left;
      else if((yaw >= 270.00) && (yaw < 340.00))
        gaze = right;
      else if(pitch >= 10.00)
        gaze = center;
      else
        gaze = (stream *)NULL;

      /* Update the focus */
      if((last_gaze != gaze) &&
         (gaze != (stream *)NULL))
        s_give_focus(gaze);
    }

  /* Keep track of the gaze */
  last_gaze = gaze;
}

/*
 * Updates the head position from polhemus information.
 */
static float
*update_head_position()
{
  float temp[6];

  fetch_polhemus(temp);

  /* Only update the yaw, pitch and roll of the head */
  temp[0] = 0.0;
  temp[1] = 0.0;
  temp[2] = 0.0;

  /* Yaw is positive counter-clockwise */
  temp[3] = temp[3] + 180.00;

  /* Pitch ranges from + 90 to - 90 */
```

```
    temp[4] = -(temp[4]);

    /* Roll */
    temp[5] = temp[5] + 180.00;


    /* Update the gaze if necessary */
    if(track_gaze)
      update_gaze(temp[3], temp[4]);

    /* Convert to radians */
    temp[3] = temp[3] / 57.3;
    temp[4] = temp[4] / 57.3;
    temp[5] = temp[5] / 57.3;

    /* Move the head */
    return temp;
  }

  /*
   * Initializes the device to read and write to the fish.
   */
  init_fish_rw(char *device)
  {
      struct termios temp_termios;
      int fd;

      fd = open( device, O_RDWR, 0);
      tcgetattr(fd, &temp_termios);

      cfsetospeed(temp_termios, B9600);
      cfsetispeed(temp_termios, B9600);

      /* Clear some flags */
      temp_termios.c_oflag &= ~(ONLCR | OPOST);
      temp_termios.c_iflag &= ~(ICRNL | IXOFF | INPCK);
      temp_termios.c_lflag &= ~(ECHO | XCASE  | ISIG | ICANON) ;
      temp_termios.c_cflag &= ~(PARENB | PARODD);

      /* And set others */
      temp_termios.c_cflag |= CS8;

      /* Set and update */
      tcsetattr(fd, TCSANOW, &temp_termios);
      tcflush(fd, TCIOFLUSH);

      /* Don't block */
      fcntl(fd,F_SETFL,O_NDELAY);

      fish_ticks = 0;

      return fd;
  }

  /*
   * Fetches the status of the four sensors from the fish.
   */
  static void
  fetch_fish(float *sensor1,
             float *sensor2,
             float *sensor3,
             float *sensor4)
  {
    int nread, bytes_read, status;
    int reply_length = 16;
    char buffer[128];
  /* Set up the output format */

  /* Request a status record */
  write(fish_rw, "R", 1);

  bytes_read = 0;

  /* Process the reply and print it out */
  while(bytes_read < reply_length)
    {
      nread = read(fish_rw, &(buffer[bytes_read]),
                   reply_length - bytes_read);
      if(nread > 0)
        bytes_read += nread;
    }
  /* Process the input into floats */
  sscanf(buffer, "%f %f %f %f", sensor1, sensor2, sensor3, sensor4);
}

/*
 * Updates focus etc based on the fish.
 */
static void
update_fish()

{
  float left_arm, right_arm, left_head, right_head;
  float left_head_velocity, right_head_velocity;
  float left_arm_velocity, right_arm_velocity;
  float left_head_accelaration, right_head_accelaration;
  struct timeb tp;
  time_t this_time;
  int i;

  /* Query the fish */
  fetch_fish(&left_arm, &right_arm, &left_head, &right_head);

  if(fish_ticks == 0)
    {
      /* Reset the clock */
      fish_ticks = FISH_TICKS;

      /* Calculate velocity and accelaration */
      i = abs(last_pos_index - 2);
      left_head_velocity = llh[i] - left_head;
      right_head_velocity = lrh[i] - right_head;
      left_arm_velocity = lla[i] - left_arm;
      right_arm_velocity = lra[i]  - right_arm;
      left_head_accelaration =
        last_left_head_velocity - left_head_velocity;
      right_head_accelaration =
        last_right_head_velocity - right_head_velocity;

      /* Update everything */
      last_right_head_velocity = right_head_velocity;
      last_left_head_velocity = left_head_velocity;

      /* Geek 6/29/94: go to relative time */
      if (first_time == 0) {
        ftime(&tp);
        base_time = tp.time;
        last_time = 0;
        first_time = 1;
      }
    /* Fetch the current time */
    ftime(&tp);
    this_time = ((tp.time-base_time) * 1000)  + tp.millitm;

    /*      printf("subtraction: %d\n", (this_time - last_time));
            printf("this %ld, last %ld, %ld, %d\n", this_time, last_time,
            tp.time, tp.millitm);  */
    if ((this_time - last_time) > FISH_UPDATE_WINDOW)
      {
        /* Check arms first */
        /*if((left_arm < 255) &&
          (right_arm < 255))
        {
        sw_gain(center->sw, OFF_GAIN);
        last_time = this_time;
        }
        else if ((left_arm < 255) &&
        (right_arm == 255))
        {
        sw_gain(left->sw, OFF_GAIN);
        last_time = this_time;
        }
        else if ((left_arm == 255) &&
        (right_arm < 255))
        {
        sw_gain(right->sw, OFF_GAIN);
        last_time = this_time;
        }
        else*/
      {
        /* Compute head point */
        if((left_head_accelaration >= 2.0) &&
           (right_head_accelaration >= 2.0)) {
          gaze = center;
        }
        else if((left_head_velocity <= -1.0) &&
                (right_head_velocity >= 2.0)) {
          gaze = left;
        }
        else if ((right_head_velocity <= -2.0) &&
                 (left_head_velocity >= 1.0)) {
          gaze = right;
        }
        else
          gaze = (stream *)NULL;

        /* Update focus */
        if((last_gaze != gaze) &&
           (gaze != (stream *)NULL))
          {
            last_time = this_time;
            s_give_focus(gaze);
          }

        last_gaze = gaze;
      }
    }
  }

else
  /* Decrement clock */
  fish_ticks--;

/* Add sensor values to FIFOs */
lrh[last_pos_index] = right_head;
llh[last_pos_index] = left_head;
```

```
  lra[last_pos_index] = right_arm;
  lla[last_pos_index] = left_arm;

  /* Update the index */
  last_pos_index = (++last_pos_index)%FISH_TICKS;

}

/*
 * Plays feedback in swindow.
 */
static void
play_tone(stream *s,
          char *filename,
          int delay)
{
  time_t current_time;

  /* If there's a filename play it */
  if(filename != (char *)NULL)
    {
      if(delay)
        {
          time(&current_time);
          if ((current_time - last_tone_time) <
              tone_time_interval)
            sleep(tone_time_interval -
                  (current_time - last_tone_time));
        }
      /* Save the last time */
      time(&last_tone_time);
      sw_play(s->sw, filename);
    }
}

/*
 * Plays a sound segment on a stream. Records starting time as well.
 */
static void
play_segment(stream *s,
             segment *current_segment,
             long start,
             long stop)
{
  /* Record the start time and position of this segment */
  ftime(s->start_time);
  s->start_pos = start;
  s->stop_pos = stop;


  /* Play the segment */
  sw_play_segment(s->sw, current_segment->audio_file,
                  start, stop);
}

/*
 * Playes a particular segment on a stream. Called from segment_cb
 * only. Typical interface is play_segment.
 */
static void
play_segment_on_stream(stream *s,
                       VARRAY *segments,
                       int segment_index)
{
  /* Restart any paused streams */
  if(s1 != s)
```

```
    continue_stream(s1);
  if(s2 != s)
    continue_stream(s2);
  if(s3 != s)
    continue_stream(s3);

  /* Reset gain */
  if(s->gain_level != 0)
    (s->gain_level)--;

  /* If we're not at a fixed gain, then bump gain */
  if(s->gain_level == 0)
    s_delayed_gain(s, s->gains[1], gain_delay_ticks);

  /* Reset the tsms ratio and ticks */
  s->last_tsms_ratio = s->tsms_ratio;
  s->tsms_ratio = tsms_ratio;
  s->tsms_ticks = 0;
  s_tsms_ratio(s->sw->id, s->tsms_ratio);

  s->current_segment = (segment *)VarrayGet(segments, segment_index);
  s->segment_index = segment_index;

  play_segment(s,
               s->current_segment,
               s->current_segment->start,
               s->current_segment->stop);
}

/*
 * Loops an empty, i.e., silent buffer into stream s. This is
 * necessary because of a bug in stereoclient.
 */
static void
play_empty_buf_on_stream(stream *s)
{
  s_async_play_buf(s->sw->id, &silence_bb);
}

/*
 * Sets a timer ticks long after which the gain get set to gain.
 */
static void
s_delayed_gain(stream *s,
               float gain,
               int ticks)
{
  s->delayed_gain = gain;
  s->gain_delay_ticks = ticks;
}

/*
 * Decays the gain over time. On the gain is completely decayed the
 * gain_level (focus) is set to zero.
 */
static void
decay_gain(stream *s)
{
  float temp_gain;

  /* Decay the gains on the windows */
  if((temp_gain = sw_get_gain(s->sw)) > s->gains[0])
    sw_gain(s->sw, (float)(temp_gain - s->rates[s->gain_level]));
  else
    s->gain_level = 0;
}
```

```
/*
 * Checks to see if the delay timer has expired and if so sets the
 * gain  appropriately.
 */
static void
delay_gain(stream *s)
{
  /* Check if gain delay is on */
  if(s->gain_delay_ticks > 0)
    {
      s->gain_delay_ticks -= 1;
      if(s->gain_delay_ticks == 0)
        {
          s->gain_delay_ticks = 0;
          sw_gain(s->sw,
                  s->delayed_gain);
        }
    }
}

/*
 * Handles gain decay and sets any delayed gains as appropriate.
 */
static void
update_gain(stream *s)
{
  decay_gain(s);
  delay_gain(s);
}

/*
 * Updates the playback speed when focus level is three or greater.
 * We need to halt the sound, change the speed, and continue playing.
 *
 */
static void
update_tsms_ratio(stream *s)
{
  if(s->gain_level >= 4)
    {
      s->tsms_ticks++;
      if((s->tsms_ticks%tsms_ticks) == 0)
      {
        if(s->tsms_ratio < max_tsms_ratio)
          {
            /* Time to update the tsms ratio */
            s->tsms_ratio += tsms_rate;
            s_tsms_ratio(s->sw->id, s->tsms_ratio);
          }
      }
    }
}

/*
 * Callback that loops segments onto a stream. If termination condition
 * is S_P_EOF, then the next segment is played.
 */
static void
segment_cb(int fd,
           char *client_data,
           int termination)
{
  segment *temp;
  thread *t;
  stream *s;
```

```
    /* Figure out which segment finished playing */
    /* and play the next */
    if (fd == s1->sw->id)
      {
        s = s1;
        t = s1->thread;
      }
    else if(fd == s2->sw->id)
      {
        s = s2;
        t = s2->thread;
      }
    else if (fd == s3->sw->id)
      {
        s = s3;
        t = s3->thread;
      }
     else if (fd == s4->sw->id)
       t = (thread *)NULL;

    if(t != (thread *)NULL)
      {
        if(termination == S_P_EOF)
          {
            /* Play the attention tone */
            if(s->attention == FALSE)
              {
                if(t->current_segment_index  < t->num_segments)
                  t->current_segment_index++;

                /* We're going to play something here */
                s->attention = TRUE;

                if(t->current_segment_index < t->num_segments)
                  play_tone(s, ATTENTION_TONE, TRUE);
                else
                  play_tone(s, NO_SEGMENT_TONE, FALSE);
              }
            else
              /* Now play the correct segment */
              {
                s->attention = FALSE;

                /* Play next segment */
                if (t->current_segment_index < t->num_segments)
                  {
                    if(s->paused_pos > 0)
                      {
                        play_segment(s, s->current_segment,
                                     s->paused_pos,
                                     s->current_segment->stop);
                        s->paused_pos = -1;
                      }
                    else
                      play_segment_on_stream(s, t->segments,
                                             t->current_segment_index);
                  }
              }
          }
      }
  }

  /*
   * Callback that keeps a buffer of silence playing on the fourth
   * stream. FIXES a stereoclient bug that causes the fourth stream to
```

```
 * stop playing.
 */
static void
buffer_cb(int fd,
          char *clientdata,
          int termination)
{
  /* Just keep playing an empty buffer */
  s_async_play_new_buf(fd, &silence_bb);
}

/*
 * Callback that handles time dependent changes (delays and gain
 * decay). Gets called every TICK_SIZE milliseconds.
 */
static void
time_cb(caddr_t *data)
{
  float temp_gain;
  float *location;
  char outstring[DISPLAY_LENGTH];
  char c1, c2, c3;
  int i;

  if(output_fd >= 0)
    {
      /* Print some status info if necessary */
      c1 = ' ';
      c2 = ' ';
      c3 = ' ';

      if (left == focus)
        c1 = '*';
      if (center == focus)
        c2 = '*';
      if (right == focus)
        c3 = '*';

      for(i = 0; i < DISPLAY_LENGTH; i++)
        outstring[i] = ' ';

      sprintf(outstring, " %c%5.2f  %c%5.2f  %c%5.2f", c1, left->sw->gain,
              c2, center->sw->gain, c3, right->sw->gain);
      write(output_fd, erase_string, DISPLAY_LENGTH + 1);
      write(output_fd, outstring, DISPLAY_LENGTH);
    }


  if(polhemus_rw >= 0)
    {
      location = update_head_position();
      if(track_head)
        sw_move_head(location);
    }

  if(fish_rw >= 0)
    update_fish();

  /* Update the gains on the streams */
  if(stagger_start(s1))
    {
      update_gain(s1);
      update_tsms_ratio(s1);
    }

  if(stagger_start(s2))
```

```
      {
        update_gain(s2);
        update_tsms_ratio(s2);
      }

    if(stagger_start(s3))
      {
        update_gain(s3);
        update_tsms_ratio(s3);
      }
}

/*
 * Initialize the swindows system. Allocates four windows and puts
 * them into their default positions. Also sets the gains properly.
 */
static void
initialize_swindows()
{
  sw_init();

  /* Allocate swindows */
  sw1 = sw_open();
  sw2 = sw_open();
  sw3 = sw_open();
  sw4 = sw_open();

  /* Put them in the right place */
  sw_move(sw1, 60.0, 0.0, 10.0);
  sw_move(sw2, 60.0, 300.0, 0.0);
  sw_move(sw3, 60.0, 60.0, 0.0);
  sw_move(sw4, 60.0, 180.0, 10.0);
}

/*
 * Initializes a stream and returns it.
 */
static stream
*initialize_stream(swindow *sw,
                   thread *t,
                   float level_0_gain,
                   float level_1_gain,
                   float level_2_gain,
                   float level_3_gain,
                   float level_4_gain,
                   float level_0_rate,
                   float level_1_rate,
                   float level_2_rate,
                   float level_3_rate,
                   float level_4_rate,
                   int start_delay_ticks)
{
  stream *s;

  /* Open four streams. */
  s = (stream *)calloc((size_t)1, sizeof(stream));

  s->sw = sw;

  /* Set up the gains properly */
  sw_gain(s->sw, level_1_gain);
  s->delayed_gain = level_1_gain;
  s->last_gain = level_1_gain;
  s->gains[0] = level_0_gain;
  s->gains[1] = level_1_gain;
  s->gains[2] = level_2_gain;
```

```
    s->gains[3] = level_3_gain;
    s->gains[4] = level_4_gain;
    s->rates[0] = level_0_rate;
    s->rates[1] = level_1_rate;
    s->rates[2] = level_2_rate;
    s->rates[3] = level_3_rate;
    s->rates[4] = level_4_rate;
    s->paused_pos = -1;
    s->gain_level = 1;
    s->last_gain_level = 1;
    s->gain_delay_ticks = 0;
    s->last_tsms_ratio = tsms_ratio;
    s->tsms_ratio  = tsms_ratio;
    s->tsms_rate   = tsms_rate;
    s->start_delay_ticks = start_delay_ticks;
    s->thread = t;
    s->current_segment = (segment *)NULL;
    s->segment_index = 0;
    s->attention = FALSE;
    s->start_time = (struct timeb *)calloc((size_t)1, sizeof(struct timeb));
    s->focus_time = (time_t)0;
    return s;
  }

  /*
   * Initializes the streamer configuration to default values.
   */
  static void
  init_streamer_config()
  {
    /* Set gains to default values */
    level_0_left_gain = LEVEL_0_LEFT_GAIN;
    level_1_left_gain = LEVEL_1_LEFT_GAIN;
    level_2_left_gain = LEVEL_2_LEFT_GAIN;
    level_3_left_gain = LEVEL_3_LEFT_GAIN;
    level_4_left_gain = LEVEL_4_LEFT_GAIN;

    level_0_right_gain = LEVEL_0_RIGHT_GAIN;
    level_1_right_gain = LEVEL_1_RIGHT_GAIN;
    level_2_right_gain = LEVEL_2_RIGHT_GAIN;
    level_3_right_gain = LEVEL_3_RIGHT_GAIN;
    level_4_right_gain = LEVEL_4_RIGHT_GAIN;

    level_0_middle_gain = LEVEL_0_MIDDLE_GAIN;
    level_1_middle_gain = LEVEL_1_MIDDLE_GAIN;
    level_2_middle_gain = LEVEL_2_MIDDLE_GAIN;
    level_3_middle_gain = LEVEL_3_MIDDLE_GAIN;
    level_4_middle_gain = LEVEL_4_MIDDLE_GAIN;

    /* Set rates to default values */
    level_0_left_rate = LEVEL_0_LEFT_RATE;
    level_1_left_rate = LEVEL_1_LEFT_RATE;
    level_2_left_rate = LEVEL_2_LEFT_RATE;
    level_3_left_rate = LEVEL_3_LEFT_RATE;
    level_4_left_rate = LEVEL_4_LEFT_RATE;

    level_0_right_rate = LEVEL_0_RIGHT_RATE;
    level_1_right_rate = LEVEL_1_RIGHT_RATE;
    level_2_right_rate = LEVEL_2_RIGHT_RATE;
    level_3_right_rate = LEVEL_3_RIGHT_RATE;
    level_4_right_rate = LEVEL_4_RIGHT_RATE;

    level_0_middle_rate = LEVEL_0_MIDDLE_RATE;
    level_1_middle_rate = LEVEL_1_MIDDLE_RATE;
    level_2_middle_rate = LEVEL_2_MIDDLE_RATE;
    level_3_middle_rate = LEVEL_3_MIDDLE_RATE;
  level_4_middle_rate = LEVEL_4_MIDDLE_RATE;

  /* Other */
  tsms_ticks = TSMS_TICKS;
  gain_delay_ticks = GAIN_DELAY_TICKS;
  middle_start_delay = MIDDLE_START_DELAY;
  right_start_delay  = RIGHT_START_DELAY;
  left_start_delay = LEFT_START_DELAY;
  tsms_ratio = TSMS_RATIO;
  max_tsms_ratio = MAX_TSMS_RATIO;
  tsms_rate = TSMS_RATE;
  tone_time_interval = TONE_TIME_INTERVAL;
  focus_time_window = FOCUS_TIME_WINDOW;
}

/*
 * Reads a configuration file.
 */
static void
read_streamer_config_file(char *filename)
{
  char value[256];

  /* This isn't really very efficient */

  /* Set the variable values from the configuration file */

  /* Set gains for left channel */
  if(CaddSearchForKeyValue("level_0_left_gain", filename, value, 256)
     == TRUE)
    level_0_left_gain = atof(value);

  if(CaddSearchForKeyValue("level_1_left_gain", filename, value, 256)
     == TRUE)
    level_1_left_gain = atof(value);

  if(CaddSearchForKeyValue("level_2_left_gain", filename, value, 256)
     == TRUE)
    level_2_left_gain = atof(value);

  if(CaddSearchForKeyValue("level_3_left_gain", filename, value, 256)
     == TRUE)
    level_3_left_gain = atof(value);

  if(CaddSearchForKeyValue("level_4_left_gain", filename, value, 256)
     == TRUE)
    level_4_left_gain = atof(value);

  /* Set gains for middle channel */
  if(CaddSearchForKeyValue("level_0_middle_gain", filename, value, 256)
     == TRUE)
    level_0_middle_gain = atof(value);

  if(CaddSearchForKeyValue("level_1_middle_gain", filename, value, 256)
     == TRUE)
    level_1_middle_gain = atof(value);

  if(CaddSearchForKeyValue("level_2_middle_gain", filename, value, 256)
     == TRUE)
    level_2_middle_gain = atof(value);

  if(CaddSearchForKeyValue("level_3_middle_gain", filename, value, 256)
     == TRUE)
    level_3_middle_gain = atof(value);

  if(CaddSearchForKeyValue("level_4_middle_gain", filename, value, 256)
      == TRUE)
     level_4_middle_gain = atof(value);

   /* Set gains for right channel */
   if(CaddSearchForKeyValue("level_0_right_gain", filename, value, 256)
      == TRUE)
     level_0_right_gain = atof(value);

   if(CaddSearchForKeyValue("level_1_right_gain", filename, value, 256)
      == TRUE)
     level_1_right_gain = atof(value);

   if(CaddSearchForKeyValue("level_2_right_gain", filename, value, 256)
      == TRUE)
     level_2_right_gain = atof(value);

   if(CaddSearchForKeyValue("level_3_right_gain", filename, value, 256)
      == TRUE)
     level_3_right_gain = atof(value);

   if(CaddSearchForKeyValue("level_4_right_gain", filename, value, 256)
      == TRUE)
     level_4_right_gain = atof(value);


   /* Set rates for left channel */
   if(CaddSearchForKeyValue("level_0_left_rate", filename, value, 256)
      == TRUE)
     level_0_left_rate = atof(value);

   if(CaddSearchForKeyValue("level_1_left_rate", filename, value, 256)
      == TRUE)
     level_1_left_rate = atof(value);

   if(CaddSearchForKeyValue("level_2_left_rate", filename, value, 256)
      == TRUE)
     level_2_left_rate = atof(value);

   if(CaddSearchForKeyValue("level_3_left_rate", filename, value, 256)
      == TRUE)
     level_3_left_rate = atof(value);

   if(CaddSearchForKeyValue("level_4_left_rate", filename, value, 256)
      == TRUE)
     level_4_left_rate = atof(value);

   /* Set rates for middle channel */
   if(CaddSearchForKeyValue("level_0_middle_rate", filename, value, 256)
      == TRUE)
     level_0_middle_rate = atof(value);

   if(CaddSearchForKeyValue("level_1_middle_rate", filename, value, 256)
      == TRUE)
     level_1_middle_rate = atof(value);

   if(CaddSearchForKeyValue("level_2_middle_rate", filename, value, 256)
      == TRUE)
     level_2_middle_rate = atof(value);

   if(CaddSearchForKeyValue("level_3_middle_rate", filename, value, 256)
      == TRUE)
     level_3_middle_rate = atof(value);

   if(CaddSearchForKeyValue("level_4_middle_rate", filename, value, 256)
      == TRUE)
     level_4_middle_rate = atof(value);
```

```
    /* Set rates for right channel */
    if(CaddSearchForKeyValue("level_0_right_rate", filename, value, 256)
       == TRUE)
      level_0_right_rate = atof(value);

    if(CaddSearchForKeyValue("level_1_right_rate", filename, value, 256)
       == TRUE)
      level_1_right_rate = atof(value);

    if(CaddSearchForKeyValue("level_2_right_rate", filename, value, 256)
       == TRUE)
      level_2_right_rate = atof(value);

    if(CaddSearchForKeyValue("level_3_right_rate", filename, value, 256)
       == TRUE)
      level_3_right_rate = atof(value);

    if(CaddSearchForKeyValue("middle_start_delay", filename, value, 256)
       == TRUE)
      middle_start_delay = atoi(value);

    if(CaddSearchForKeyValue("right_start_delay", filename, value, 256)
       == TRUE)
      right_start_delay = atoi(value);

    if(CaddSearchForKeyValue("left_start_delay", filename, value, 256)
       == TRUE)
      left_start_delay = atoi(value);

    if(CaddSearchForKeyValue("tone_time_interval", filename, value, 256)
       == TRUE)
      tone_time_interval = atoi(value);

    if(CaddSearchForKeyValue("focus_time_window", filename, value, 256)
       == TRUE)
      focus_time_window = atoi(value);

    if(CaddSearchForKeyValue("tsms_ticks", filename, value, 256)
       == TRUE)
      tsms_ticks = atoi(value);

    if(CaddSearchForKeyValue("gain_delay_ticks", filename, value, 256)
       == TRUE)
      gain_delay_ticks = atoi(value);

    if(CaddSearchForKeyValue("tsms_ratio", filename, value, 256)
       == TRUE)
      tsms_ratio = atof(value);

    if(CaddSearchForKeyValue("max_tsms_ratio", filename, value, 256)
       == TRUE)
      max_tsms_ratio = atof(value);

    if(CaddSearchForKeyValue("tsms_rate", filename, value, 256)
       == TRUE)
      tsms_rate = atof(value);
  }

  /*
   * Initializes four streams. Three have data and the fourth will have
   * a silent buffer looped though it.
   */
  void
  s_initialize(char *filename1,
               char *filename2,
              char *filename3,
              char *config_filename,
              int output,
              int thead,
              int tgaze,
              int tfish,
              int fb)
{
  int i;


  /* Read in the default configuration */
  init_streamer_config();

  /* What about output ? */
  if(output)
    output_fd = output;
  else
    output_fd = -1;

  /* Initialize the polhemus if required */
  if(thead || tgaze)
    polhemus_rw = init_polhemus_rw("/dev/ttya");
  else
    polhemus_rw = -1;

  if(tfish)
    fish_rw = init_fish_rw("/dev/ttya");
  else
    fish_rw = -1;

  if(thead)
    track_head = TRUE;
  else
    track_head = FALSE;

  if(tgaze)
    track_gaze = TRUE;
  else
    track_gaze = FALSE;

  if(fb)
    feedback = TRUE;
  else
    feedback = FALSE;

  /* Set up the erase string */
  for(i = 0; i < DISPLAY_LENGTH + 1; i++)
    erase_string[i] = '\010';

  /* Read in the configuration file if there is one */
  if(config_filename != (char *)NULL)
    read_streamer_config_file(config_filename);

  /* Initialize the swindows */
  initialize_swindows();

  /* Initialize the segments */
  t1 = initialize_thread(filename1);
  t2 = initialize_thread(filename2);
  t3 = initialize_thread(filename3);


  /* Set focus to NULL */
  focus = (stream *)NULL;
```

```
/* THIS IS A HACK, because the sound servers are writing to pipes */
for(i=0; i < SILENCE_BUFFER_LENGTH; i++)
  silence[i] = '\377';
silence_bb.maxLen = SILENCE_BUFFER_LENGTH;
silence_bb.currLen = SILENCE_BUFFER_LENGTH;
silence_bb.data = silence;

/* Open four streams */
s1 = initialize_stream(sw1, t1,
                       level_0_middle_gain,
                       level_1_middle_gain,
                       level_2_middle_gain,
                       level_3_middle_gain,
                       level_4_middle_gain,
                       level_0_middle_rate,
                       level_1_middle_rate,
                       level_2_middle_rate,
                       level_3_middle_rate,
                       level_4_middle_rate,
                       middle_start_delay);
center = s1;
s2 = initialize_stream(sw2, t2,
                       level_0_right_gain,
                       level_1_right_gain,
                       level_2_right_gain,
                       level_3_right_gain,
                       level_4_right_gain,
                       level_0_right_rate,
                       level_1_right_rate,
                       level_2_right_rate,
                       level_3_right_rate,
                       level_4_right_rate,
                       right_start_delay);
right = s2;
s3 = initialize_stream(sw3, t3,
                       level_0_left_gain,
                       level_1_left_gain,
                       level_2_left_gain,
                       level_3_left_gain,
                       level_4_left_gain,
                       level_0_left_rate,
                       level_1_left_rate,
                       level_2_left_rate,
                       level_3_left_rate,
                       level_4_left_rate,
                       left_start_delay);
left = s3;
s4 = initialize_stream(sw4, (thread *)NULL,
                       OFF_GAIN,
                       MIDI_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       OFF_GAIN,
                       0);

/* Set up the gaze variables */
gaze = (stream *)NULL;
last_gaze = (stream *)NULL;

/* Play the first page */
play_empty_buf_on_stream(s4);
```

```
  /* Always continue playing to the end of the segment */
  sw_register_callback(S_PLAY_DONE_EV, segment_cb, (char *)NULL);

  /* Loop and empty buffer into s4 */
  sw_register_callback(S_PLAY_BUF_EV, buffer_cb, (char *)NULL);

  /* Periodically wake up and decay gains, etc */
  SmSetTimeoutCallBack(TICK_SIZE, time_cb, (caddr_t *)NULL);
}

/*
 * Read in the segments and add them to a thread.
 */
static thread
*initialize_thread(char *filename)
{

  thread *t;
  VARRAY *segments;
  int num_segments;

  /* Allocate a thread */
  t = (thread *)calloc((size_t)1, sizeof(thread));

  /* Initialize the segments from a file */
  segments = initialize_segments(filename, &num_segments);

  t->segments = segments;
  t->num_segments = num_segments;
  t->current_segment_index = 0;
  return t;
}

/*
 * Delays the start of a stream by start_delay_ticks.
 */
static int
stagger_start(stream *s)
{
  /* Stagger the stream start */
  if(s->start_delay_ticks > 0)
    {
      s->start_delay_ticks -= 1;
      return FALSE;
    }
  else if(s->start_delay_ticks == 0)
    {
      s->start_delay_ticks = -1;
      s->attention = TRUE;
      play_tone(s, ATTENTION_TONE, FALSE);
      return TRUE;
    }
  else
    return TRUE;
}

/*
 * Try to estimate the current position of the sound. Since we're
 * writing to a pipe we need to use a timer to get the position. This is
 * inherently inaccurate.
 */
static long
current_pos(stream *s)
{
  struct timeb current_time;
```

```
    long current_pos;

    /* Figure out how long this segment has been playing */
    ftime(&current_time);
    current_pos = (current_time.time * 1000) +
      current_time.millitm;
    current_pos = current_pos -
      ((s->start_time->time * 1000) +
       s->start_time->millitm);
    current_pos = current_pos + s->start_pos;
    return current_pos;

  }

  /*
   * Paused a stream at the current position
   */
  static void
  pause_stream(stream *s)
  {
    if(s->paused_pos < 0)
      {
        /* Pauses a stream at the current position */
        /* Use continue_stream to start it again */
        s->paused_pos = current_pos(s);

        /* Halt the stream */
        sw_halt_and_flush_queue(s->sw);

        s->gain_level = 0;

        play_empty_buf_on_stream(s);
      }
  }

  /*
   * Restarts a stream at the current position.
   */
  static void
  continue_stream(stream *s)
  {
    /* Figure out where to restart it */
    if(s->paused_pos > 0)
      {
        long pos;

        /* Stop looping empty stuff */
        sw_halt_and_flush_queue(s->sw);

        pos = s->paused_pos;

        /* Be sure to set this here ! */
        s->paused_pos = -1;

        /* Play the segment */
        if(s->current_segment != (segment *)NULL)
          play_segment(s, s->current_segment, pos, s->stop_pos);
      }
  }

  void
  play_level(int level,
             stream *s)
  {
    float r, theta, phi;
    if(feedback)
      {
        if(s != (stream *)NULL)
          {
            r = s->sw->slocation[0];
            theta = s->sw->slocation[1];
            phi = s->sw->slocation[2];
          }
        else
          {
            r = 0.0;
            theta = 0.0;
            phi = 0.0;
          }

        /* Move the feedback window */
        sw_move(s4->sw, r, theta, phi);
        while(cre_update_audio() < 0);

        /* Play a short note */
        cre_msg_midi(s4->sw->b_id, MIDI_PATCH, CHNL, INSTR, 0);
        cre_msg_midi(s4->sw->b_id, MIDI_NOTE_ON, CHNL,
                     ((OCTAVE_DN1 + level)*12)+ NOTE_C, VELOCITY);
        cre_msg_midi(s4->sw->b_id, MIDI_NOTE_OFF, CHNL,
                     ((OCTAVE_DN1 + level)*12)+ NOTE_C, VELOCITY);
        while(cre_update_audio() < 0);
      }
  }

  /*
   * Gives focus to the stream.
   */
  void
  s_give_focus(stream *s)
  {
    time_t current_time;

    /* If this stream isn't in focus then unfocus first */
    if(s != focus)
      {
        /* This is a lot like s_unfocus, except that it has slightly */
        /* different behavior. Here we don't reset the gain_level to */
        /* zero or reset the focus_time */
        if(focus != (stream *)NULL)
          {
            /* Restart any paused streams */
            if(s1 != focus)
              continue_stream(s1);
            if(s2 != focus)
              continue_stream(s2);
            if(s3 != focus)
              continue_stream(s3);

            /* Reset the gain and speed on the focus */
            focus->last_gain_level = focus->gain_level;
            focus->last_gain = sw_get_gain(focus->sw);
            focus->gain_level = 0;
            if(sw_get_gain(focus->sw) != OFF_GAIN)
              sw_gain(focus->sw, focus->gains[0]);

            /* Save the tsms_ratio */
            focus->last_tsms_ratio = focus->tsms_ratio;

            /* Reset the focus unless it's at the default */
            if(focus->tsms_ratio > tsms_ratio)
              {
```

```
              focus->tsms_ratio = tsms_ratio;
              s_tsms_ratio(focus->sw->id, focus->tsms_ratio);
            }

          /* Save the time that this stream was last in focus */
          time(&current_time);
          focus->focus_time = current_time;

          /* Set the focus to Null */
          focus = (stream *)NULL;
        }

      /* Bug in the Beachtron */
      usleep(100000);
    }

  /* Get the current time */
  time(&current_time);

  /* See if where within the focus window */
  if((s != focus) &&
     ((current_time - s->focus_time) < FOCUS_TIME_WINDOW))
    {
      /* Reset gain and tsms_ratio */
      s->tsms_ratio = s->last_tsms_ratio;
      s_tsms_ratio(s->sw->id, s->tsms_ratio);
      s->gain_level = s->last_gain_level;

      if(s->gain_level == 0)
        {
          s->gain_level++;
          sw_gain(s->sw, s->gains[1]);
        }
      else
        sw_gain(s->sw, s->last_gain);

      play_level(s->gain_level, s);

      /* If we're resetting to level 4 we need to pause the other */
      /* streams again. */
      if(s->gain_level == 4) {
        if(s1 != s)
          pause_stream(s1);
        if(s2 != s)
          pause_stream(s2);
        if(s3 != s)
          pause_stream(s3);
      }
    }
  else if(s->gain_level == 0)
    {
      s->gain_level++;
      play_level(s->gain_level, s);
      sw_gain(s->sw, s->gains[1]);
    }
  else if(s->gain_level == 1)
    {
      s->gain_level++;
      play_level(s->gain_level, s);
      sw_gain(s->sw, s->gains[2]);
    }
  else if (s->gain_level == 2)
    {
      s->gain_level++;
      play_level(s->gain_level, s);
      sw_gain(s->sw, s->gains[3]);
    }
  else if (s->gain_level == 3)
    {
      /* Okay we're going to level 4, */
      /* need to pause the other streams */
      if(s1 != s)
        pause_stream(s1);
      if(s2 != s)
        pause_stream(s2);
      if(s3 != s)
        pause_stream(s3);

      /* Reset the gain on the primary_stream */
      s->gain_level++;
      play_level(s->gain_level, s);
      sw_gain(s->sw, s->gains[4]);
    }

  /* Save the focus and time */
  focus = s;
}

/*
 * Turn off the focus. That is, reset the gain_level and the speed.
 */
void
s_unfocus()
{

  if(focus != (stream *)NULL)
    {
      /* Restart any paused streams */
      if(s1 != focus)
        continue_stream(s1);
      if(s2 != focus)
        continue_stream(s2);
      if(s3 != focus)
        continue_stream(s3);

      /* Reset the gain and speed on the focus */
      focus->last_gain_level = 0;
      focus->gain_level = 0;
      focus->last_gain = focus->gains[0];
      play_level(focus->gain_level, focus);
      if(sw_get_gain(focus->sw) != OFF_GAIN)
        sw_gain(focus->sw, focus->gains[0]);

      /* Also reset the focus time */
      focus->focus_time = (time_t)0;

      /* Reset the tsms_ratio to the default */
      focus->last_tsms_ratio = tsms_ratio;

      /* Reset the focus unless it's at the default */
      if(focus->tsms_ratio > tsms_ratio)
        {
          focus->tsms_ratio = tsms_ratio;
          s_tsms_ratio(focus->sw->id, focus->tsms_ratio);
        }
      focus = (stream *)NULL;
    }
}

/*
 * Bumps the segment counter on the stream that's the current focus.
 * THIS FUNCTION MIGHT BE STALE AT THE MOMENT.
 */
void
s_next_segment(int direction)
{
  if(focus != (stream *)NULL)
    {
      if(((focus->thread->current_segment_index + direction) <
          focus->thread->num_segments) &&
         ((focus->thread->current_segment_index + direction) >= 0))
        {
          sw_halt_and_flush_queue(focus->sw);
          focus->thread->current_segment_index += direction;
          focus->attention = TRUE;
          play_tone(focus, FOCUS_NEW_PAGE_TONE, FALSE);
        }
      else
        {
          /* Pause at the current position */
          focus->paused_pos = current_pos(focus);

          /* Halt the sound and play no segment tone */
          sw_halt_and_flush_queue(focus->sw);
          focus->attention = TRUE;
          play_tone(focus, NO_SEGMENT_TONE, FALSE);
        }
    }
}

void
s_rotate()
{
  float r, theta, phi;
  stream *temp;

  /* Rotate the positions of the streams */
  temp = center;
  r = center->sw->slocation[0];
  theta = center->sw->slocation[1];
  phi = center->sw->slocation[2];

  sw_move(center->sw,
          right->sw->slocation[0],
          right->sw->slocation[1],
          right->sw->slocation[2]);

  sw_move(right->sw,
          left->sw->slocation[0],
          left->sw->slocation[1],
          left->sw->slocation[2]);

  sw_move(left->sw, r, theta, phi);

  center = left;
  left = right;
  right = temp;
}


/*
 * Initializes a set of segments by reading in the DBs in filename.
 * Return value is the number of segments. Side effects global
 * variable, segments. In this case segments correspond to story
 * boundaries as derived from closed caption information.
 */
static VARRAY
*initialize_segments(char *filename,
                         int *num_segments_ptr)
{
  segment *temp;
  VARRAY *segments;
  int num_segments;
  FILE *infile;
  char line[MAX_LINE_LEN];
  char audio_filename[MAX_LINE_LEN];
  long start, stop;
  int length;
  char c;

  if((infile = fopen(filename, "r")) == (FILE *)NULL)
    {
      perror("Can't open segments file");
      exit(-1);
    }

  segments = (VARRAY *)calloc((size_t)1, sizeof(VARRAY));
  VarrayInit(segments, NUM_SEGMENTS);

  num_segments = 0;

  /* Convert it to an array of segments */
  while(!feof(infile))
    {
      if((c = getc(infile)) == '<')
        {
          /* It's an audio file, get the rest of the line */
          fgets(line, MAX_LINE_LEN, infile);

          /* How long is it ? */
          length = strlen(line);

          /* Put in a null */
          line[length - 2] = '\0';

          strncpy(audio_filename, line , length);
        }
      else if (c == '#')
        /* It's a comment. Toss the line */
        fgets(line, MAX_LINE_LEN, infile);
      else
        {
          /* It's a range, so put the character back on the stream */
          ungetc(c, infile);

          /* Get the start and stop */
          fscanf(infile, "%ld %ld\n", &start, &stop);

          /* Allocate segment structure */
          temp = (segment *)calloc((size_t)1, sizeof(segment));

          /* Fill in the values */
          strcpy(temp->audio_file, audio_filename);
          temp->start = start;
          temp->stop = stop;
          temp->duration = temp->stop - temp->start;

          /* Save the segment in array */
          VarraySet(segments, num_segments, temp);

          num_segments++;
        }
    }
  /* Close the file */
  fclose(infile);

  /* Set up return values */
  *num_segments_ptr = num_segments;
  return segments;
}

/*
 * Frees the storage allocated to a thread.
 */
static void
free_thread(thread *t)
{
  int i;
  segment *s;

  /* Free all of the segments */
  for(i = 0; i < t->num_segments; i++)
    {
      s = (segment *)VarrayGet(t->segments, i);
      free(s);
    }

  /* Free the segment varray */
  VarrayFree(t->segments);

  /* Free the thread */
  free(t);
}

/*
 * Free the storage allocated to a stream.
 */
static void
free_stream(stream *s)
{
  /* Free the timeb struct */
  free(s->start_time);

  /* Free the stream itself */
  free(s);
}

/*
 * Kills the streams and free their storage.
 */
void
s_kill()
{

  /* Free threads */
  free_thread(t1);
  free_thread(t2);
  free_thread(t3);

  /* Free streams */
  free_stream(s1);
  free_stream(s2);
  free_stream(s3);
  free_stream(s4);

  /* Free swindows */
  sw_kill();
}
```