A Digital Audio Player for the HyperWave Internet Server

Günter Geiger

A Digital Audio Player for the HyperWave Internet Server

Master's Thesis

at

Graz University of Technology

submitted by

Günter Geiger

Institute for Information Processing and Computer Supported New Media (IICM),
Graz University of Technology
A-8010 Graz, Austria

January 1997

Advisor:	o.Univ.-Prof. Dr. Dr.h.c. Hermann Maurer
Supervisor:	Univ.Ass. Dr. Keith Andrews

Ein digitaler Audio Player für den HyperWave Internet Server

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Günter Geiger

Institut für Informationsverarbeitung und Computergestützte neue Medien (IICM),
Technische Universität Graz
A-8010 Graz

Januar 1997

Diese Diplomarbeit ist in englischer Sprache verfaßt.

Begutachter:	o.Univ.-Prof. Dr. Dr.h.c. Hermann Maurer
Betreuer:	Univ.Ass. Dr. Keith Andrews

Für meine Eltern

Abstract

This thesis describes the Harmony Audio Player, an application for playback and link editing of digital audio documents managed by the HyperWave Internet server and Web document management system. Harmony is the Unix client and authoring tool for HyperWave.

The Harmony Audio Player supports numerous diverse digital audio formats and was carefully designed to achieve maximum portability between different Unix platforms, which traditionally provide incompatible audio output facilities. A graphical waveform tool with moving cursor reflects the current playback position during audio output.

Local audio documents can be uploaded onto the HyperWave server, in a manner similar to other media types in Harmony. Particularly interesting are the Harmony Audio Player's facilities for editing and following hyperlinks in audio documents – both source and destination anchor regions can be specified interactively. A stand-alone version of the Audio Player is available for general-purpose multi-platform, multi-format audio playback, such as a helper application for a Web browser.

Kurzfassung

Diese Diplomarbeit beschreibt den Harmony Audio Player. Der Harmony Audio Player ist ein Programm mit dessen Hilfe Audio Dokumente, die auf einem HyperWave Server liegen abgespielt werden können. Harmony ist eine Unix/X11 Applikation, mit deren Hilfe Daten von einem HyperWave Server betrachtet und editiert werden können.

Der Harmony Audio Player unterstützt mehrere verschiedene digitale Audioformate und wurde konzipiert um auf mehrere Unix Plattformen portierbar zu sein, die traditionellerweise inkompatible Audio Schnittstellen besitzen. Eine grafische Darstellung der Wellenform mit sich bewegendem Cursor zeigt die aktuelle Abspielposition des Players an.

Auf ähnliche Weise wie bei anderen Dokumenttypen können lokale Audiodokumente in das HyperWave System eingebracht werden. Zusätzlich können Hyperlinks in Audiodokumenten editiert und verfolgt werden. Dabei ist es möglich sowohl “Source Anchors” wie auch “Destination Anchors” zu definieren. Eine weitere Version des Audio Players ermöglicht es das Programm auch ohne Harmony zu betreiben und als mehrere Audio Formate unterstützenden Helfer für einen Web Browser zu verwenden.

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Acknowledgments

I would like to thank my supervisor Keith Andrews for his help and advice in writing and correcting this work. Bernhard Marschall, Michael Pichler and Jürgen Schipflinger for their help in mastering the difficulties of programming. All the people at the IICM and especially the HyperWave team who gave me the possibility for working in an ambitious team on a great idea.

Credits

- Figure 2.1 is taken from Keith Andrews work [And96] and used with permission.
- Figure 7.1 is taken from [And96].

Contents

Chapter 1

Introduction

The Harmony Audio Player is a Audio Player which offers the possibility to insert and follow hyperlinks in audio data. This thesis describes the Harmony Audio Player and related topics. Harmony is the Unix/X11 client for the HyperWave system.

Chapter 2 presents a history of the Internet starting with the basic concepts and Internet protocols. A description of different Internet services like Telnet, FTP, Electronic Mail and Network News follows. The section on Internet Information Systems describes Archie, Gopher and WAIS. The last section of Chapter 2 deals with hypermedia systems, starting with a general description of hypertext and hypermedia and the World Wide Web and finally presenting an overview of HyperWave and it's concepts.

The third chapter describes the digital representation of audio. The amount of memory used for storing audio data, and why audio data is difficult to handle in a networked context.

Chapter 4 describes different techniques for reducing the data size of audio data and methods of storing and transmitting digital audio data. Many different audio file formats exist and an Internet application should be able to handle most of them. The second part of Chapter 3 describes the important ones in detail, and mentions the audio file formats that are not supported (until now) by the Harmony Audio Player.

In order to support different Unix platforms it is necessary to use a common system for audio output. The first section of Chapter 5 discusses audio programming on Unix platforms in general. The following sections point out the advantages and disadvantages of different systems for audio programming.

Chapter 6 describes hyperlinks in audio data. It focuses on the representation of links in audio and the possibilities for editing hyperlinks in audio. The first part of the chapter looks at different audio players and editing tools. The second part introduces hyperlinks in audio, how they can be represented and what possibilities they offer. A new concept for time scheduled multimedia presentation triggered by audio is described.

Chapter 7 talks about the implementation of the Harmony Audio Player. It describes the concept of Harmony and its viewers. The most important C++ classes of the Harmony Audio Player are presented.

The last chapter is an outlook on different kind of things which are not yet implemented in the Harmony Audio Player, such as a user interface for a link hierarchy and the possibility of audible link representation. The implementation of different audio file formats and live audio streams for Internet radio and Internet phone would also be a feature to be implemented in the Harmony Audio Player.

Appendix A is a user guide to the Harmony Audio Player. It describes how the user can interact with the Harmony Audio Player.

Chapter 2

The Internet and Hypermedia

2.1 History

The Internet is a computer network, combining thousands of different networks, to form one huge entity. Some say its birthday is 20 years ago [Kro92], when the U.S. Defense Department started the Arpanet, a computer network where each computer was connected to others over several different routes, in order to withstand damage in a nuclear war. The design goal was, that the computers cannot rely on the functionality of the net itself.

Others say that the Internet started in the early eighties, as an evolution from the Arpanet. The design of the Arpanet was used for building a network of educational institutions, the NSFnet, which provided connectivity between the universities in the U.S. It is interesting, that the design of the Arpanet to withstand damage, was one of the main reasons for the growth of the Internet. Since the net still worked, when some connections were out of order, it was easy to improve network connectivity gradually. Most users would not recognize, that something has changed, except maybe that their connectivity improved slightly.

Ever more new computers were connected to the Internet, causing the Internet to become slower, causing the connections to improve, causing new computers to be connected, ...

2.1.1 The Internet Protocol (IP) and Transmission Control Protocol (TCP)

Since computers on the Internet are not connected directly to each other, there has to be some mechanism for data to get from one point to another. This is done with routers, special computers which look at the receiver the data is addressed to, and send the data on its way in a specific direction. Typically routers are connected to other computers or routers and can decide to whom they send the data. The job of addressing the data is done by the Internet Protocol (IP). It puts the data together with the sender's and the receiver's address.

One could address ones data with IP and send it to the next router, which would look at the address and forward the data in the right direction. But sometimes the data to be sent is very large - this causes problems:

- The data line is blocked while transmitting the data.
- If something goes wrong during transmission, the whole process has to be restarted.

Therefore the data has to be divided into smaller chunks before sending. This is done according to the Transmission Control Protocol (TCP). The data is divided into small data chunks and each chunk is given a number. When arriving at the receiver, the chunks are collected, and put together again in the correct sequence. The problems which are handled by TCP are

- Putting the packets together in the correct order.
- Ensuring that all packets arrive at the receiver through re-sending of lost packets.

A second protocol which handles divided IP packets is the User Datagram Protocol (UDP). This protocol is much faster than TCP, because it lacks the control mechanism for ensuring that all the data sent will arrive at the receiver. UDP is used for streaming audio protocols such as Real Audio mentioned in Section 4.1.7.

2.1.2 Internet Addresses and Domain Name Scheme

Internet addresses are 32 bit numbers. There would be nearly enough numbers to assign an Internet address to every human, but still the growth of the Internet some kind of lack of different Internet addresses arose. The 32 bit address is divided into 4 8-bit numbers, each ranging from 0 to 255. Normally an Internet address is written as four numbers (e.g. 193.170.129.55). There are three classes of Internet addresses: Class A, class B, and class C. A class A address forms a group which includes all addresses which have the same number at the most significant part of the Internet address (e.g 193). A class B address includes all numbers beginning with for example 193.170 and class C includes all numbers beginning with 193.170.129. Because there are only 256 class C addresses in each group most of the organizations tried to receive class B addresses (which are 65534 addresses). This lead to a shortage of class B addresses.

In order to be easier remembered by human persons, each of these Internet addresses has their own name. To make the administration for the large number of computers connected to the Internet easier the domain name system was developed. The whole address space was divided into different domains with appropriate names (com, edu, gov, mil, org, net and different country specific domain names). The authority for subdividing their specific domain into sub-domains was given away to the maintainers of a specific domain. They again divided their domain into different domains and delegated authority for further naming. This leads to a hierarchical naming scheme, where each hierarchy is responsible for the uniqueness of its names. With this system no two computers can have the same name.

2.2 Internet Services

In order to use the Internet different applications were developed. This section describes the first suite of applications which were available to communicate on the Net. Every computer on a TCP/IP network provides different services. In order to distinguish which service a remote computer wants to access, each connection is made giving a specific port number. According to this port number the computer decides which program to start. Or the other way round, so-called daemons, programs which run all the time, just listen to the connection requests which come from the net and respond, if the request is on their specific port. Different services have so-called well known ports. This means that for example when connecting to a computer using port number 21, the FTP service will answer.

2.2.1 Telnet

Telnet is the most basic tool for connecting to different computers. With the telnet command it is possible to connect to a specific port on a specific computer. After the connection was successful, everything that is sent by the remote computer is displayed on the terminal and everything that is typed in is sent to the remote computer.

Most of the time, telnet is used to connect to another computer using the remote computers telnet service. With the telnet service the user can login on the remote computer and use its computing resources.

2.2.2 FTP

The File Transfer Protocol (FTP) was designed in order to send and receive data (files) over the Internet. The FTP application allows the user to navigate in the remote computers file system and to send and receive files to and from the computer. This is a frequently used application, because many computers on the Internet provide anonymous FTP access, which means users connected to the Internet can login via FTP on a remote computer, navigate through a specific part of its file system, and download the files of interest.

2.2.3 Electronic Mail

The electronic mail system is one of the most frequently used tools on the Internet. Most of the time it is used to send text messages to other users, but it can also be used to send different kinds of multimedia data over the net (The MIME Extensions [Moo93]).

On the Internet there are different mail servers, which store mail messages and forward them. So the electronic mail “hops” from one computer to another until it reaches its destination computer, where it is put into a user specific mail file. From this file the user can fetch his mail with a mail reader and read it. Messages which cannot be delivered are sent back to the sender, with some kind of description why the delivery was not possible.

2.2.4 Network News

The Network News system is a collection of documents, similar to email messages, which are part of different newsgroups. These newsgroups have a hierarchical naming scheme, which divides them into categories. The main News categories are comp, nes, rec, sci, soc, talk, misc, and alt. These categories then are further divided into different sub-categories.

Basically, the News system consists of News servers which carry the messages of the newsgroups. Data is added to a newsgroup by different users and then transmitted over the Internet to the other news servers. Most of the servers just store news for a distinct amount of time. After this amount is expired the particular News messages are deleted.

2.3 Internet Information Systems

With the immense growth of the Internet and its use as an information source, it became necessary to provide some tools for searching and finding information. Information systems not only provide a means of accessing data, but additionally provide a structuring scheme in order to find the data.

2.3.1 Archie

Via FTP it is possible to access a large amount of data over the Internet, but how should the user know, where the data can be found. One way to do this is to use an Archie server. An Archie server is just an index of different anonymous FTP servers. The index is created by a program which logs into a large amount of known FTP servers and receives their directory structure. The directory structure is indexed and sent to all Archie servers. If contacted the Archie server then allows the user to search in this index and sends back the server and location were the specific matching files are located. It is possible to search for filenames, but most of the time, file names do not give good information about what's in the file. Archie additionally provides the “whatis”-index, an index of short descriptions of the stored files.

2.3.2 Gopher and Veronica

Gopher was developed at the University of Minnesota and is a menu driven information hierarchy. With Gopher it is possible to navigate in an hierarchical structure and find resources on the Net by selecting different menu entries, which stand for a specific topic. Each topic then can have further menu entries or can provide access to a particular resource. Gopher servers are interconnected, so when choosing a particular topic it is possible to change to another Gopher server. Each of the servers is structured in some way. How a particular server is structured is up to the maintainer of the server. Gopher provides a transparent interface to the user, which means that it does not make any difference with which Internet service the resource can be accessed. If the resource is chosen, Gopher initiates the correct client to get the resource. Browsing with Gopher is somehow similar to searching in a library where the information is stored after a specific scheme. The scheme is up to the Gopher server administrator. The disadvantage of this is that the information is not structured uniformly, on the other hand the overhead for a common structuring scheme and its maintenance is not necessary. Each information provider is able to organize the information in the way best suited for a specific purpose.

Gopher presented a new feeling for users when accessing the Internet. It is the first Internet Information System which offers besides the information itself a structuring scheme to put the information in order.

Veronica provides an index of menu entries to different Gopher servers. With this system the whole Gopher space can be searched for specific keywords.

2.3.3 Wide Area Information Servers

Wide Area Information Servers (WAIS) provide access to different indices of information. One can think of WAIS servers as being data bases that have a full text index and allow clients to access this index and look for particular topics. A typical WAIS search comes up with a set of documents containing specific indices. The documents are ranked by the number of times a specific word or phrase appears in them. WAIS software is also used for indexing WWW and Gopher servers.

2.4 Hypermedia Systems

2.4.1 Hypertext and Hypermedia

With hyperlinked data organization information is stored in a similar manner as in the human brain. The human thoughts are not linear. When reading a document sometimes a specific topic catches ones attention. Hyperlinks provide the possibility to follow this thoughts and access information to related topics. The reader

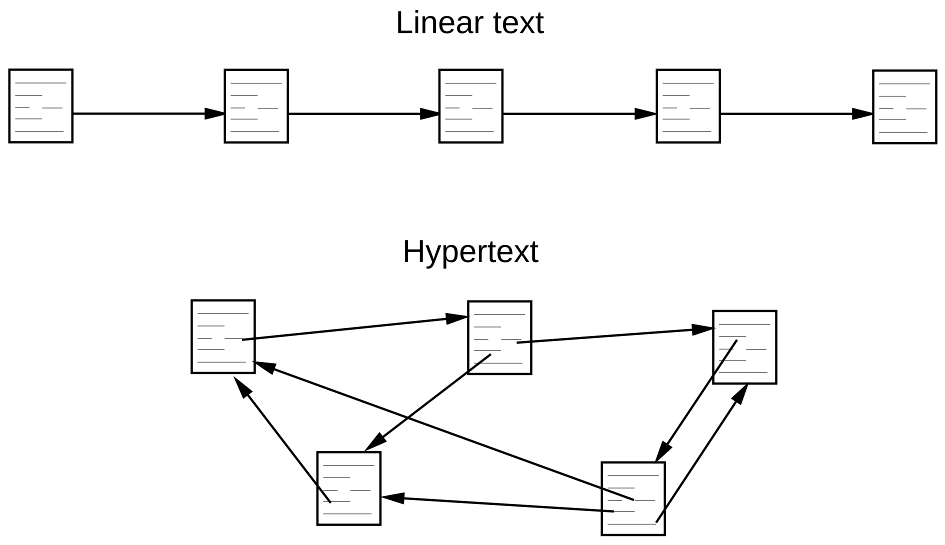


Figure 2.1: Linear text and hypertext.

is able to decide if the thought should be followed or not. Therefore most of the modern online document browsers provide hyperlink support to a certain degree.

Figure2.1 shows how the information is stored in a hyperlinked environment in comparison to the linear structure of a conventional document [And96].

The next step was to provide this possibility (following a link to a more detailed description of a topic) for other kinds of information than text. Hypermedia documents are multimedia documents with hyperlinks.

2.4.2 The World Wide Web

The World Wide Web (WWW) became so popular in the past few years, that "The Web" is already used as a synonym for the Internet. Basically, the concept of the Web consists of so-called hypertext documents. These hypertext documents are written in a language called the Hypertext Markup Language (HTML). Hypertext documents are documents which may contain links to other documents anywhere on the Net.

These links consist of addresses where the referenced document is to be found. The documents can even be on a different host. The addresses are in a specific format, called the Uniform Resource Locator (URL). Not only other hypertext documents can be accessed through the URL, but also different services such as FTP.

Additionally what made the WWW such a success was the availability of WWW browsers, which not only made it possible to access the WWW but also can be used for FTP and other Internet services as Electronic Mail, Network News, etc. The browsers are able to display various kinds of multimedia formats. The WWW is a tool with which one can hop from one interesting topic to another . . . but what about finding a specific topic?

Since hyperlinks, hyperlinks are links in hypertext documents, can point to any document they want, navigation in the World Wide Web can be cumbersome. The WWW does not provide a means of hierarchical structure as Gopher does. It does not provide a means for indexing as in WAIS. Since the structure of the WWW is built up mostly by individuals it is not controllable any more. When a hypertext document is written, it is the responsibility of the author to check if the links in the document are up to date. It is possible that documents disappear, and all the links pointing to these documents are turned into so-called dangling links.

WWW Search Engines

In order to provide a tool for searching in the WWW so-called WWW Robots were developed. A WWW robot is a program, that accesses hypertext documents and follows their links to different documents. It then makes an index about all hypertext files it accessed. This index then can be searched by the user. This method of providing a index for searching causes a lot of problems. Not only is it very expensive in terms of network and computing resources, but also is it very hard to keep the index consistent. WWW robots do not have any other means of removing dangling links than periodically checking the referenced documents.

2.4.3 HyperWave

HyperWave is an information system which was developed at the Graz University of Technology by a group around Hermann Maurer [hyperw]. It tries to compensate the disadvantages of the WWW by offering the following tools for structuring a distributed information space.

- Links in HyperWave are bidirectional and stored as separate data units on the HyperWave server. This technique enables the authors of HyperWave documents to keep the hyperlink structure within the HyperWave system consistent. If a document is deleted, all the links pointing to it can be deleted. So updating of the database is done while editing.

- HyperWave provides a hierarchical data structuring scheme similar to Gopher. Additionally to the hyperlink structure the HyperWave data can be searched by browsing through a document tree.
- Hyperlinks can appear in any kind of multimedia data. This document describes the possibilities of incorporating hypermedia links into audio data.
- HyperWave is aware of any changes in its document structure, maintaining an index over this data becomes easy.
- HyperWave provides a user and group structure for its documents and allows access control.
- The system supports multiple languages.
- The data on HyperWave servers can be accessed through highly customizable gateways by WWW clients.

These design concepts make HyperWave one of the most interesting information systems for the Internet. In fact, HyperWave is said to be the first information system of a new generation.

The fact that links are stored not in the document itself, but in a link database improve the behavior of links while editing, deleting and following the links. A browser does not have to parse through the data in order to know where links occur.

Additionally, separate links offer the possibility of links in any kind of document. With this design concept not only text but all kinds of data can be embodied with hyperlinks. It is not the data format that has to be redesigned, but just a additional anchor specification has to be added, and then links in a new format are possible. This additionally helps to keep the link structure independent from the encoding of the data format. For example different file formats used to store pictures can have the same anchor format, by encoding link anchors in terms of normalized coordinates from [0,1]. The same thing is particularly true for audio data. Links in audio data are just encoded as start time and stop time, which is independent from the effective data format in which the audio data is stored.

Chapter 3

Digital Audio

3.1 Audio Signals

An audio signal consists of pressure differences in the air (or any other medium). These differences, if they occur periodically and steadily changing, are transformed to audible signals within our ear and the corresponding parts of our brain. Basically, these vibrations build the information of an audio signal. In order to store or transport this information it has to be transformed into an electric signal which is done by a microphone. To reproduce the signal it is retransformed with a loudspeaker into vibrating air. Storing electric analog data in a digital device makes it necessary to quantize it. The different forms of quantization are one reason for the differences in audio formats, which will be discussed later.

The audio signal has a few specifications which it has to meet in order to be audible.

- The signal should change steadily. Only changes in the air pressure are audible, not the air pressure itself.
- Our ear has lower and upper frequency bounds for which signals are recognizable. Only those frequency components of a signal which are within these bounds are transformed into neural stimuli. Generally it is said that our hearing range extends from 20 Hz to 20000 Hz with the upper bound decreasing with our age (1kHz per 10 years loss of bandwidth).
- A signal should have a minimum sound pressure level and should not exceed a maximum level. The range of sound pressure level, we are able to hear is 0 dB to 120 dB (which makes a difference of 120 dB to be quantized).

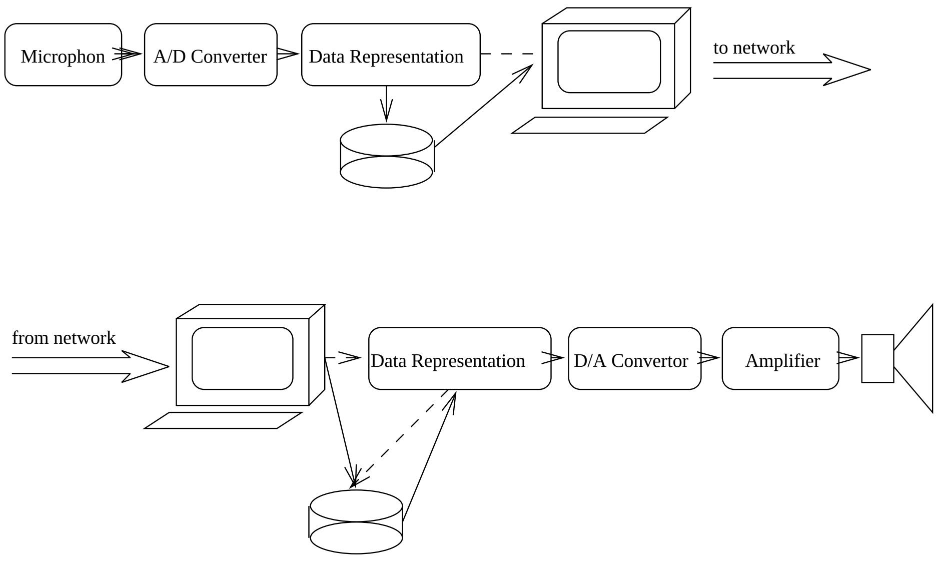


Figure 3.1: Transport of audio data over a network.

A signal which is ideally quantized should meet these specifications to be reproducible without an audible difference.

Figure 3.1 shows a block diagram of an overview for audio data transport. Basically the following steps are to be taken in order to transmit an audio signal from its source to its destination using a digital network.

- On the sender's side:

    1. Transform the audio waves into an analog signal (using a microphone)
    2. Transform the analog signal into a digital signal via quantization and sampling.
    3. Optionally process the signal according to its later purpose.
    4. Optionally store the signal in an appropriate device (Harddisk, CD, . . .)
    5. Transmit the signal. Typically using a computer and a network.

- On the receiver's side:

    1. Receive the data from the network.

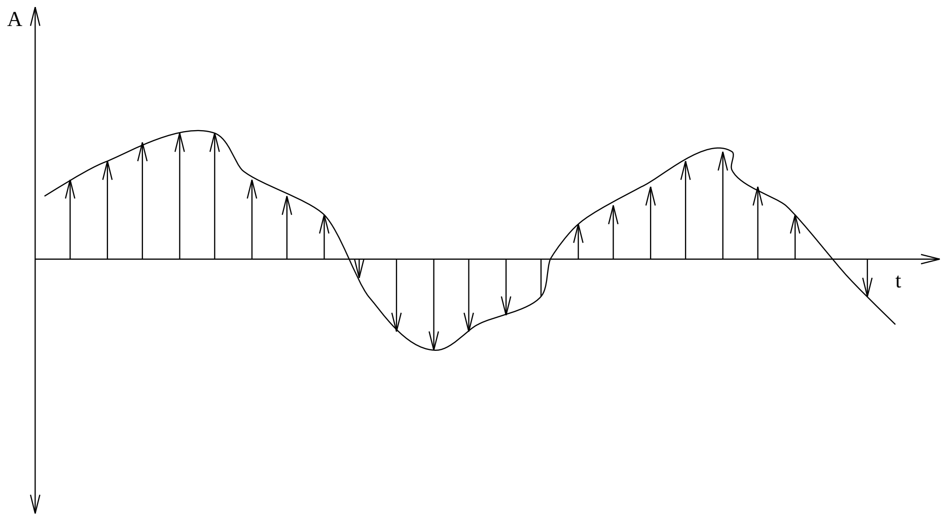


Figure 3.2: Example for quantization of an arbitrary signal in the time domain.

2. Decode the signal if encoded.
3. Store the signal either in encoded form (using less storage) or in raw form (perform the decoding only once).
4. convert it into analog form.
5. Produce audio waves with the help of a loudspeaker and amplifier.

The steps which should be taken over by a software application for transmitting audio data over the network are step three, four and five on the sender's side, and step one, two and three on the receiver's side.

3.2 Quantization

In order to come up with a digital representation of a signal, the signal is put into discrete values. Figure 3.2 shows the time quantization of an arbitrary signal. The chosen steps for time resolution depend highly on the characteristics of the signal. These characteristics are described in the Fourier transform theory [AVO75]. Basically the Fourier analysis describes every signal $x(t)$ as summation of weighed cosine waves. Each of the waves having a different frequency, amplitude and phase.

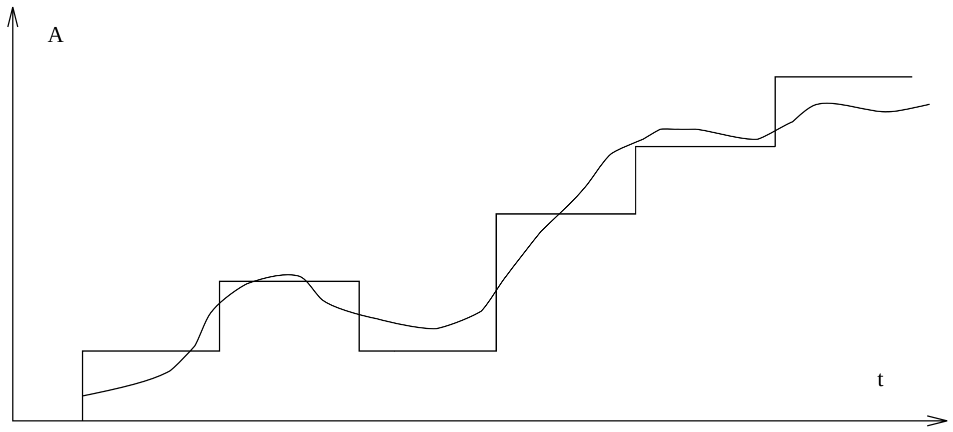


Figure 3.3: Difference between original signal and quantized signal is known as quantization noise.

$$x(t) = \int_{-\infty}^{+\infty} X(f) cos(\omega t + \phi)$$

$X(f)$ is called the Fourier Transformation of x(t) and is described in terms of the frequency. According to the Fourier Analysis and the sample theorem [AVO75] a signal is fully reproducible after sampling if it is band-limited. That means, if the signal does not contain any frequencies higher than a distinct border, then a signal is, when sampled with the appropriate frequency, fully reproduce-able. It is worth noting that the signal after quantization in the time domain still consists of continuous (non-discrete) amplitude values and still has not any loss of information.

The next step to be taken is the amplitude quantization. After this step, the signal is not reproducible in its original state anymore, because of the inevitable quantization noise which is added. Figure 3.3 shows the origin of this noise, which is supposed to be random with an amplitude of half a bit.

3.3 Sample Rate and Aliasing

The sample rate is also called sample frequency, it's unit is Hz (Hertz). In order to quantize a signal with an upper bound of 20 kHz it is necessary to sample it at two times the desired bound, which is in this case 40 kHz, to keep it fully reproduce-able. The signal must not contain frequencies higher than this bound, because of the occurrence of aliasing frequencies. Figure 3.4 shows how these aliasing frequencies come into being.

Signal 1 has half the frequency (twice the period) of the sample frequency. It can be correctly reproduced. Signal 2 has a period which is too small, and hence a frequency, which is too high to be correctly reproducible. It will be reproduced as a signal with the following frequency f_3:

$$f_3 = Samplefrequency - f_2$$

Since the new frequency does not belong to the original signal it has a very bad effect on the quality of the reproduced sound, which means it will disturb the original signal more than any other distortion, produced by signal transformations.

3.4 Signal to Noise Ratio

As pointed out in the introduction to this chapter, our ear is able to resolve the dynamic range of an audio signal in 1 dB steps. In fact, dB is a ratio, and defined as $20 * log(P_1/P_2)$. P_2 is the amplitude of the smallest audible signal at the frequency of one kHz, measured in air pressure (Pa). It's value is 20 μPa. So 0 dB means the signal having 20 μPa altering air pressure. 1 dB is 22 μPa, 20 dB is 200 μPa. This unit is called sound pressure level. The examples above show, that if the air pressure is multiplied by 10, the sound pressure level increases by 20 dB. Important is, that if the air pressure is multiplied by two, the sound pressure level approximately increases by 6 dB.

Now, if we quantize the values of the audio signal, we have to take care of meeting the demands of the dynamic range which our ear is able to resolve, in order to obtain the best possible quantization resolution. If we add one bit to the value representing the audio signal sample, we are able to quantize two times the range as before, which means 6 dB additional sound pressure level. 16 bit quantization means $16 * 6dB = 96dB$ dynamic range. This is very close to the maximum resolution of our ear. (Supposing nobody really wants to hear sound with a level close to the edge of pain, which is 120 dB). The highest dynamic range in music is produced by classic orchestras with approximately 100 dB.

With a quantization noise of $\frac{1}{2}bit(2^{-1})$, we could obtain a theoretical Signal to Noise Ratio of 96 dB (Same formula as above, the SNR increases by 6 dB for each additional bit). In practice this ratio is impossible because of the lack of ideal D/A and A/D Converters.

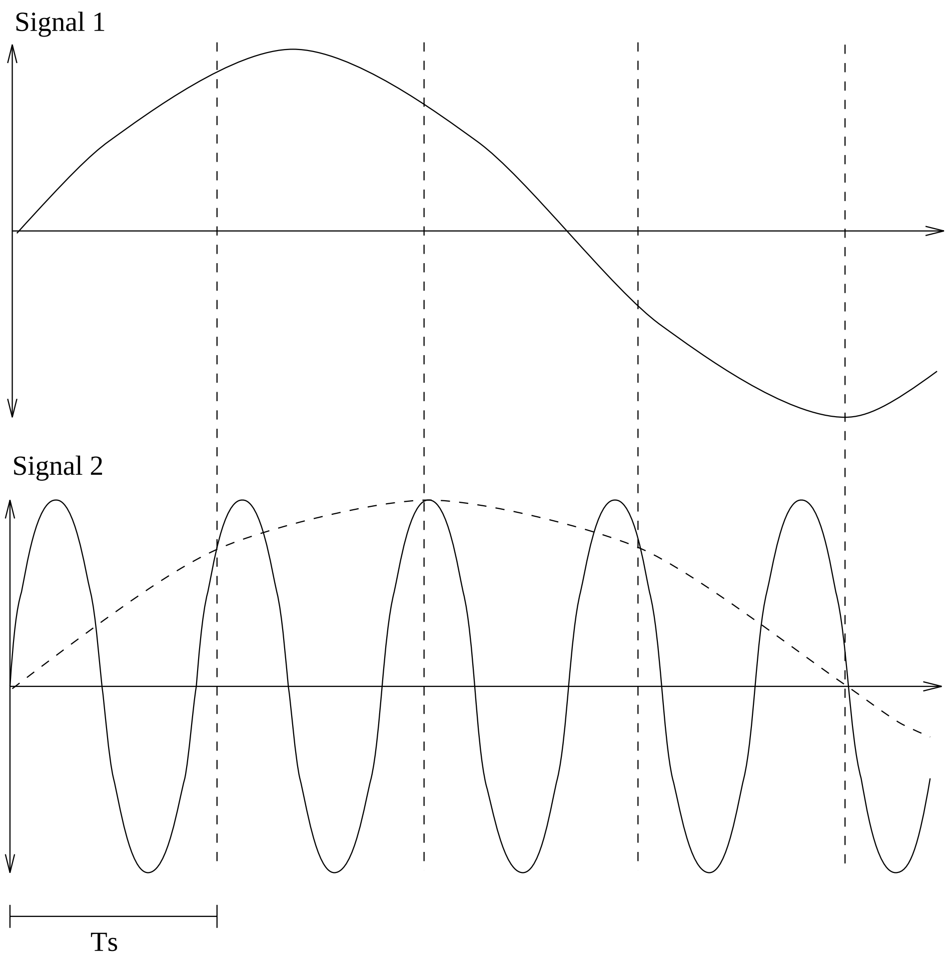


Figure 3.4: Sampling the signal at a lower frequency leads to an aliasing frequency.

3.5 Sample Rate and Sample Resolution in Practice

Supposing we have a stereo signal with 44 100 Hz sample rate, and 16 bits per sample dynamic resolution, we would get $2 * (stereo) 2 * (16bit) 44100 = 176000$ bytes per second, which is 1,408 Mbit/s . This is the standard sample rate and dynamic resolution value for audio CD's. The standard for DAT is even higher (48 kHz sample rate). This data rate is much too high to be easily transmitted over standard local area networks.

Luckily, in most applications it is not necessary to use full bandwidth and full dynamic range:

- Speech over a normal telephone circuit is band-limited with 4000 Hz.
- Most recording and playback devices do not have an appropriate Signal Noise Ratio to resolve with 96 dB.
- For speech transmission it is not necessary that the audio is transported without a loss of quality, but without a loss of intelligibility.
- Most applications do not need stereo sound.

Although the quality of audio for speech transmission is not very important, the more important it is for music transmission. There are several possibilities to decrease the need of data bandwidth for music and other applications:

- Reduce the dynamic range. FM broadcast stations for example use a very compressed signal in order to increase their broadcasting range. For most kinds of music it is enough using 8 bit instead of 16 bit.
- Reduce the sample rate. Most people do not hear frequencies up to 20 kHz, and most of the audio signals do not even contain frequencies up to that level.
- Use an encoding algorithm. Sony Mini-disk systems and Philips DCC offer CD qualityßound at much lower data rates using psychoacoustic properties for data encoding (MPEG). For lower quality speech encoding it is possible to reduce the data rate down to 2000 bit/s or less (LPC,CELP).

Due to the limited storage, transmission possibilities, and the different application purposes for audio data, the need for different audio formats is obvious. They will be described in the next chapter.

Chapter 4

Audio Coding and Audio File Formats

There are two topics, about which one should be concerned, in dealing with audio formats. The first is the method for storing audio data on disk, the so-called audio file format, the second is how the effective audio data is to be encoded, the audio coding technique. These two topics are closely related, so that the methods of storing the audio data on disk and memory should provide information about the encoding of the data. Nevertheless there do exist some older audio file formats which does not provide that kind of information. These formats are mostly vendor and hardware specific (e.g. the hardware cannot handle any other audio codings, so there's was no need to specify the format). With the possibilities of today's hardware these old style formats are rarely used, and are in fact of nearly no use on an information system with different computers and hardware possibilities, such as on the Internet.

A source for information on audio file formats and audio coding techniques and their standardization in Europe is the Open Information Interchange Initiative [OII]. Most of the standards are described by the International Telecommunication Union ITU (formerly Comité Consultatif Internationale de Téléphones et Télégraphes CCITT).

The OII is aiming at a standardization, and most new developed audio coding techniques are provided with their own audio file format (for telephone formats there's no need for a corresponding audio file format). Only a few audio coding techniques are used for storing and transmitting audio on the Internet. The most common formats, (besides raw sampling data) are μ-law and probably the Real Audio format.

Audio file formats are divided into two classes. The “native” audio file formats, which are able to store just one audio format, and the interchange file formats (AIFF, RIFF(WAV), .snd) which are able to store different types of audio coding techniques. All modern audio file formats are interchange formats.

I will try to separate these two topics in order to give a clear overview. For the audio programmer, this means supporting a specific audio file format means in some cases supporting more than one audio coding technique. Luckily there do exist different kinds of application programming interfaces (API) for handling different audio file formats. This topic is discussed later.

4.1 Audio Coding Techniques

4.1.1 Raw Sound Data

As shown in the previous chapter, audio data has three native properties, which are:

1. Sample rate.
2. Sample resolution.
3. Number of channels.

These three properties apply to every kind of audio format. Raw sound data has only these three properties and one additional property, concerning the representation of the sound data in the hardware. If a computer stores data in its memory, it is stored byte by byte. This means, even if the processor is able to handle 64 bit in its registers, the data is built out of 8 bit broad units. If a processor fetches a 2 byte (16 bit) word from memory, it can do it in two ways. First fetching the less significant, then the most significant byte of the word or reverse. The two kind of processors are called little endian machines and big endian machines. In order to get the right results out of 16 bit data, the sound application has to take care of the storing order of the two bytes of an integer. So, when reading a 16 bit value (short integer) from a file, where bytes are stored in big endian order, and handling the data on a little endian machine, the bytes have to be swapped !

4.1.2 Mu-law and A-law Encoding

The μ-law coding scheme is originated from an US telephone standard (CCITT standard G.711). There is also an European standard, called A-law, which is similar to μ-law, but is not used on computer networks. The μ-law code is an 8-bit code which represents a dynamic range of 14 bit. This is achieved through logarithmic encoding. On Sun platforms there exists a device called /dev/audio with built-in support for μ-law encoding. On machines which do not have hardware support for μ-law, the data has to be decoded into 16 bit words in order to not loose any quality. The mathematical definitions of A-law and μ-law are the following.

$$y = \frac{sgn(m)}{ln(1+u)} * ln(1 + u * \frac{m}{mp}) \qquad \frac{m}{mp} =< 1$$

value of u = 100 mp . . . signal peak, m . . . signal value

4.1.3 GSM and Other Telephone Standards

Global System for Mobile Communication (GSM) is an European telephone standard. It is used to transmit encoded speech over satellites. It is not used that often in computer network transmission, but software for decoding and encoding does exist. GSM 06.10 encoding scheme compresses frames of 160 13 bit samples into 260 bits. For compatibility with Unix the software available is capable of handling 160 16-bit linear samples sampled at 8 kHz and code it into 33 bytes. The method is very lossy, but provides a compression rate of 1:13. The proposed extension for gsm coded files is ".gsm".

4.1.4 Standard Compression

Audio data is hard to compress with standard methods. The second disadvantage of standard compression (for example Huffman coding) is its use of computer resources (mainly CPU time). On Macintosh computers there used to exist a headerless audio file format, which was in Huffman coded form. Additionally Huffman coding is used within MPEG-1 audio layer III (See later).

4.1.5 Pulse Code Modulation (PCM) and Adaptive Differential Pulse Code Modulation (ADPCM)

These two form standards for transmission of data over serial lines (CCITT G.711 and CCITT G.722). Pulse Code Modulation is a form of transmitting data with a resolution of several bits on one bit lines (PCM bit-streams). In this context, PCM is the transmission of a 4kHz mono signal in μ-law coded form. This signal then is to be transmitted over a ISDN line with 64 Kbit/s. In the context of computer based audio application, the “Raw Sound Data” is also referred to as being PCM data.

Adaptive Differential Pulse Code Modulation (ADPCM) accomplishes 7kHz rated signals at the same data transfer rate by just decoding the differences between two consecutive audio samples.

4.1.6 MPEG-1 Audio

MPEG is the acronym for Moving Pictures Experts Group. This group developed a standard called “Coding of Moving Pictures and Associated Audio for Digital Storage Media”, including three layers for audio compression. The MPEG-1 standard is designed to work with the data rates from standard double speed CD-ROM storage devices. The MPEG-2 standard is designed four quad speed CD-ROM drives, but has not yet established itself, and so most of the data is still stored in MPEG-1 format.

The three audio layers of MPEG-1 offer different compression rate at increasing compression time and delay introduced through compression. The first layer offers fast coding and decoding. The disadvantage of layer I is, that the compression rate is not that high as in other layers, and there is a audible difference between the original audio and the MPEG encoded audio. The most commonly used format is MPEG-1 layer II. It achieves compression rates of 5:1 to 12:1 using lossy coding with psycho-acoustical properties. Even though the compression is lossy, it still sounds very similar to the original signal due to the use of these properties. The best and most complex coding scheme is layer 3. A layer 3 coder/encoder is able to code (encode) all three layers of MPEG-1 audio. On most computer systems this layer III coding/encoding cannot be done in real-time. MPEG-1 supports only up to 2 channels.

MPEG-2 audio will be an enhancement to MPEG-1 audio, with the main difference of supporting 5 full bandwidth channels and one 100 Hz special effects channel. [MPE]

4.1.7 Real Audio

Real Audio [Rea] is a format developed for transmission over a network using UDP (see Chapter 1). Although the standard is proprietary and owned by Progressive Networks Inc. it became very popular, because no transmission delay is introduced through slow networks. The format is widely used by Internet Broadcasting Companies to transmit real time audio. Real Audio players are free, but if a site wants to transmit Real Audio Files by itself it has to purchase a Real Audio server.

The Real Audio format is designed to work over standard modem lines using 14400 baud (with .ra files) and 28800 baud lines (with the new .ram files). It adapts itself to lower transmission rates by just accepting the loss of some UDP packets. The packet loss is not that disturbing, because of the clever arrangement of the audio data within each UDP packet. Each packet does not contain subsequent audio data, but some part of the audio data of the last 4 seconds. If for example a UDP packet containing 100 ms of sound gets lost, it will not be silent for 100 ms, but the next 4 seconds there will be a distributed loss of some data.

The Real Audio group does not provide any clear information on the coder, but it is obvious that no other complicated compression scheme can be used, because of the immunity of the player to UPD packet loss. (This is in fact just an assumption, it could also be solved with an greater amount of redundancy, but then again the data rate to be transmitted would be increased). The Real Audio protocol must provide its own communication channels and is in fact not a real WWW application, but an application which is called up by a WWW client/server. In case of packet loss (which is inevitable with slow links using UDP) it does not make any sense to store the audio data locally.

4.2 Audio File Formats

For Internet multimedia transmission the MIME (Multimedia Interchange Mail Extension) protocol has been developed [Moo93] . This standard describes rich text formats, video, audio and other attachments to Electronic Mail. However, MIME typing is not a very reliable way to determine the audio file format of the data sent. Therefore it is important to be able to distinguish between different audio file formats by reading the first few bytes, which is possible with all formats that contain a header.

The following sections describe the most important audio file formats [vR95].

4.2.1 Creative’s .voc Files

MIME type: audio/x-voc
Extensions: .voc

The .voc audio file format [vR] was developed by Creative Labs Inc. for their Sound Blaster cards. It is a file format made out of chunks, which defines different kinds of blocks. Each of the blocks starts with a type byte and three length bytes. With this method it is possible to jump from block to block:

- Header block:
  This block contains a version number and the offset to the effective data blocks.
- Sound data block:
  The first two bytes of this block are sample rate and compression type, then the actual sound data follows.
- Sound continue block:
  This block just contains sound data with no format specification. The format specification of the last sound data block is still valid.
- Silence block:
  This block just has a length. The unit of this length is sampling cycles.
- Marker block:
  Allows markers to be set between the different chunks.
- ASCII block:
  Allows text data to be embedded into the audio file.
- Repeat block:
  Has the number of repetitions as first two bytes.
- End repeat block:
  Marks the end of the part, which is to be repeated.
- Terminator block:
  Marks the end of the audio file.

The .voc file format supports at most 8-bit samples, and therefore cannot be used for high quality audio output.

4.2.2 NeXT .snd and Sun .au Files

MIME type: audio/basic
Extensions: .au .snd

The NeXT .snd and Sun .au files contain a header with information on the currently stored audio data and the audio data itself. The header consists of the following structure:

```
typedef struct {
        int magic; /* magic number ‘‘.snd’’ */
        int datalocation; /* offset to the data */
        int datasize; /* the size of the data */
        int dataFormat; /* the data format code */
        int samplingRate; /* the sampling rate */
        int channelCount; /* the number of channels */
        char info[4]; /* optional text information */
} SNDSoundStruct;
```

The representation of the audio file format on disk and in memory is described by this structure. The magic variable is the magic number, in order to recognize the data as a .snd or .au file. It contains the number 0x2e736e64, which happens to be “.snd” in ASCII code. The data-location field points to the beginning of the sound data, counted from the beginning of the file. In other words it contains the length of the header. This makes it possible to expand the info field from 4 bytes to user defined length. `dataFormat` stores the audio format of the following data. The numbers 0 - 255 are reserved by NeXT for specification of internal formats (mainly the number of bits per sample, but also some compression schemes like μ law with silence detection), the numbers above 255 can be used for user specification of formats. `SampleRate` and `channelCount` are what they say: Sample rate and Number of channels in the sound data.

If the application recognizes the header data structure it can easily derive the properties of the data and, after requesting or emulating the specified hardware resources, play the raw data. The .snd and .au files are in big endian format, but there is also a file format similar to .snd used by DEC systems in little endian order.

4.2.3 SGI, Apple AIFF and AIFF-C Files

MIME type audio/x-aiff
Extensions: .aif .aiff .aifc

Originally developed by Electronic arts Inc. the IFF format was extended to AIFF (Audio Interchange File Format) by Apple [AIF89], the AIFF standard is more flexible than NeXT's .snd format, because it does not use one fixed header, but several data-chunks, where one of those is the sound data chunk. Here are the data fields common to all AIFF chunks:

```
typedef struct {
        ID ckId;
        long ckSize;
        char ckData;
} Chunk;
```

where ID is a 4 byte char field.
The chunks are as follows:

- fixed header describing the absolute length of the file and the format (AIFF or AIFFC), also called form chunk ID = FORM
- common chunk: describes the format of the sound data, sample rate etc. ID =COMM
- a sound data chunk containing the sound data. ID = SSND

These three chunks have to be included in the minimal AIFF audio file. Additionally there are

- Marker chunk: contains one or more markers or cue-points into the sound data
- Comment chunk: contains comments which can be references to marker chunks.
- Instrument chunk: this data is mainly used by sample players. It offers the possibility to include loops, attack, decay and other sampler related information such as base note of the sample, velocity, ...

- Raw MIDI chunk, AES data chunk, User defined data chunks,...

These data chunks make the AIFF audio file format very flexible for use in professional audio applications. The AIFF format was expanded for compressed data into the AIFF-C format. Developed for Apple with the Motorola processors, the AIFF File format only supports big endian byte order.

Audio coding techniques supported are μ-law, A-law, ADPCM and other.

4.2.4 WAV (RIFF) Files

MIME type: audio/x-wav
Extensions: wav

The most popular audio file format on the Internet is the WAV file format from Microsoft [AIF92]. Even if the Unix platforms all have their own preferred audio file format other than WAV, every application on the Internet dealing with sound should be able to read this kind of format in order to access the large amount of already stored data.

The WAV File format is very similar to Apple's AIFF and AIFF-C formats. It contains different chunks of which the following are mandatory.

- The file starts with the magic pattern RIFF (denoting the beginning of the chunks, and used as a container).

- In the WAVE chunk the properties of the sample data chunk are described.

- The sample data chunk may contain one block of sampled data or more blocks of sample data and silence.

Additionally to these mandatory chunks there are the following chunks, which often do appear after the sample data chunk.

- The cue chunk provides marker to different cue points in the sample data chunk.

- The play-list chunk describes in which order to play the different cue points.

The order of bytes is always little endian, the preferred byte order of Intel processors.

4.2.5 MIDI File Format

Musical Instruments Digital Interface is a standard which was developed by different synthesizer vendors. It has two parts, a hardware specification and a data transmission protocol. The data transmitted consists of control signals telling which note a synthesizer should play. It is possible to control up to 16 instruments with one MIDI data stream. The MIDI File Format [MID] is a file format for this kind of information (different from the MIDI Sample Dump Format mentioned in the next Section, which really is a format for audio data). The MIDI File Format is very compact, because it mainly contains control information. In addition to control sequences, a MIDI file contains time stamps in order to determine when a MIDI event should occur. Principally the distribution of MIDI data over the Internet does not make any sense, because different synthesizers produce different sounds with the data, but since Creative Inc. sold its Sound Blaster audio cards with built in FM-synthesizer (Yamaha's OPL synthesizer chips) most PC's with audio hardware are able to play MIDI files. The synthesizer chip provides a certain degree of uniformity.

4.2.6 Other Audio File Formats

Just to mention them, there are other Audio File Formats, but it is not very likely, that somebody will try to distribute them over the Internet. Among those are:

- The AVR File Format (Audio Visual Research) is used by several commercial applications on the Macintosh and on Atari ST. This is some sort of enhanced NeXT .snd format.
- MIDI Sample Dump Standard is a file format that is used when transporting sample data between synthesizers.
- NIST SPHERE format is widely used in the speech processing community.
- The AFsp library (see next chapter) proposes a audio file format (enhanced NeXT/Sun format) for speech processing. The advantage with enhanced NeXT/Sun formats is their their upward compatibility.
- DigiDesigns Sound Designer I and Sound Designer II formats are used on Power PC's with their ProTools package (a widely use hard disk recording system) and the Sound Designer sound editor.

- IRCAM, BICSF, csound is a sound file format used in computer music, but never became that popular in commercial applications.
- EBICSF enhanced BICSF is a legal NeXT - .snd format with additional BICSF data in the user defined area.
- MOD files originated from the Amiga and are something between files with digitized sounds and MIDI files. This format contains short sound samples and a scheme how these samples should be played back. With this method they have a relatively small file size.

Chapter 5

API's for Audio Programming

The main problem of doing audio programming on different Unix platforms is that there is no common API (Application Programming Interface) for audio output. Every vendor has developed its own API: mostly just opening, reading and writing to audio device files, or an audio server as in SGI's IRIX.

Some effort has been taken to provide an audio interface similar to the X-Window System. In these systems, audio is handled by an audio server. Clients can connect to the server and make requests for audio output or input. The audio data is then transferred to the server. When communicating with TCP/IP it is possible for those systems to be network transparent, which means that the application can run on a different machine than the audio-server does. SGI's audio server for example uses System V IPC (Inter Process Communication) for communication and therefore is not network transparent. Because of the fact that a many UNIX systems do not have any audio hardware, it is an advantage of having a system like this, so the audio output can take place on any X-Terminal which is capable of doing audio output. This chapter describes some of the network transparent systems and vendor specific API's.

5.1 Audio Programming on Unix Platforms

5.1.1 Unix and Real Time Applications

Unix was never designed to meet the requirements for a real time processing environment. Audio output is a task which requires real time abilities. Audio output requires a minimum call-back time. Call-back time is the time after which the audio process should put new data to the output queue, after the data was sent to the audio hardware. By providing a buffer for audio, this minimum call-back time can be extended to the time-equivalent of the audio buffer size. With this technique it is possible to meet the real-time requirements of audio output under normal conditions, ie.:

- The system load should not be too heavy.
- The computational costs of the audio output is small in comparison to the system performance.

Real Time Extensions for Unix Systems

In order to meet the requirement for a better real-time ability of Unix systems, different systems have included real-time extensions to their operating systems. The IEEE 1003.1b standard (better known as POSIX 1b) proposes these extensions, including:

- The possibility of changing the scheduling policy and priority for real time tasks.
- The possibility of memory locking (define a part of memory which cannot be swapped out).

These system calls are very dangerous and can cause the system to hang. Therefore they are only allowed for applications with superuser rights. For a distributed environment it is not stable enough to use this kind of process scheduling. Use of these system calls is one step towards using UNIX platforms as personal computers, in the sense where one person can control the processes which are run.

Finally, without using the real-time extensions it is not possible to guarantee the proper working of an audio application. All that can be said is, in most situations it will work.

5.1.2 Synchronizing Audio and Event Processing

As discussed before, audio output is a very time critical task. It was said that the call-back time should not exceed a critical amount of time. When doing audio output and event processing at the same time, it is important how this call-back is achieved. Principally there are the following possibilities:

1. Provision of a separate process for audio output.
2. Task scheduling is done in one single process.

The advantage of the first solution is, that there exist different servers for audio output, and the only thing for the application to do is to synchronize audio output and event processing. One disadvantage is, that when dealing with the audio data in the application itself (visual audio output), the data has to be loaded into the application additionally.

The advantage of the second solution is that the same data can be used in the graphical and the audio output, and the overhead of synchronization does not exist. Since the Harmony Audio Player provides visualization of audio, the second possibility is discussed in more detail.

There are several possibilities to do audio output and normal event processing in one process at the same time.

1. Open audio in non blocking mode.
2. Open audio in blocking mode.
3. Using the select call.

For the first solution this means that any write call to the audio hardware (or transmission to the audio server) should return immediately when it is not possible to write the specified amount of data. (This is the case if the output buffers for audio are full). The application can then go on with normal event processing and retry the audio operation later.

Opening audio in blocking mode can introduce some delays in event processing (the process stays in the audio output part), on the other side it is easier than dealing with the unsuccessful audio write calls. This solution can be combined with the third one.

With the select call it is possible to determine if output (input) to a specific file-descriptor is possible. If audio output is possible, then the process would go on with audio output, otherwise it will go on with event processing. This technique makes it possible to get rid of the delay caused by blocking audio calls.

The InterViews Toolkit for GUI programming provides a dispatcher class, which uses the select call to determine which part of the application's code should be started. If the audio programming API offers the possibility to use the select call to determine if audio data can be output, then this would be the best solution, otherwise the select call just points to a file-descriptor which is always writable, and the this solution would behave just like audio output in blocking mode.

5.2 AudioFile

The AudioFile system was developed by DEC's Cambridge Research Laboratory in 1989 for DEC audio hardware. Later support for other system and their audio hardware was added. It was intended to be for audio what X is for graphics. The sources of the AudioFile system are distributed freely. The most widespread version is version 2, but version 3 is already released and adds some very interesting features to the AudioFile system.

At the moment there are servers available for Digital RISC systems running Ultrix, Digital Alpha AXP running OSF/1 and for Sun SPARCstations running SunOS. The new version 3 of AudioFile adds support for SGI IRIX. Still the AudioFile system did not become that popular until now, thats why there are just a few hardware architectures for which a server does exist.

5.2.1 The AudioFile server

As in X nomenclature, the server is that part of an application which provides access to the hardware, in this case the audio hardware. The server is therefore the part of the audio system which has to be ported to different hardware and operating system types. The communication between the client and server is initiated by the client over TCP/IP. The server then responds to the request of the client, in this context by giving information about the underlying audio hardware capabilities and by finally processing the audio data being sent by the client. At the moment the server supports multiple audio devices, each with just one sample rate. Format conversion in the server was added with version 3. A great advantage of the AudioFile library is the device time feature, which allows for correct timing. A sample counter is held in the server, and data being read or written can be synchronized with this time line.

5.2.2 The AudioFile API

Typically the programming of an application to use an audio server involves these three steps:

- Opening the connection to the audio server.
- Looking for available devices and audio capabilities, and trying to set up the server correctly.

- Sending the data to the server.

The AudioFile system version 2 did not provide data conversion utilities, therefore the data had to be in the correct format when sent to the server. With the version 3 of AudioFile, this has changed and the whole SOX library (see later) was included into the source tree for accessing different audio file formats.

5.2.3 Advantages and Disadvantages of AudioFile

Obviously the disadvantage of the AudioFile system is its lack of support for a wide range of UNIX systems. Therefore it is not fully suitable for an audio application, which should run on most UNIX systems available today. As with every other network transparent system a application requires the audio server to be installed. This is sometimes a reason for not using the application’s audio facilities. Numerous features were added to the AudioFile library in version 3.

A second disadvantage until now was, that the data sent to the server has to be in hardware specific format, and so the application has to do all the conversion routines. This has changed with the new version 3. Still the data has to be sent with the correct sample rate (Most of the servers provide just one sample rate). This will be changed in the next release of AudioFile.

5.3 NAS

The NAS (Network Audio System) is in its spirit the same as AudioFile, with some additional features in server capabilities. NAS also tries to be X for Audio, which means it is a fully network transparent system. It was developed by NCD and therefore support for NCD X terminals or even NCD PC-X servers is available. NAS is also available in source format and porting to different platforms is easy. Currently NAS provides support for Linux (OSS), SGI, Sun and HP platforms.

5.3.1 The NAS server

The NAS server is divided into two parts. One part is device-dependent and is equivalent to the AudioFile system server. The device-independent part of NAS is what makes it more powerful and more complicated too. The NAS server provides different utilities in the server, which are not offered by other systems yet:

1. Sound data can be stored in the server, in so-called buckets, therefore the overhead of data transmission is not necessary anymore with some applications. The amount of audio data that can be stored depends highly on the memory capabilities of the hardware the server runs on.

2. Audio in the server is handled in so-called flows, which makes it possible to connect, add and multiply different audio streams in the server.

3. Data conversion is done in the server, the client just has to tell the server about the format being sent.

5.3.2 The NAS API

In analogy to the server capabilities the NAS API provides functions for accessing these. When playing sounds which are stored in the server the application has to track event call-backs from the server. These call-backs can be of different types (overflow, underflow, timer call-backs, etc.). There are a lot of high level audio programming functions which for example play a file from disk, or write an audio file to disk, play audio data from a bucket and so on. Additionally to accessing the server capabilities the NAS API provides a library for reading different audio file formats. Audio file formats supported are:

- .snd, .au - Sun NeXT formats
- .voc Sound Blaster's native format
- .wav Microsoft's audio file format
- .aiff Apple and SGI format

Different high level routines are provided for fast and simple implementation of common audio tasks.

5.3.3 Advantages and Disadvantages of NAS

The NAS comes closer to the goal of being some kind of X for audio than the AudioFile system. Its API offers many of possibilities from low level routines to high level audio playback and record routines. There is no other audio library which offers these high level routines. Most of the audio API's just offer routines for reading and writing one audio file format, whereas NAS provides routines for the four most widely used formats.

The problem with NAS is, that it is far from being a standard (like X) and one can assume that a user wanting to use an application with NAS support must additionally install the NAS system on his display host. NAS does provide the possibility to do a select on the connection socket and is therefore well fitted for audio output in a select loop. Overall at the moment NAS seems to be the best network transparent audio system available.

5.4 Broadway

Broadway Audio System [Bro] is a part of the X Consortium's new efforts to create a network computing standard. Development started in early 1996 and is not finished yet. The specification of the future API exists and so it does make sense to compare it with the other network transparent audio systems. The Broadway Audio Systems has most of its functionality from NAS and additionally uses the time-line principle of AudioFile for better synchronization.

5.4.1 The Broadway Audio Server

The Server for the Broadway Audio System will provide the same functionality as a NAS server. Additionally it will be possible to define extensions to the system. The server handles format conversions, multiple audio streams, but does not provide (in its first implementation) the sophisticated connection structure of NAS. The time-line principle offers the possibility for closer synchronization.

5.4.2 The API for Broadway

Again as in NAS the API provides low level routines to play from buffers, and high level routines to make it easy to play directly from a file. The C API is written in an object-oriented manner, and therefore provides better encapsulation and separation of audio objects. There are objects in the server, which can be accessed over the network from the client, and objects in the client, examples for server objects are:

- Port objects are objects to whom the client sends audio data.

- Format objects are connected to port objects and device objects and define the format the data is in.
- Device objects are objects which access the hardware directly. Data conversion from port objects to device objects is done automatically using their appropriate format objects.

Additionally there are client side objects:

- File objects allow access different audio file formats
- Player objects encapsulate all the low level objects for audio playback and can be constructed with a file object

The objects have a small set of operations in common.

5.4.3 Advantages and Disadvantages of Broadway

Being the newest approach to a common network transparent audio system, the Broadway Audio System is able to combine the advantages of the other systems. It does this in several ways. It uses for example the format handling from NAS and the synchronization handling from AudioFile. Additionally its style stays close to the X Window System and therefore its API will be understood faster by programmers used to X programming, than the rather complicated API of NAS. The object-oriented paradigms additionally make the API and its possibilities more comprehensible. Coming from the X Consortium it is more likely that the Broadway Audio System will become a standard than for the other audio systems.

The disadvantage of Broadway is that it is not finished yet, and therefore not usable today. The need for it is still here, a port to Windows NT is also under discussion.

5.5 Rplay

A not very well known full featured audio server and API is Rplay [Mar], which is distributed under the GNU public software license. Originally written for Sun SPARC stations the code was ported to different platforms. Currently the program supports Linux (OSS), FreeBSD (OSS), SGI, Sun and HP-UX.

5.5.1 The Rplay Server (rplayd)

The rplayd speaks two different kinds of protocols. One for very fast access over binary UDP packets. With this kind of communication it is possible to send commands to the server, but the server never replies. It is possible to start, pause or stop a sound, or to play a sound a distinct amount of times.

Additionally the rplayd supports client communication in human readable format. It is therefore very easy to connect to the server and look at its status via the telnet command (see Chapter 1). The documentation for doing that is provided with the rplay source distribution. Most of the time audio is played directly from a file by the server. For this task the server provides support for different audio file formats: au, snd, aiff, wav, voc, ub, ul, G.721 4-bit, G.723 3-bit, G.723 5-bit, and GSM. It is also possible to send audio data directly to the rplay server.

5.5.2 The Rplay API

Programming the Rplay API is done with very few function calls. This includes opening the connection to the sever, sending commands to the server (the commands are the same as commands typed in in a telnet session when monitoring the rplayd) as NULL terminated char strings, and piping audio data with the put command. Sample rate conversion is done automatically. One disadvantage is that the server, when started remains in the same state and cannot adapt to different samples rates. This involves a sample rate conversion for all rates, that do not match, even if the audio hardware would have been able to play the specified rate.

5.5.3 Advantages and Disadvantages of Rplay

Although the rplay API is easy to learn and provides a lot of functionality, the rplay system is not very well fitted until now for programming an audio application. Its disadvantages are:

- Fixed hardware settings in the server.
- The audio file library is not accessible from the client.

5.6 Other Common Audio API's

On most platforms audio programming means using the open, read, write and specially ioctl system calls to perform action upon the audio device files. It is common on all platforms to output audio through writing to the device file and input audio through reading. What makes really the difference between these API's on different computers is the possibility of customization of the audio device through the ioctl call. The Linux system for example defines this call the following:

```
int ioctl(int d, int request, ...)
```

Where the ellipse (...) is commonly a pointer to a type and is used as argument or return value for the call. To change or query the state of an audio device it is necessary to provide the file descriptor of the device (d), then an integer variable called 'request' identifying the request or command which should be sent to the device, and some arguments to change the state of the device or as a return variable for information from the device. Each hardware has its different queries and commands for the audio device and so are the integer numbers assigned for those commands.

What these API's have in common is the possibility to use the select system call to look at the state of the device, so the blocking of the audio process can be prevented. It is worth noting that these are all low level API's, and they are used by the higher level network transparent system for accessing the audio hardware on different computer systems. On the other hand they have the advantage, that no other library has to be installed on the system, and the effort for installing the software written with these is very small.

I will just give a short overview of these API's using Suns API and the API for OSS (Open Sound System), the audio hardware specific device driver API for Linux and other Unix platforms. Section 5.6.3 covers SGI's audio server and its API.

5.6.1 OSS Application Programming Interface

The OSS (Open Sound System) API has developed from the sound driver for Linux and is now open for different UNIX systems, which use the appropriate audio hardware. The list of sound cards supported is rather long, and includes most of the ISA bus cards that are available for PC's. But every UNIX platform with ISA bus can use this device driver too. At the moment there exists a driver for Linux and FreeBSD on IBM PC compatible hardware and for Sun's SPARC stations.

There are different ioctl calls to change and query the state of the audio device. Most of the change requests return the value of the changed parameter, so the application can verify if the device is able to handle the requested parameters. For example a request for setting the sample rate to a specific value will return the effective value, to which the sample rate was set, which differs in most cases from the requested one. (Most digital to analog converters support only a few different sample rates).

5.6.2 Sun DBRI Application Programming Interface

Sun currently provides 3 different kinds of audio output devices.

- audioamd is a built in low quality internal speaker, which can output μ-law and A-law encoded data with 8000 Hz.
- The dbri (Dual Basic Rate ISDN and Audio Interface) audio device which can do high quality audio in stereo at rates up to 48000 Hz.
- audiocs - the Crystal Semiconductor 4231 Audio Interface is used in SPARC 5 systems and provides equal audio quality to the dbri device.

These three devices have a common API. Similar to the OSS API a device called /dev/audio has to be opened. The device driver of /dev/audio is just an interface to one of the three "real" device drivers. They all have a common API.

The state device can be queried for the name of its underlying audio hardware with the GET_DEV ioctl call. For setting the audio parameters, there's just one structure, which holds all configuration data for the device and can be passed via the AUDIO_SETINFO ioctl call. Just a few ioctl call are necessary for audio programming on Sun's. The Sun audio device can only be opened once.

5.6.3 SGI Soundscheme Server and Audio Library (AL) API

SGI took a different approach to solve the audio programming problem. It is not possible to access the audio over low level device drivers, but IRIX provides an audio-server concept similar to the concepts already mentioned, but lacking the network transparency. The audio server is called soundsheme and waits for requests from applications using the System V IPC system calls. The data is transferred using shared memory, which has the advantage of speed over the socket based data transfer method of the other audio servers. Synchronization possibilities are also very tight, using the IRIX specific Shared Arena system. With this it is possible to use the select call for communication between two processes which are attached to the same shared memory area. It is possible for more than one application to access the audio hardware simultaneously. The audio streams are just added together by the sound server.

On the clients side the API for programming the AL is a little bit more comfortable than the device driver API's, but it is still a low level API (There is no "play" command for a whole audio file).

5.7 Libraries for Audio File Handling

A second topic, after the problem of audio output, that is important with audio programming, is the possibility of handling different audio file formats. Most audio API's (except the NAS system and the forthcoming Broadway Audio System) do not provide the possibility of reading different audio file formats. In order not to be dependent on a special Audio API, a separate library should be used for audio file reading. This section provides information on two libraries of this kind.

5.7.1 SOX

SOX is a program for different audio tasks (commonly said to be the "swiss army knife" for audio file formats). It can convert most kinds of audio file formats and do some other tasks like sample rate conversion, and different kinds of effect processing.

Source code for the SOX package is free and includes all the conversion routines of the main program in a library called ST (Sound Tools). With this library it is possible to to read and write all different kinds of audio file formats.

Each single format is in its own file, and therefore it is possible to just include the code for a small number of commonly used formats. These properties of the SOX package make it very easy to include the SOX code in an audio application, and use it as interface to the stored audio data.

5.7.2 The AFsp Library

The AFsp library is a audio file library used in the speech processing research community. Nevertheless it is probably the best audio library for reading and writing audio on different platforms at the moment. It provides very easy access to the audio files with the possibility of positioning.

It is not allowed to include the library in a program which is distributed for a fee. Platforms supported are SunOS 4 and 5 (Solaris), IRIX 5.2, DEC Ultrix, OSF/1, HP-UX, Linux, ...
Audio file formats supported for reading include:

- Headerless audio files
- Sun audio files
- RIFF WAVE files
- AIFF/AIFF-C audio files
- NIST SPHERE audio files
- IRCAM audio files
- INRS-Telecom audio files
- ESPS sampled data feature files
- Text audio files (NATO/ESPRIT CD-ROM format)

The following file formats are supported for writing:

- Headerless audio files
- AFsp (Sun) audio files
- RIFF WAVE files
- AIFF-C audio files

5.8 Conclusion

At the moment there exists no real standard for audio programming under the various flavors of Unix. Applications using audio on different platforms should in each case be flexible enough for adding a new audio API.

The most elaborated system is the Network Audio System, although it is not possible to use its high level routines, in order to stay open for the other, vendor specific API's and for coming standards (e.g. Broadway, …). In order to stay independent of a specific audio system it is necessary to have a file interface which can be used for different sorts of audio file formats and audio API's.

The SOX package mentioned in the previous section does fits this purpose and makes the audio file interface of the application independent of audio API's. Additionally, SOX is ported to most computer platforms and its source code is free, so it is possible to include it in the application source tree.

NAS and Rplay are at the moment the only systems which do provide data conversion in the server. To be independent of this facility it is necessary to include data conversion functionality in the application itself. This is the price that has to be paid to be independent of audio API's.

Chapter 6

Audio Players and Hyperlinks

6.1 Audio Players and Editing Tools

When designing a graphical user interface, is is important to look at similar applications. Most users expect that they can interact with a new application in a similar way than with other applications with which they are familiar. On the other hand most of these applications may have some points, which could be done better. For the purpose of comparison I will describe a few audio editing applications.

6.1.1 Digidesign Sound Designer

The Sound Designer is a sound editing tool for the Apple Macintosh. It provides two different kinds of waveforms for the user. One channel is an overview of the audio data and is located at the top of the applications main window (see Figure 6.1). The second view is only a small part of the audio data. It is disturbing, that the zoomed view of the sound data cannot be expanded to fit a larger amount of the audio file in the window. If an overview is wanted, the user has too look at the overview waveform. There is no stage between the two.

6.1.2 NeXT Sound Editor

The user interface of the NeXT Sound Editor is in my opinion a little bit more intuitive than the approach Digidesign took with its Sound Designer. The Sound Editor only shows one Waveform, which can be zoomed in and out. This makes it possible to navigate in the audio data, while keeping the application's main

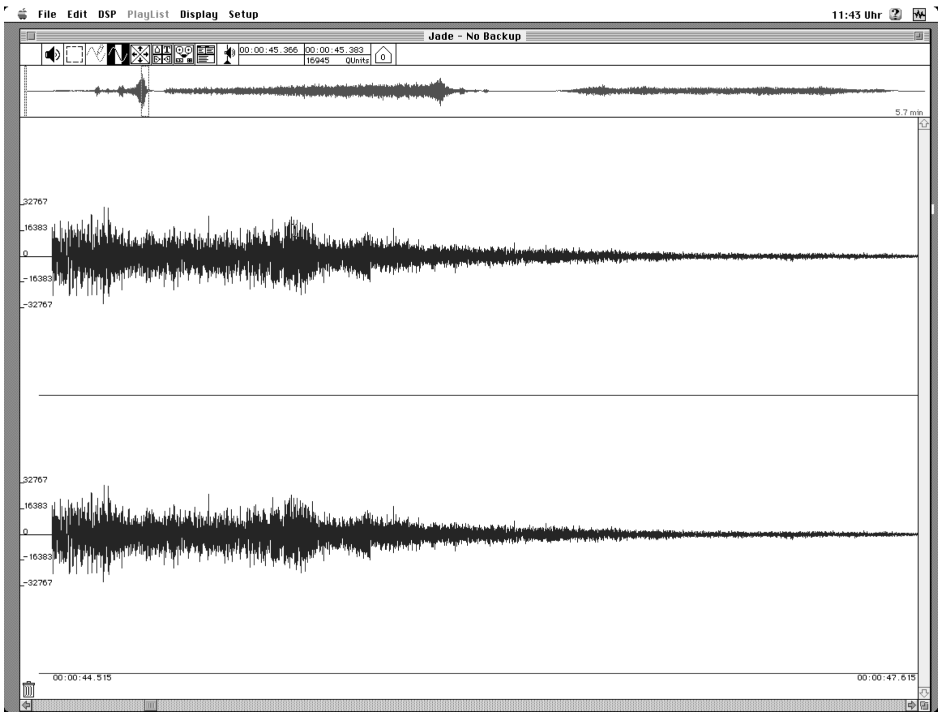
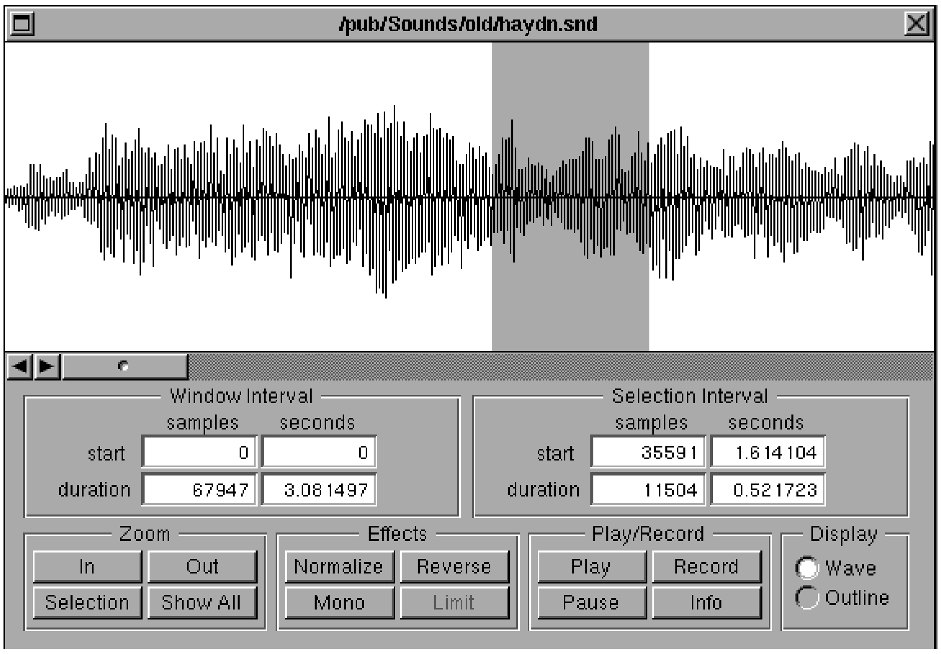


Figure 6.1: The Sound Designer user interface.



	samples	seconds
start	0	0
duration	67947	3.081497

	samples	seconds
start	35591	1.614104
duration	11504	0.521723

Figure 6.2: NeXT Sound Editor user interface.

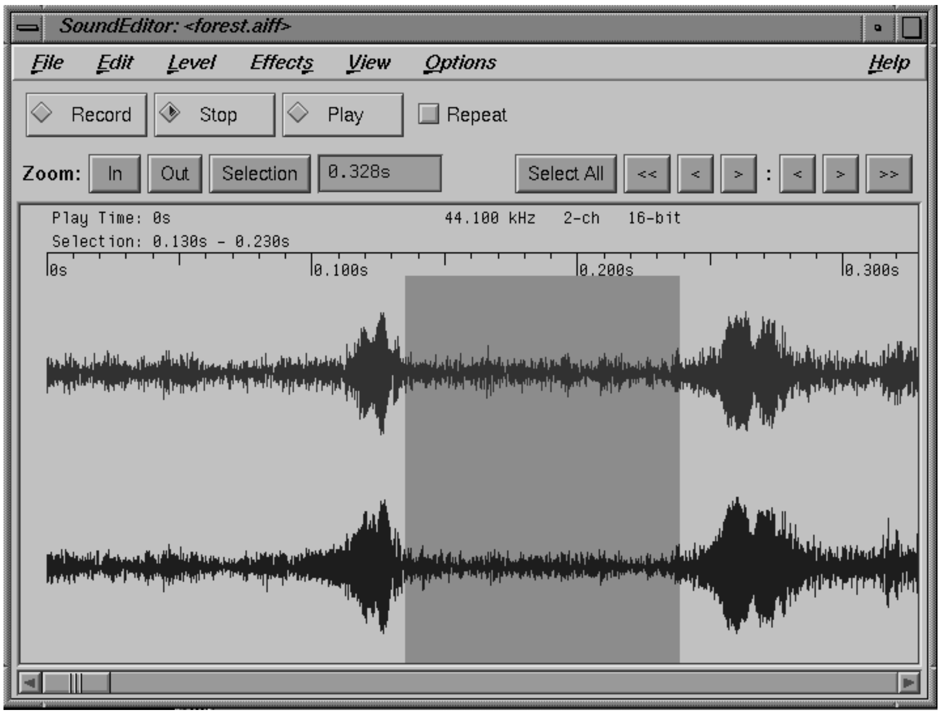
window small. Keeping the window small is one criteria for the Harmony Audio Player, because Harmony has its own window for each viewer, and it is very common that more than one viewer will be open. Figure 6.2 shows audio data in mono, stereo data would show two waveforms. The NeXT Sound Editor is able to read and write NeXT .snd audio files.

6.1.3 IRIX Sound Editor

Silicon Graphic tries to be number one among Unix workstations not only in graphics but also in audio. It provides therefore rather good audio hardware and software. The Sound Editor from IRIX behaves very similar to NeXT's Sound Editor. The SGI Sound Editor is able to read and write .aiff and .aifc files.

6.1.4 Cool Editor for Microsoft Windows

Most users will be used to programs running on a PC with Microsoft Windows. The Cool Editor is an example of a sound editor for PC's. It provides the possi-



0.328s

Play Time: 0s 44.100 kHz 2-ch 16-bit
Selection: 0.130s - 0.230s

Figure 6.3: IRIX Sound Editor User Interface.

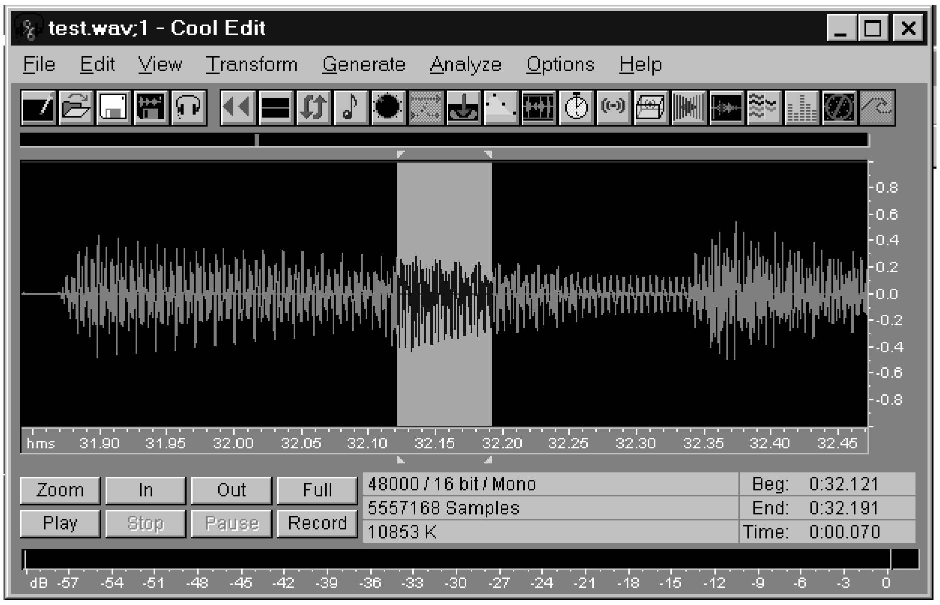


Figure 6.4: Cool Editor user interface.

bility of editing sound in the common sense, such as cutting and pasting, and the possibility to apply many different sound effects like de-noising, amplitude shaping, echo, etc. It also has a marker area in the waveform itself. An area is marked with the left mouse button. The Cool Editor is able to read and write .wav audio files.

6.1.5 User Interface Wish List

As can be seen by the examples above, most of the sound editing tools are very similar in their usage. They provide a waveform window, in which the marker is directly drawn with a different color. All editors, except the Sound Designer allow the data to be arbitrary zoomed. Most of the tools allow to zoom directly to the marked area, but in my opinion it is better to provide a fixed zooming factor. This has two reasons:

- Most of the time the important points in the sound data are the edges of the marked area. When zooming to the marked areas the edges become the edges of the wave window, and are not visible anymore.

- When zooming to the marked area, the zooming factor is arbitrary and cannot be easily judged by the user. It is better to provide fixed zooming factors, so the user knows what can be seen after the zooming operation.

The Harmony Audio Player provides a similar user interface as the NeXT Sound Editor, the Cool Editor or the IRIX Sound editor.

6.2 Hyperlinks in Audio

As mentioned in Chapter 1, the possibility of making references to other documents or data in one document (so called links) started with text documents and was expanded to different types of multimedia documents (hypermedia links). There has been different kinds of links in audio until now, even if they were not used for information systems, but for music applications. For example all samplers provide tags in their audio data (see audio file formats) where other kind of audio was inserted, or some part of the audio file was looped. These were the first kind of links in audio, but they were restricted to point to other audio data.

Different from text documents, where links are encoded in the document text itself, as in HTML, it is not useful to encode the links in the multimedia data too. The advantage of having links encoded in the document is, that if the data in the document changes (e.g. a paragraph is added) then links still stay at their correct place. In multimedia formats it is very hard to scan for links inside the data, because it cannot be said if a link marker is really a link or just audio data. Therefore links are typically managed externally and point to some place in the document.

HyperWave provides the ideal structure for multimedia links, because links and documents are different objects. It is not a disadvantage, that every time when the document changes the links are to be updated, because with audio there's no other real possibility for implementing hypermedia links.

The questions remaining are how should the user recognize links in audio? How can the user follow audio links ? These topics are discussed in the next sections.

6.2.1 Visualization of Audio Links

Until now visual links are the only kind of links that exist for navigation in hypermedia documents. HyperWave provides links inside images, inside film clips

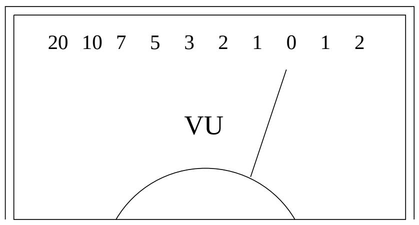


Figure 6.5: A typical VU-meter for audio visualization.

and even inside 3d models. Even if the format for hypermedia links in audio is defined for HyperWave, there is not a strict definition how these links should be represented in the Audio Player.

Since users often expect a link to be some visual icon, which can be clicked on with the mouse and followed, it would be desirable to implement audio links as visual links. Why this is not that easy has different reasons:

- Audio is not visual, which means, if implementing visual links in audio there has to be a visual abstraction for the audio data.
- Audio has (like movies) a strong time component. One cannot see a whole audio file just like one can see a text file. It is not easy to provide navigation in the time domain (like scrolling in a text).
- Normal links do not have a time interval in which they are valid, but audio links do. Audio files just have one dimension which can be used for defining links, and this dimension is time.

Visual Abstraction for Audio

There are different kinds of visual abstractions for audio, each of them suited to a special purpose.

One of the most common visualizations is the VU - meter shown in Figure 6.5, which indicates the equivalent of the power of the audio signal. This kind of visualization does not provide the right means for representing audio with hyperlinks because of its lack of memory. A VU-meter just shows the current state of the audio data, but not the previous states and the forthcoming. But principally this is one of the best abstractions for audio, because it is easy for most people to distinguish between silent and loud.

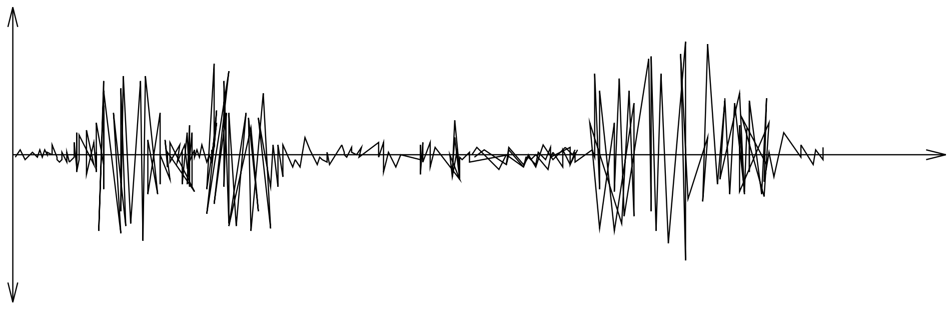


Figure 6.6: Example for a graphical representation of audio with a waveform

Another very simple method is to draw the audio signal as a waveform in a time – amplitude coordinate system. With this representation one can follow the audio signal point by point, but without any recognizable parallels between the heard sound and its visualization. This representation has one interesting feature, that is, if the resolution in the time domain is that small, that the audio wave itself cannot be followed any more (Figure 6.6), then the drawn graph is in some point an equivalence of the power of the audio signal again, which means, that loud and silent passages can easily be detected. Additionally the time information is shown.

When a cursor is provided, pointing to the appropriate part of the audio wave, the user can easily recognize which wave pattern belongs to which audible tag. This kind of graphical representation is used in most editors for audio data, and therefore has the advantage of being well known by most users, therefore providing an intuitive representation of the audio data.

6.2.2 Inline Links

Up to now only text documents and VRML documents support the notion of inline links. Inline images in text data are pictures, which are directly drawn into the hypertext browser, becoming part of the hypertext document for the user, but effectively are stored in a different file. Inline links are links, whose referenced document should be presented in line (on a particular place in the text data). If we extend this concept we come to two different notions for inline links in and inline links pointing to audio.

Inline Audio Links in Text Documents

Text documents provide so-called inline links, where images are directly loaded into the text browser. An audio file being inlined means, that the audio file is played in the background, when the text data is displayed. An audio file that can be inlined should have the following properties:

- It should not contain anchors to important information, because inlined audio should not have any visual representation, and therefore the anchors cannot be seen or followed.
- The sound data should be loop-able, which means the audio file can be played over and over again in the text browser as background sound.

In order to support inline audio links in text documents, the audio player should be able to start in the background and loop the audio file to be played.

Automatic Links in Audio Documents

The Harmony Audio Viewer supports a new kind of link attribute for hypermedia links, very similar to the inline links in text documents, so-called “automatic” links. An automatic link in audio data means, that when the playing of the audio reaches the start of a link, the link is followed automatically. All sorts of links except links to audio itself can be declared as automatic links in audio.

Automatic links in audio do not present the data at the location, where the link was defined (which would be the case for inline links in text documents), but at the time where the automatic link is defined. This concept is due to the main property of audio data not being location but time. With automatic links in audio it is possible to present hypermedia documents in an time scheduled manner, which is similar to video, but is not a multimedia format by itself, but a scheme for presenting different multimedia documents.

Chapter 7

The Harmony Audio Player

This chapter describes the Harmony Audio Player in full detail. The first section describes Harmony and its document viewers. The second section describes the User interface of the Harmony Audio Player. Then details of the implementation are covered, including how the audio player handles different audio file formats and different audio API's.

7.1 HyperWave and Harmony

Harmony is the native HyperWave authoring tool for Unix/X11 platforms. It provides possibilities to browse HyperWave servers, to edit hypermedia documents and to define hyperlinks between documents. All the advantages of the HyperWave system can be accessed through Harmony.

7.1.1 The Harmony Architecture

Harmony is made out of different parts (see Figure 7.1), which are all separate processes. The main process is the Harmony Session Manager. The other processes are the Harmony viewers. For each viewer there is a daemon, which waits for a connection from the Session Manager to fork the effective viewer. The viewers communicate with the Session Manager over TCP/IP connections.

It is the responsibility of the Harmony viewer to connect to the server and to receive the document. The Session Manager only provides the information, where the data can be found.

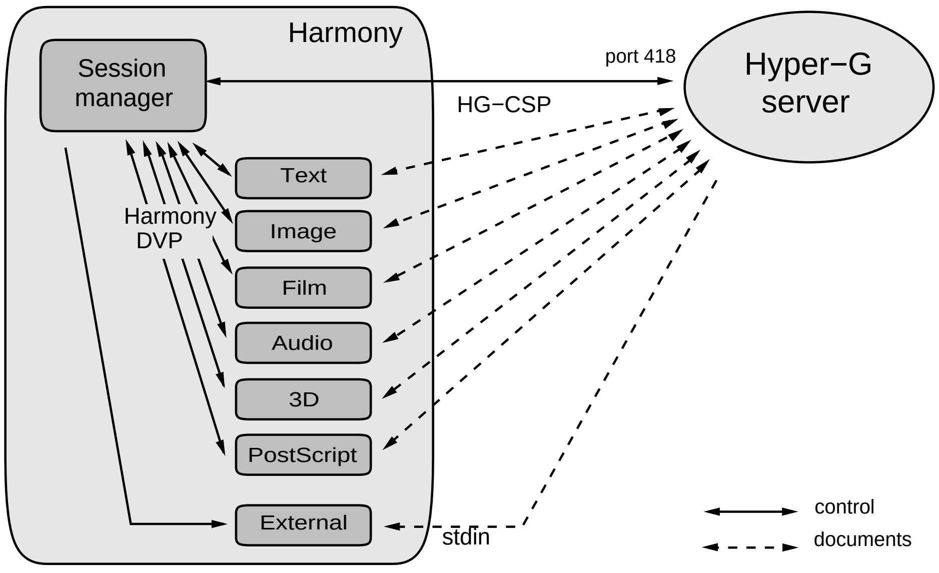


Figure 7.1: The Harmony Architecture.

7.1.2 The Harmony Session Manager

The Session Manager is the heart of Harmony [DH94]. In the Session Managers main window the structure of the data on the HyperWave system is displayed. This structure is a collection hierarchy. HyperWave collections are similar to directories in a file system. They can contain different other HyperWave objects. HyperWave objects are:

- Collections
- Hypertext documents
- Postscript documents
- Image documents
- Film documents
- Audio documents
- Virtual reality documents

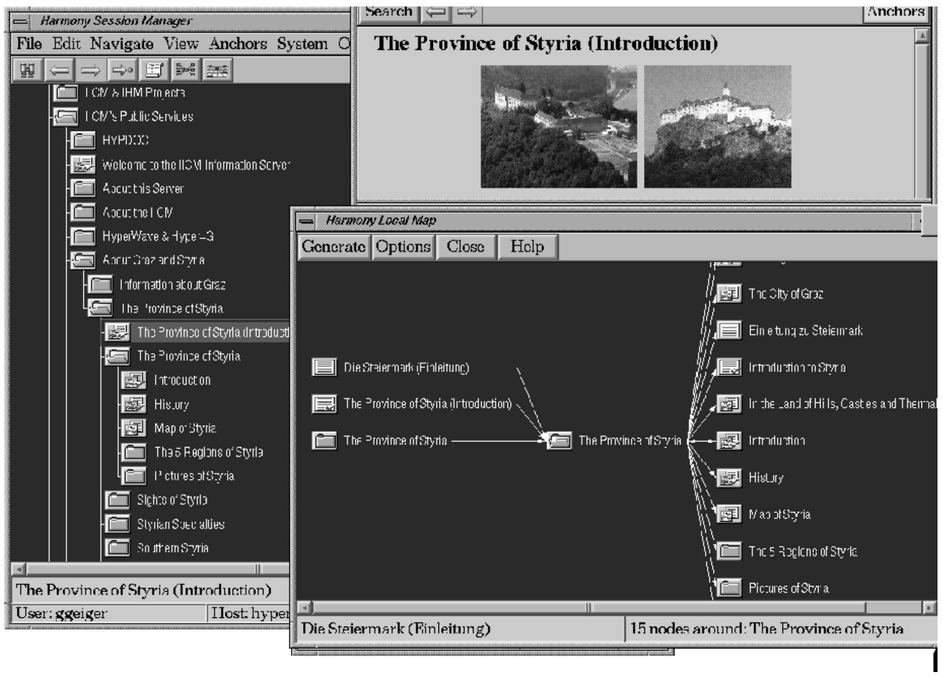


Figure 7.2: The Harmony Session Manager.

- Clusters
- FTP documents
- Telnet connections
- Generic documents
- Anchors (are not displayed in collections)

The root node of a collection tree is the Hyper Root. The Hyper Root cannot be a part of other collections. It's children are the root collections of the different HyperWave servers. The Session Manager executes the following tasks:

- HyperWave searches are started in the Session Manager.
- The Session Manager provides the possibility to insert documents into the HyperWave system.

- User management.
- Permission control.
- Control of the different viewers.
- Other kinds of data visualization.
  - Besides the tree structure of the collections it is also possible to display a “local map” (see Figure 7.2). A local map shows all the links from a document and their destination as well as the documents that point to the current document.
  - The information landscape provides a three dimensional view of object dependencies. It is possible to walk virtually through the information nodes.
- Help in navigating through the HyperWave collection hierarchy.

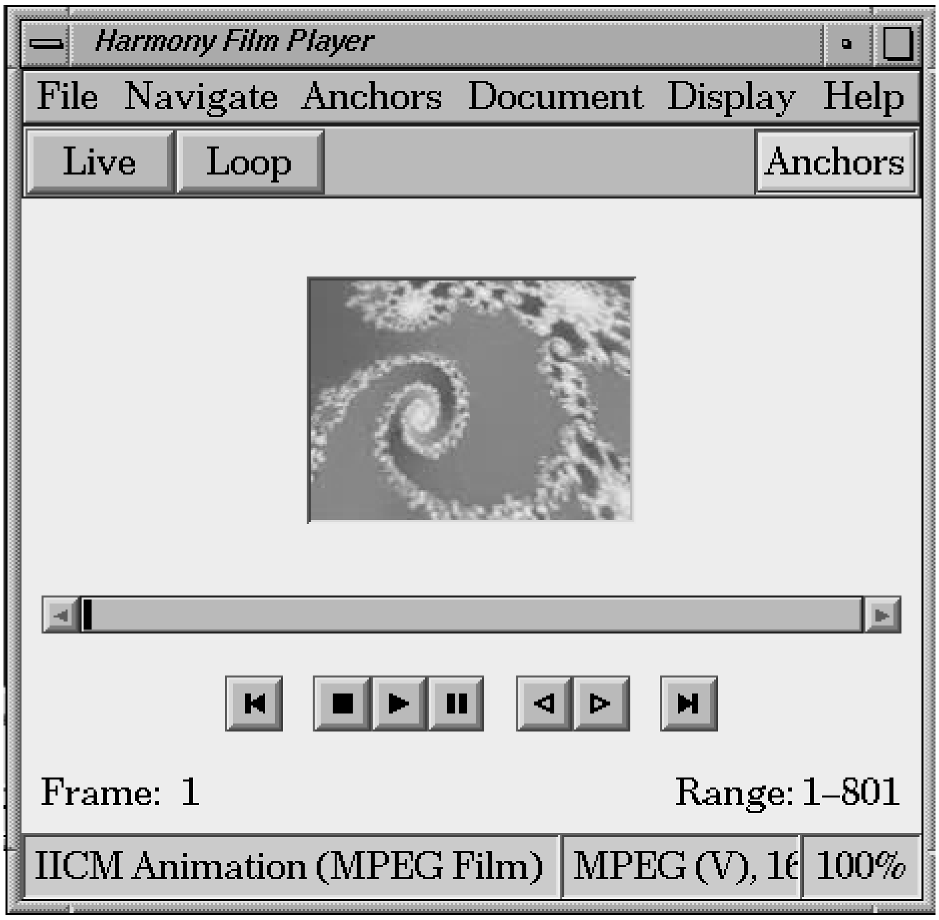
When clicking on an icon in the Session Manager, the specific document viewer is started and displays the document. Multiple instances of document viewers can be started.

Clusters are collections of documents, which are displayed together, forming a multimedia document.

7.1.3 Harmony Document Viewers

In the Harmony system document viewers are stand alone programs, that communicate with the Session Manager in order to send commands (e.g. following an anchor) or receive commands (displaying a specific document). Each document type has its own viewer. The list of viewers includes:

- The Text Viewer
- The Postscript Viewer
- The Image Viewer
- The Film Player
- The Audio Player



Frame: 1 Range: 1–801

IICM Animation (MPEG Film) | MPEG (V), 16 | 100%

Figure 7.3: The Harmony Film Player.

- The 3D-Scene Viewer

Some of these viewers are equipped with an editing tool. The Film Player is in some respect very similar to the Audio Player. Therefore I will describe the Film Player in more detail.

The Harmony Film Viewer

The Harmony Film Player (Figure 7.3) is started by the Session Manager when a film document should be displayed. The destination anchor of the document is marked in the scrollbar beneath the film display window. The toolbar provides different buttons for changing the behavior of the player. When the “Loop” button is pressed, the destination anchor is played in a loop. The “Live” button determines if the film should be played during loading (possibly with some delays). This feature is very hard to implement with audio, because a delay in audio would be very disturbing. The film player has seven buttons to control the position and state of the film being played. The play, stop and pause buttons (marked with common play,stop and pause icons) are used to start the film and stop the film. Two buttons are provided for stepping forward and backward through the film frame by frame. The first and last button in this row are for going immediately to the beginning or to the end of the film.

The Harmony Film Player offers the possibility to define links in movie sequences. These links have a time limit, just as links in the audio player. Additionally they have a spatial component like links in the image viewer. Link positions in the Film Player are interpolated from start position to the end position of the link, making it possible for links to follow an object in the film sequence. The links can be shaped as circles, ellipses, rectangles or closed polygons.

7.2 User Interface of the Harmony Audio Player

Figure 7.4 shows the Audio Harmony Player as it appears on the screen. I will describe the different buttons and menu entries as a whole.

7.2.1 The File Menu

File - Load

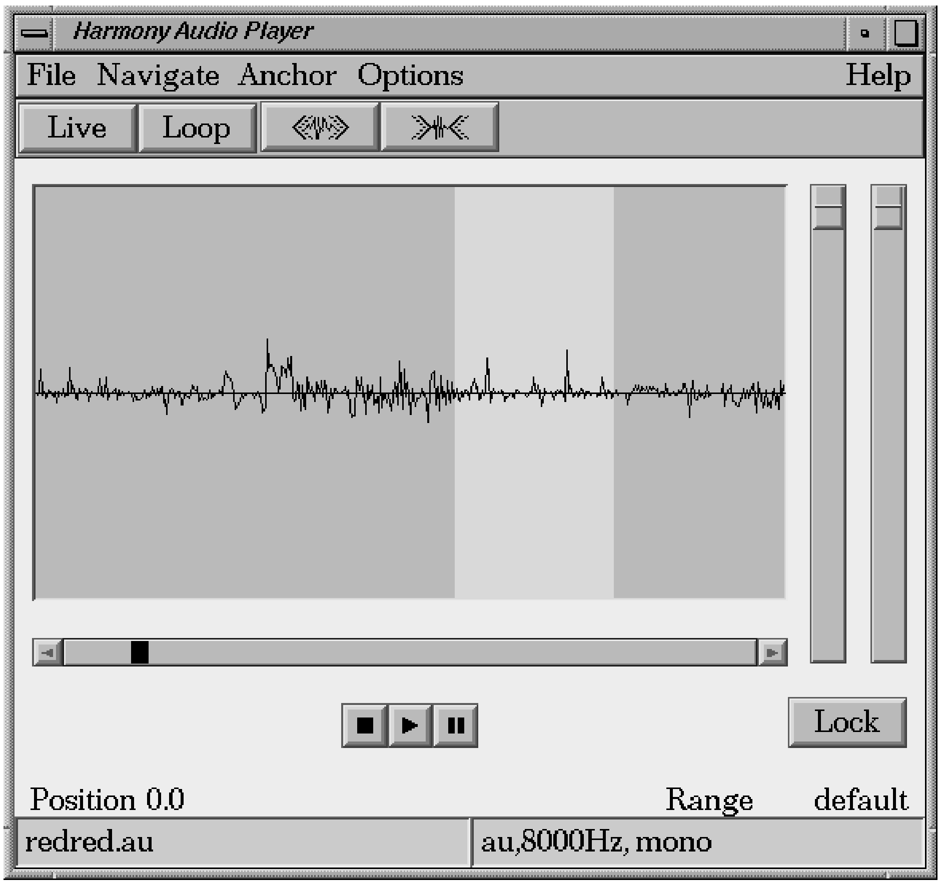
With the file menu load option it is possible to load a local file into the Harmony Audio Player. The file formats supported are the ones mentioned above.

File - Save

With the save option the current file can be saved.

File - Exit

The Exit entry quits the Harmony Audio Player.



Position 0.0

Range default

redred.au

au,8000Hz, mono

Figure 7.4: The Harmony Audio Player.

7.2.2 The Anchor Menu

Anchors - Next

The next anchor in the audio file is selected and highlighted. The audio data scrolls to this anchor.

Anchors - Previous

Scroll to the previous anchor in the current audio file and highlight its region.

Anchors - Follow

Follow the currently highlighted anchor.

Anchors - Define As Source

Define the currently marked region as source anchor for a hypermedia link.

Anchors - Define As Automatic Source

Define the currently marked region as intime source anchor for a hypermedia link.

Anchors - Define As Destination

Define the currently marked region as destination anchor for a hypermedia link.

7.2.3 Options

Options - Slider Lock

If the Player can output in Stereo, two sliders appear on the right side of the Harmony Audio Player window. The two sliders can be locked together with this menu entry.

Options - Loop

If this entry is chosen, then the audio data output is restarted if it reaches the end.

Options - Live

If the Live Option is chosen the Audio Player tries to play the audio data immediately while loading.

7.2.4 The Toolbar

Zoom In and Zoom Out Buttons

The zoom in and zoom out buttons allow the user to choose how much of the audio data is presented in the waveform window.

Other Toolbar Buttons

The other toolbar buttons correspond to their appropriate Options menu entry.

7.2.5 The Wave Widget and the Scrollbar

The wave widget is the most sophisticated part of the Audio Player's Interface. It offers the possibility to mark a region, by clicking with the left mouse button in the waveform window and dragging the mouse in one direction while holding down the left mouse button. With a double click on an anchor region (green marked regions) it is possible to follow a link.

The thickness of the scrollbar shows which portion of the audio data is currently shown in the waveform window. By clicking with the left mouse button on the thumb, the portion of the wave shown can be changed. The start position for audio playback is always the left side of the currently marked region, or the whole audio file, if no region is marked. On the left and on the right side of the scrollbar there are two small arrows, with which the position of the scrollbar can be changed.

7.2.6 The Play, Stop and Pause Buttons

When pressing the play button, the marked region of the audio file is played. If nothing is marked,then the whole audio file is played.

When pressing the stop button, the audio output is stopped and the start position is reset to position zero.

With the pause button, the output of audio data can be paused. When clicking on the play or pause button after a pause the audio output is continued at the position where it paused.

7.3 Configuring the Harmony Audio Player

The Harmony Audio Player can be configured via the X-Resource Database. The following X-Resources apply to the player:

***Audio.use_nas** The players for Linux and SGI's provide a stand alone audio interface. If the NAS system should be used, then this resource should be set to true. Valid values: true, false.

***Audio.stand_alone** If this resource is true, the audio player can be used without Harmony. Valid values: true, false.

***Audio.showwave** Toggles if the waveform widget is shown. Valid values: true, false.

***Audio.sliderlock** the volume sliders can be locked together. Valid values: true, false.

***Audio.loop** determines if the audio data is played in a loop . Valid values: true, false.

***Audio.live** determines if the player tries to play the audio data while it is being loaded. Valid values: true, false.

***Audio.tmpDir** the temporary directory for audio data. Valid values: string.

***Audio.audioserver** the audio server to which the NAS system should connect. This property can additionally be specified with the AUDIOSERVER environment variable. The X-resource overwrites the environment variable. Valid values: The audio server string consists of “hostname:displaynumber”. It defaults to localhost:0.

7.4 The Software Architecture of the Harmony Audio Player

7.4.1 The InterViews C++ Library

The Harmony Audio Player is written in C++. It uses the InterViews [Mar92] library for building the Graphical User Interface (GUI) and different other tasks. The InterViews library is a free C++ GUI library written at the Stanford University and was ported to most Unix platforms. It provides an extensible framework for building and using GUI classes in an X-Windows environment. Some of the classes of the Harmony Audio Player are directly related to InterViews classes and extend their functionality.

7.4.2 Data Representation

In order to fill the gap between the audio hardware possibilities and the format of the stored data, it is necessary to convert the data sent from the HyperWave server. This conversion can be done while playing the data, or while receiving the data. If it is done while playing the data, we have the advantage, that the data is stored locally in the same format as remotely, so we could easily archive it for later use or redistribution. Data conversion while receiving the data has the advantage, that the time wasted during transfer of data with a slow connection can be filled with a possibly complicated data conversion routine (MPEG). If we suppose, that we can do our data conversion in real time, there's no need to use the transmission delay for other computations. At the moment only fast systems can do MPEG encoding in real time, so it would be nice if data conversion could be done while loading and while playing the data. So we have three kinds of data representation:

1. Audio Documents on the HyperWave server.
2. Internal audio data representation of the audio player.
3. Audio hardware data representation.

The conversion of data from the internal form to the audio hardware specific form should be done while playing. The conversion of the data from the format on the HyperWave Server to the internal format should be done while transferring the data.

Internal Data Representation

The form of the internal data representation should fulfill the following criteria:

- No loss in quality should arise from the conversion.
- The data should be easy to handle by the program.
- The amount of memory to store the data should not be too large.

The SOX package includes an audio file library which makes it possible to read most of the known file formats. I decided to use this library, because of its independence from other audio API's. The internal audio format which is provided by this library is 32 bit integer. All audio files are converted to stereo while reading.

The data representation of SOX fulfills the first point of the above requirements. The fact that the format is an integer format, makes it possible to convert the different file formats (8 bit, 16 bit) into the internal format very quickly by just shifting them.

Converting all the samples into 32 bit integers and mapping to stereo makes it easy to handle the audio data, because the only difference between the audio formats is now the sample rate.

It is true that this method is not the best solution for the third point, but it fulfills the necessary requirements, and the fact that SOX uses the same format makes it easy to implement.

Audio Hardware Data Representation

The data representation in the audio hardware can differ in the following points:

- number of bits used
- number of channels
- sample rate(s)
- encoding (μ-law)
- endianess

To stay independent from the audio API, all of this data conversion should be done in the application. The uniform structure (a buffer of 32 bit integers) in the internal data representation helps with this task. All that is needed is a routine, which transforms the internal data representation to the hardware data representation.

7.4.3 File Interface

This section describes the Audiofile object (the Audiofile class, not the library with the same name), which provides access to the different sound file formats, and uses the SOX library. The Audiofile class supports at the moment only read access to audio files.

The Audiofile class has the following interface:

```
class Audiofile {
      Audiofile(fstream&,const char* = NULL);
      int read(long*,int num);
      int seek_pos(long pos);
      int fd_read();
      char* description();
      long length();
      long sr();
      int channels();
      int size();
      int is_ok();
};
```

The class is constructed with an already opened fstream object. The second parameter is the sound file format. At the moment the following audio file formats are supported: .au (.snd), .voc, .wav, .aiff. It is possible to construct the Audiofile class instead of a file with a socket or pipe too, although some calls like seek_pos() and length() will not succeed with this kind of object.

If the second parameter is not specified, the class tries to guess what kind of audio file format it has to deal with. A short description of the class methods:

- `read()` reads num frames into the buffer specified as the first parameter of the method. It is important to know that the buffer is always filled with stereo samples, even if the source file was mono.

- `seek_pos()` makes it possible in seek-able streams to position the file pointer at the frame number pos.

- `fd_read()` returns the file descriptor for use in the select call.
- `description()` returns a string which describes the format of the audio data.
- `length()` returns the length of the file in frames.
- `sr()` returns the sample rate of the stored data.
- `channels()` returns the number of channels of the data in the file.
- `size()` returns the size in bytes of the single samples.
- `is_ok()` returns true if a valid Audiofile object was created, false otherwise.

The concept of this class is similar to Broadway's client side file object. It allows to access audio data from different audio file formats transparently. All kinds of conversion can be done in this class.

7.4.4 Audio Interface

The interface to the audio hardware is represented by a class derived from the base class for all audio interfaces.

```
class Audio_Base {
      Audio_Base(int sr);
      virtual int set_samplerate(int sr);
      virtual int is_ok();
      virtual int fd_read();
      virtual int fd_write();
      virtual int process_audio(long* buf,int len);
      virtual void setvolume(int n,float r);
      void* convert(long*,int*);
      int get_stereo();
      virtual int open(int sr);
      virtual int close();
      virtual int stop();
```

```
        virtual int start();
        int get_delaysize();
};
```

The only parameter used by the constructor is the sample rate, because by default the class tries to output the audio data in stereo, 16 bit quality. If this is not possible it tries to get the best quality it can.

Most of the methods from Audio_Base are to be overloaded from a class which implements the real audio functionality. There is only one method which is the same for all audio classes. This is the `convert(long*,int*)` method. The first parameter points to a buffer with the sample-values. Their format is 32 bit (long integer) and stereo. This is the format we get out of the Audiofile class when reading. The second parameter is a pointer to the number of samples in this buffer.

The convert method converts the input buffer into the format of the audio hardware. This format is not known in the base class, it should be specified in the appropriate derived class of Audio_Base.

Other methods of the base class include:

- `int is_ok()` returns zero if the audio setup did not succeed

Derived Classes

Most of the classes of Audio_Base do not have any functionality. Here is a description of what the different methods of a class which is derived from the Audio_Base class should do:

- `Audio_Base(int sr)` the overloaded constructor sets up the audio hardware with the sample rate as parameter

- `int set_samplerate(int sr)` with this method the sample rate of an already existing audio class can be changed to a different value. The return value is the actually used sample rate

- `int fd_read()` returns the file-descriptor for a select call when reading from the audio hardware

- `int fd_write()` returns the file-descriptor when writing to the audio hardware

- `int process_audio(long* buf,int len)` write (read) the specified (len) amount of data to the audio hardware. Again len is the number of frames (stereo samples) and the buffer is in the internal data format (stereo, 32 bit signed integer samples) . This method then calls the convert method to put the input data into the correct format.
- `void setvolume(int n,float r)` sets the volume of the audio hardware. n is the channel number (0,1) r is the volume value between 0.0 and 1.0
- `int get_stereo()` returns non zero if the audio hardware can do stereo output
- `int open(int sr)` open the audio hardware with the specified sample rate
- `int close()` close the audio hardware.
- `int stop()`, `int start()` the need of these functions depends highly on the audio API used. They just start the audio output and stop it. They have to be called immediately before starting audio output and after audio output is finished.
- `int get_delaysize()` returns the delay-size in frames introduced through audio buffering. Needed for synchronizing audio and graphics.

The Audio Class

The Audio class is derived from Audio_Base and is an interface to the platform specific audio hardware. This class exists for Sun, SGI and Linux and accesses the vendor specific audio API directly.

The NasAudio Class

On platforms supporting the NAS API this class is the interface to the audio output with NAS. The NasAudio constructor optionally accepts a second parameter which is a char* pointing to a string containing the name of the NAS server and port.

Being derived from Audio_Base, the NasAudio class provides the same functionality as the Audio class does, and can be used interchangeably with the classes derived from Audio_Base.

The NoAudio Class

The NoAudio class is also derived from Audio_Base and offers no functionality. This is the class which is used if audio setup failed.

7.4.5 Audio Output - AudioWriter class

The AudioWriter class gives the possibility to play audio from a file.

The InterViews IOHandler

The IOHandler class is a virtual base class provided by InterViews in order to add a method (or class) for processing into the main dispatch loop of the InterViews library.

A toolkit like InterViews does the following: It builds up a structure of different objects, and puts them into an initial state. These objects can be graphic objects, like buttons, sliders, or they can be any other object, without any graphical representation.

The toolkit waits for user interaction in a dispatch loop. User interactions are called events. For each event it is possible to add an event-handler, which in effect changes the state of some object. This event-handler does some processing when the event occurs. For example it changes the look of a button when the mouse pointer moves over it. There are different events, mainly from the mouse and from the keyboard.

In addition to event-processing, the InterViews toolkit provides the possibility to react to events which occur on file descriptors. The toolkit uses the select call on different file descriptors and calls the appropriate IO handler for it. These file descriptors and their IO handlers can be linked into the InterViews dispatch class.

The IOHandler class provides the interface for this IO handlers. Each class that wants to use the possibility of reacting to an IO event has to be derived from the IOHandler class.

The AudioWriter class

The only thing that is needed in order to do audio output is to look at the file-descriptor provided from the Audio class, and if it is writable, then read the next block of samples from the Audiofile class and write it to the audio hardware.

The AudioWriter class is derived from IOHandler and has a Audiofile and a class derived from Audio_Base as members.

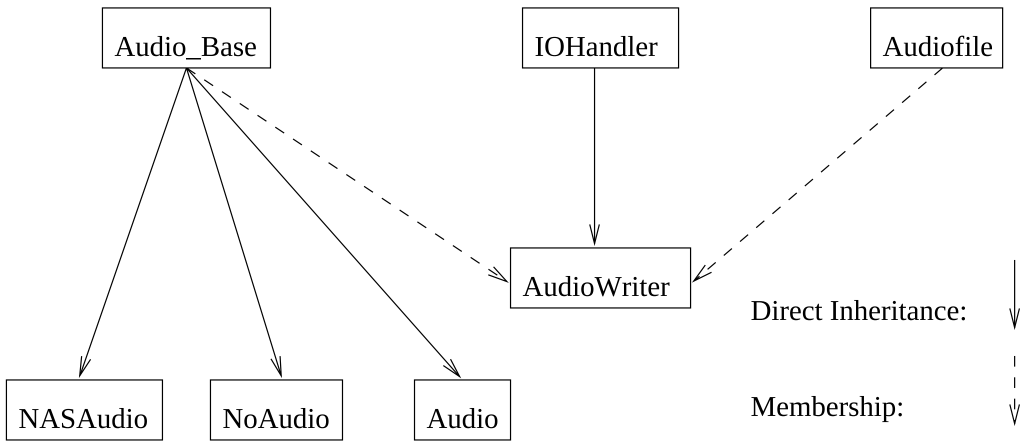


Figure 7.5: The AudioWriter class inheritance scheme.

```
class AudioWriter:  IOHandler {
public:
      Audio_Base* audio;
      Audiofile* file;
      AudioWriter(Audio_Base* base,Audiofile*);
      void set_player(Player* p);
      int inputReady(int fd);
      int outputReady(int fd);
      int is_playing();
      int load_wave(long from = 0,long to = -1);
      int play(long from,long to);
      int stop();
      void volume(int num,float r);
      long position();
      int is_ok();
};
```

This class is closely related to the Player class, described later. In some of the methods the Player class is called back from the AudioWriter. The procedures called are Player::processing_out() and Player::playingDone().

With these callbacks the graphical output that has to be done while playing can be triggered.

The methods of AudioWriter are the following:

- `AudioWriter(Audio_Base* base,Audiofile*)` the constructor needs pointers to an Audio_Base and an Audiofile object.
- `void set_player(Player* p)` this method provides the possibility to add a player, which will be called back through processing_out and processing_done.
- `int inputReady(int fd)` this method is called from the Interviews dispatcher when the file descriptor associated to the IOHandler is writable.
- `int outputReady(int fd)` this is called when the file descriptor is readable.
- `int is_playing()` returns nonzero if the class is in playing state.
- `int load_wave(long from = 0,long to = -1)` do not do audio output, but load the audio data from the file (parsing).
- `int play(long from,long to)` puts the class into playing state.
- `int stop()` stops the audio output.
- `void setvolume(int num,float r)` sets the volume of the audio hardware. n is the channel number (0,1) r is the volume value between 0.0 and 1.0.
- `long position()` returns the position of the currently played frame.
- `int is_ok()` returns non zero if the state of the class is ok.

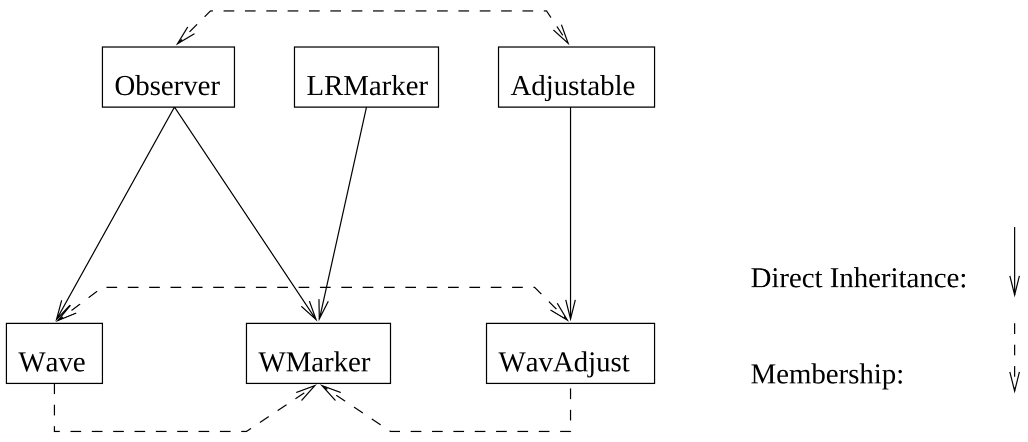


Figure 7.6: The Wave class inheritance scheme.

7.4.6 Graphical Data Representation

The graphical data representation should show the waveform of one audio channel. The wave widget is realized with the InterViews toolkit. The state of the graphical output has an internal data representation. This data is encapsulated in a so-called Adjustable. All widgets which have to deal with this internal data representation are derived from the Observer class. Figure 7.6 shows the inheritance scheme of the graphical output classes. The membership relations mean, that one class knows of the other class and can send messages to it.

The relation between the base classes is defined by the InterViews toolkit as follows. Each Observer object can be attached to an Adjustable object. In this case the Adjustable object holds the data and the Observer object does the graphical representation of the data stored in the Adjustable. Each Observer object has the possibility of changing the state of the Adjustable object. If the Adjustable object is changed, all the attached Observer objects are notified.

One can look at an Observable as just some kind of (graphical) representation of the data stored in the Adjustable.

The WavAdjust Class

In order to mark audio data, scroll forward and backward, zoom in and out, a specialized adjustable value is needed, which represents the current state of the Wave widget. This adjustable class is called WavAdjust and has the following interface:

```
class WavAdjust:  public Adjustable
{ public:
      WavAdjust (// constructor
                 float lower, // lower bound
                 float upper, // upper bound
                 float initial, // initial value
                 float small = 0.05, // small (line) scroll
                 float large = 0.25, // large (page) scroll
                 float ppu = 2000.0
                 );

      void smallScroll (Coord f);
      void largeScroll (Coord f);

      virtual Coord lower (DimensionName) const;
      virtual Coord upper (DimensionName) const;
      virtual Coord length (DimensionName) const;
      virtual Coord cur_lower (DimensionName) const;
      virtual Coord cur_upper (DimensionName) const;
      virtual Coord cur_length (DimensionName) const;
      Coord upper_mark(DimensionName) const;
      Coord lower_mark(DimensionName) const;
      Coord length_mark(DimensionName) const;
      Coord cur_value(DimensionName) const;
      virtual void scroll_to (DimensionName, Coord);
      void scroll_val(Coord);
      virtual void scale_to(DimensionName, float fraction_visible);
      virtual void zoom_to(float magnification);
      void set_cur_lower(Coord l);
      void set_cur_upper(Coord l);
      void set_lower(Coord l);
      void set_upper(Coord l);
      void set_lower_mark(Coord l);
      void set_upper_mark(Coord l);
}
```

The WavAdjust class represents different ranges of values. Most of the time Adjustables are used for the representation of objects that are scroll-able. Therefore this class provides methods for accessing a lower, and upper bound. These are the absolute bounds. Within these bounds a small range (the visible window of a scroll-able object) can be manipulated. It can be moved or it can be made larger and smaller.

In addition to the native Adjustable properties the WavAdjust provides properties which are used for the representation of a waveform:

- The `cur_value` represents the position of a cursor. This is the value which indicates the current playing position of the audio player.
- An additional range, which is used for editing purposes. This range indicates which part of the audio data is marked. There is only one adjustable marker for the Wave widget.

Most of the methods of the WavAdjust class are for manipulating these different range values. The main purpose for this WavAdjust object is to notify all objects which do some kind of graphical representation of its contents when some other object changes one of the values. Therefor it is possible to provide several means for scrolling a value, and all other objects which represent this value can update their representation.

For performance reasons the WavAdjust class additionally provides for every Observer object a means of determining what kind of change took place, when the Observer object is notified. Therefore the different Observer objects can decide if they should react to the change that was made.

The WMarker and Wavemarker Classes

These two classes are closely related and are responsible for the marker widget, which manipulates and shows the marked area within a wave object. The classes are implemented as Monoglyph and can be wrapped over a wave widget, adding the possibility to mark and display a marked area (WMarker). The WaveMaker is an Interviews InputHandler class. With this class it is possible to respond to user interaction such as mouse-clicks and keyboard-entries. The WaveMarker class provides the user input functionality for the Wave widget.
The user actions are:

- marking an area
- following an anchor

The WMarker object is responsible for drawing all the anchors defined in an AnchorList object. The AnchorList object just contains a list of HyperWave anchors, and is not discussed in this document. It has the same form as the AnchorList objects in the other Harmony viewers. All other user interaction is provided through the toolbar and the menus following the conventions of the other Harmony viewers.

```
class WMarker:  public LRMarker, public Observer {
public:
      WMarker(Wave* g,WavAdjust* adj,const Color* overlay,
              const Color* underlay,AnchorList* a=nil) :
              LRMarker((Glyph*) g,overlay,underlay);
      void detach();
      void attach();
      void stop_scrolling();
      void selectAnchor(float f);
      void mark(Coord l,Coord r);
      void mark_to(Coord m);
}


class WaveMarker:  public InputHandler {
public:
      WaveMarker(Glyph* g,Style* s);
      void press(const Event& e);
      void drag(const Event& e);
      void double_click(const Event& e);
      void release(const Event& e);
};
```

The Wave Class

The Wave class is responsible for the output of the graphical representation of the audio data. For this purpose it uses an array of values, which represent the audio data which is to be drawn. It assumes that the values in the array have the same range as a signed 32- bit integer variable (from -2.147483e+09 to +2.147483e+09). This array is drawn into the widget's body using properties and ranges defined in the WavAdjust object.

When playing the audio data, the Wave widget is just partially redrawn. The Wave widget is scroll-able, zoom-able and draws a cursor which is moved synchronously with the audio played. Here's the interface of the Wave widget:

```
class Wave :  public Glyph, public Observer {
public:
      Wave(Style* style,WavAdjust* adj,AnchorList* l);
      boolean set_buffer(Coord* b,long len);
      void set_buflen(long len);
      void show_unit(float u);
      void show_fraction(float f);
      void zoom_in();
      void zoom_out();
      float get_zoom();
      void scroll_value(float v);
      void scroll(Coord f);
      void scroll_forward();
      void scroll_backward();
      void restrict_redraw();
      void redraw();
      void detach();
};
```

- `set_buffer()` tells the wave widget where the data buffer to draw is located.
- `set_buflen()` sets the length of this buffer.
- `show_unit` tells the widget about the initial zoom factor. A unit is an abstract value, which normally defaults to one second.
- `zoom_in()` and `zoom_out()` change the zoom factor by two or one half.
- `get_zoom()` returns the current zoom value.

7.4.7 The Player Class and the HgAudioPlayer Class

The main class of the Harmony Audio Player is divided into two parts. The HgAudioPlayer class, which is responsible for loading documents from the HyperWave server and which is derived from the Player class. The Player class is responsible for the functionality of the Harmony Audio Player.

The Player Class

```
class Player :  public HgViewer {
public:
      Player(HgViewerManager* manager,
             const char* server = nil,
             const char* fname = nil,
             boolean alone = false);
      void processing_done();
      void set_position(long p);
      void processing_out(long* buffer,long num);
      void check_inlines();
      virtual void processing_in(char* buffer,long num)
      void openaudio(istream&,long len = 0,const char* = nil);
      void closeaudio();
      void makeWindow();
      void remakeWindow();
      Glyph* menu();
      Glyph* compose();
      void play();
      void stop();
      void pause();
      void quit();

      void previousAnchor();
      void nextAnchor();
      void makeSourceAnchor();
      void makeSourceInlineAnchor();
      void makeDestinationAnchor();
      void makeDefaultDestinationAnchor();
      void deleteAnchor();
      void followAnchor();

      void toggleMono();
      int checkMark();

      void back();
      void forward();
      void history();
      void hold();
      void zoom_in();
      void zoom_out();
      void toggleLoop();
      void toggleLive();
      void toggleLock();
      int getLoop();
      int getLive();
      int getLock();
      boolean loading();
      boolean standalone();
      openFile(char *);
      saveAs(char *);
      Window* window();
      Object* document();
      HgViewerManager* manager();
      void error(const char*);
      void fatalError(const char*);
      int port() const;
      void childExited(int, int);
      void childSignaled(int, int);
      void childStopped(int, int);
      void cleanup();
};
```

Most of the functionality of the Harmony Audio Player is done by the Player class. It provides callback functions for most of the menus and buttons in the player. These methods change the internal state of the player and make it behave differently.

Examples of callback functions are:

- Anchor menu callbacks
  - `void previousAnchor()` go to the previous anchor in the document.
  - `void nextAnchor()` got to the next anchor.
  - `void makeSourceAnchor()` make the marked region a source anchor.
  - `void makeSourceInlineAnchor()` make the marked region a source intime anchor.
  - `void makeDestinationAnchor()` make the marked region a destination anchor, if no region is marked use the whole document as destination.
  - `void deleteAnchor()` delete the currently selected anchor.
  - `void followAnchor()` follow the currently selected anchor.
- History callbacks
  - `void back()` go back in history.
  - `void forward()` go forward in history
  - `void history()` show history
  - `void hold()`
- `play()`, `stop()`, and `pause()` are called by the play, stop and pause buttons.
- `toggleLoop()`, `toggleLive()` and `toggleLock()` change the Loop, Live and Lock (locking of the volume sliders) state.

The Player class is responsible for building the graphical user interface. Methods dealing with GUI construction are `void makeWindow()`, `void remakeWindow()`, `Glyph* menu()` and `Glyph* compose()`.

The HgAudioPlayer Class

```
class HgAudioPlayer :  public Player {
      HgAudioPlayer(HgViewerManager* manager,
                    const char* aserver=nil,
                    const char* fname = nil,
                    boolean alone = false);
      void loadingDone(int);
      void load(const char *, const char *,const char * = 0);
      int loadConnect(const char* doc, const char* anch,
                    const char* info);
      void do_load(const char* doc, const char* anch,
                    const char* info,boolean connect);
      void browse(const char *);
      void terminate();
      void processing_in(char* buffer,long num);
      int port() const;
      int open_tmp_file();
};
```

The HgAudioPlayer class implements the the general HgViewer virtual methods `load()`, `loadConnect()` and `browse`. the `processing_in()` method is called by the `Reader::read()` method. The HgAudioPlayer has a pointer to the Reader class as a member. The Reader class is implemented in the same way as the Reader classes of other Harmony viewers.

Chapter 8

Outlook

8.1 Auditory Icons

Links in the Harmony Audio Player are currently shown as marked areas in the waveform window. This is the visual representation of a link in audio. Additionally links in audio could have an audible representation. This representation should be easy recognizable to listener. In order to make audible marks at the beginning and at the end of a link, each link should have an opening and a closing sound. The opening and closing of a door, would be a good example. The opening and closing of a door is also inspired by the fact, that the link offers the possibility to “step into“ another document.

A second possibility of auditory icons is a sound during the time a link is valid. The disadvantage of this solution is, that the link sound can be very disturbing when link is valid for a longer time.

The implementation of auditory icons should be done in the AudioWriter class. The sound is stored in a buffer and played when triggered by the beginning or end of a link. One problem is that the auditory icons have to be adapted to the sample rate and format of the audio data loaded (this is not true when using NAS, because NAS can convert the data in the server).

8.2 Visual Icons

It is possible to select and follow the links either when the player is stopped, or when the player is active by clicking on the marked area. When the player is stopped, it is possible to scroll to the link position and follow the link, just like one would do when following a link in a text document. During playback it is not possible to scroll and therefore a link can only be followed when it is visible in the waveform window. A problem with the visualization of links in audio is that it is possible that links can overlap. Currently when clicking on an overlapped link in the waveform window, the link that starts later is followed.

It should be possible to provide visual link icons outside the waveform window, which allow to represent a hierarchy of overlapped links. The user then has the possibility to choose which link to follow. This icons should only be visible while the links are active (when the cursor is over a link). In this case it is a list of links which is displayed as click-able icons.

8.3 Data Compression, MPEG support

At the moment the Harmony Audio Player only supports the main audio file formats and only one compression schemes (μ-law). It would be desirable, especially for an audio application which has to download its data from sometimes slow Internet links, to add more support to different compression schemes. This can be done in two ways.

- add support for a new audio coding technique in the SOX library.
- add support for decoding on the fly.

The .wav and .aifc audio file formats support different compression schemes. These compression schemes could be implemented using the sound tools (SOX) library skeletons, or by adding the support into the wav.C and aiff.C files of the library.

When adding MPEG support it would be desirable to decode the data while it arrives from the HyperWave server. With this technique even slower machines would be able to play MPEG encoded audio files. On the other hand, if the link to the HyperWave server is rather fast, then the decoding while downloading can cause the download time to increase. Free MPEG decoding software is available. Including MPEG decoding in the Harmony Audio Player would be a matter of figuring out how these MPEG decoders work. If the decoding can be done with fixed sized buffers, then the integration of MPEG into the Harmony Audio Player would be possible.

8.4 Live Audio Streams

The problem with live audio streams is that they do not have a limited amount of data. So it is not possible to store the waveform of the audio data in the Wave widget. The Wave widget provides the possibility to implement live audio streams. In this case only a small part of the waveform is loaded into the Wave widget. This is possible because the buffer, where the wave data is stored is not a part of the Wave widget, and can be set by the controlling program.

As mentioned earlier the integration of Real Audio in Harmony is not possible (Although it would be possible to implement a function which starts the Real Audio Player as a plug-in). Different kinds of free Internet radio exist, but most of them are very simple and do not allow broadcasting as Real Audio does (Real Audio provides different relay stations, where the live data stream is distributed to different locations). Internet radio would be an interesting feature for the HyperWave system, because the different HyperWave servers, and their interconnections offer the possibilities to do this kind of broadcasting. However implementing Internet radio in HyperWave goes far beyond the scope of an Audio Player for Harmony.

Bibliography

[AIF89] Audio Interchange File Format, A Standard for Sampled Sound Files, Version 1.3. Electronic document available at `mitpress.mit.edu://pub/Computer-Music-Journal/Documents/SoundFiles/AIFF-C.ps.Z`, 1989.

[AIF92] Multimedia Programming Interface and Data Specification v1.0. Electronic document RIFFMCI:RTF available from `ftp.microsoft.com`, 1992.

[And96] Keith Andrews. *Browsing, Building, and Beholding Cyberspace.* PhD thesis, Graz University of Technology, 1996.

[Aud] Audiofile release notes. `ftp://crl.dec.com/pub/DEC/AF/ReleaseNotes.txt`.

[AVO75] Ronald W. Schafer Alan V. Oppenheimer. *Digital Signal Processing*. Prentice Hall, 1975.

[Bro] The Broadway Audio System. `http://www.x.org/consortium/audio/`.

[DH94] Dalitz and Heyer. *Hyper-G*. dpunkt-Verlag, 1994.

[ISO94] ISO. ISO-MPEG-1 Audio: A generic standard for coding of high-quality digital audio. *J. Audio Eng. Soc.*, 42(10), October 1994.

[Kro92] Ed Krol. *The Whole Internet*. O'Reilly & Associates, Inc, 1992.

[Mar] Mark Boyns boyns@sdsu.edu. Rplay. `ftp://sunsite.unc.edu/pub/linux/apps/sound/servers/rplay-3.2.0b5.README`. Mailinglist: rplay@sdsu.edu.

[Mar92] Mark A. Linton, Paul R. Calder, John A. Interrante, Steven Tang, John Vlissides. *InterViews Referenz Manual version 3.1*. Board of Trustees of the Leland Stanford Junior University, 1992.

[Mau96] Hermann Maurer, editor. *HyperWave: The Next Generation Web Solution*. Addison-Wesley, May 1996. `http://www.iicm.edu/hgbook`.

[MID] Standard midi-file format spec. 1.1. Electronic Document. `http://www.geocities.com/SiliconValley/2704/midi.html`.

[Moo93] K. Moore. RFC 1522 MIME (Multipurpose Internet Mail Extensions) Part Two Message Header, September 1993. Extensions for Non-ASCII Text.

[MPE] The MPEG Homepage. `http://www.cselt.stet.it/mpeg`.

[OII] Open Information Interchange Standards, Audio Interchange Standards. Electronic Document. `http://www2.echo.lu/oii/en/audio.html`.

[Rea] Real Audio. `http://www.realaudio.com`.

[vR] Guido van Rossum. Audio file formats FAQ. `ftp://ftp.cwi.nl/pub/audio/`.

[vR95] Guido van Rossum. Machine Tongues XVIII: A Childs garden of Sound File Formats. *Computer Music Journal*, Volume 19(Number 1), Spring 1995.

Appendix A

User Guide

A.1 Starting Up the Player

A.1.1 Starting from Harmony

The Harmony Audio Player is started by the Session Manager in order to display and play back an audio document. When starting up, the main window of the audio player is shown, the progress indicator at the bottom of the window shows that the player is in loading state. A bar in the second part of the progress indicator shows how much of the data is loaded. When loading is finished, the audio player parses the received data and constructs the waveform. After that, the player is ready.

Figure A.1 shows the appearance of the Harmony Audio Player after startup. The waveform uses most of the applications main window. There are different marked areas in the waveform. These areas are source anchors. The destination anchor is marked in the scrollbar beneath the waveform window. The progress indicator shows the name of the loaded document in the left part and the properties of the audio data loaded in the right part.

A.1.2 Starting Up the Stand Alone Audio Player

The Harmony Audio Player can be used stand alone. In this case the player is started form the command line with the command:

```
aplay filename
```

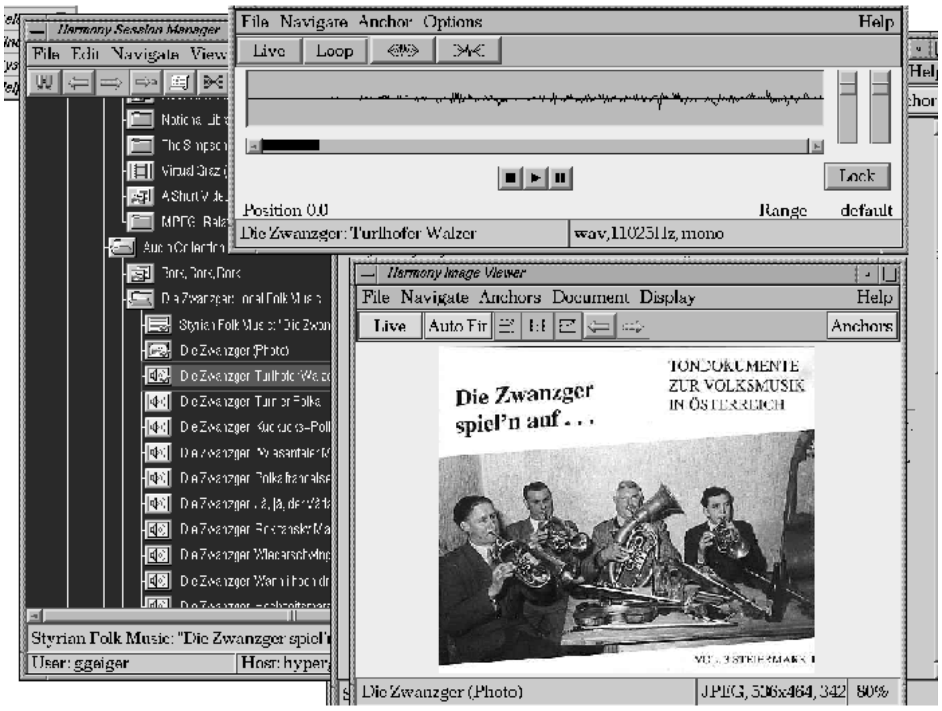


Figure A.1: The Harmony Audio Player after Startup with other Harmony tools.

The aplay command is just an alias for `haraudiod -s`. The general haraudiod command syntax is:

```
  haraudiod [-p portnumber] [-d audiohost:number]  [-s
[filename ]]
```
The parameters mean the following:

- `-p portnumber`, where portnumber is a number which identifies the connection to the harmony session manager. When specifying a portnumber, harmony has to be started with the command line parameter -ap portnumber. For normal operation this parameter is not used.

- `-s` puts the Audio Player into stand alone mode. This is the same as starting the player with the `aplay` command.

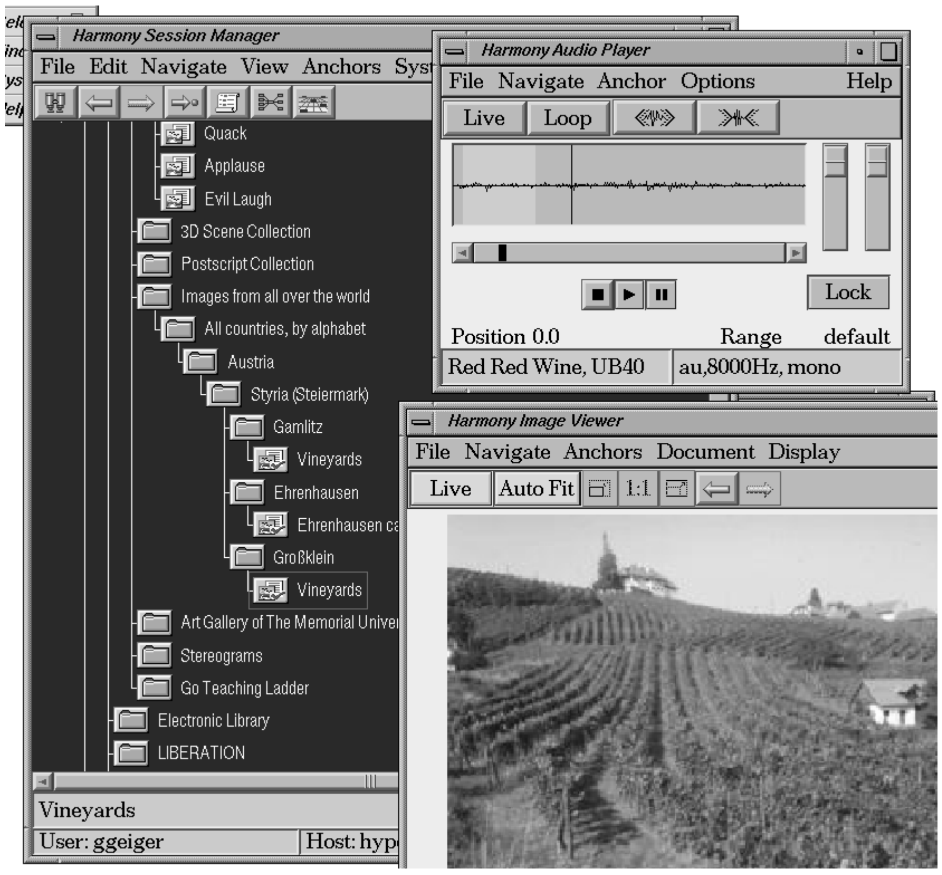
- `-d audiohost:number` is used with the NAS audio output option. It specifies the host and number (typically 0) where the NAS audio server is started. The audio server has to be started with the -aa option (e.g. au -aa) in order to allow access for all hosts.

- *`filename`* is the name of the audio file which should be loaded into the player when using the standalone mode.

When starting up the stand alone player all Harmony specific commands are disabled. The stand alone player can be used to verify the readability of audio files and for general playback of different kinds of audio files without converting to the machine specific audio file format and as a helper application for Web Browsers such as Netscape.

A.1.3 Loading and Saving Audio Files

If no audio file was given as parameter at the command line, it is possible to load the audio file from a local disk with the File pulldown menu entry “Load”. Additionally, it is possible to save a file with the “Save” entry. Load and Save work in the standalone mode and in the Harmony mode of the Player, therefore allowing files from the HyperWave server to be saved locally. The extension of the file is not important for the audio player, because the player detects the file format by reading the file header.



Vineyards

User: ggeiger Host: hyp

Position 0.0 Range default

Red Red Wine, UB40 au,8000Hz, mono

Figure A.2: The Harmony Audio Player while playing an audio file with automatic links.

A.2 Playing, Stopping and Scrolling

After the audio data is loaded the player either starts playing immediately or waits until the button with the play icon is pressed. This behavior depends on the X Resource settings for the “Live” property. The behavior can be changed with the Live button in the toolbar or the Live menu entry in the options pull down menu. When the play button is pressed, the player starts to play the destination area of the document. This is only done the first time when playing. Afterwards, the player plays either the whole audio document or a marked area (see Section A.4.2 Marking an Area). If the audio document contains intime anchors the appropriate viewers are started while playing and the linked documents are displayed (see Figure A.2).

When the player is finished playing it resets itself to the starting point. With the stop button audio output is stopped and the player is reset to the start position. With the pause button, the player stays at its position and audio output is stopped. The output starts at the same position if the play or pause buttons is pressed after a pause.

A.3 Following and Selecting Anchors

Anchors are selected by clicking once in the marked area in the Harmony Audio Player. When selected, the anchors change their color. It is possible to step through the whole list of anchors with the anchors pull down menu entries “Next” and “Previous” as well as with the TAB key on the keyboard.

When the RETURN key is hit the selected anchor is followed. Another possibility of following an anchor is double clicking with the mouse in an anchor area in the waveform window.

A.4 Inserting Anchors in Audio Documents

With the Harmony Audio Player, as with all Harmony viewers, it is possible to define links in audio files. The procedure for uploading audio documents and editing links are principally the same as in other Harmony viewers.

A.4.1 Inserting Audio Files in the HyperWave system

Insert the audio file with the session managers file - insert function. Supported file formats at the moment are:

- Microsoft WAV files
- Apple and SGI AIFF files
- Sun,NeXT .au and .snd files
- Creative’s .voc files

The type field of the insert panel should be “Audio”. In order to be sure that the audio file is in the correct format the audio player should be started by clicking on the newly created audio icon.

The audio file is loaded into the Harmony Audio Player. Now there’s the possibility to insert links into the audio data. There are two buttons on the toolbar, which make it possible to zoom the graphical representation of the audio data. Most of the time there’s a reasonable default resolution. If the audio data should be better visible it is also possible to resize the window, which will result in a larger waveform. Now after the play button is pressed the audio document should be heard. In the wave window a cursor moves over the waveform. This cursor points at the position in the audio file that is currently played. The part where the link should appear in the visual representation should then be identified. For some kinds of data this is very easy, for other kinds it is hard.

A.4.2 Marking an Area and Defining Links

A link region is marked by clicking and dragging with the left mouse button (see Figure A.3. If an area is marked, then the play button will only play the marked area. If the link is correct, the “Define as Source” item from the anchors menu is chosen. One can link any document (destination anchor) into the audio file. The procedure is the same as with the other harmony viewers. If the link should be an intime link, the menu entry “Define as Intime Source” should be used. The same procedure should be followed in order to define a destination anchor in audio. Instead of “Anchor - Define Source Anchor”, the “Anchor - Define Destination” has to be clicked on in order to define the marked area as a destination anchor. If the link is not correct its borders can be changed by clicking with the right mouse button into the wave widget. For fine tuning it is possible to zoom the waveform with the zoom buttons on the toolbar.

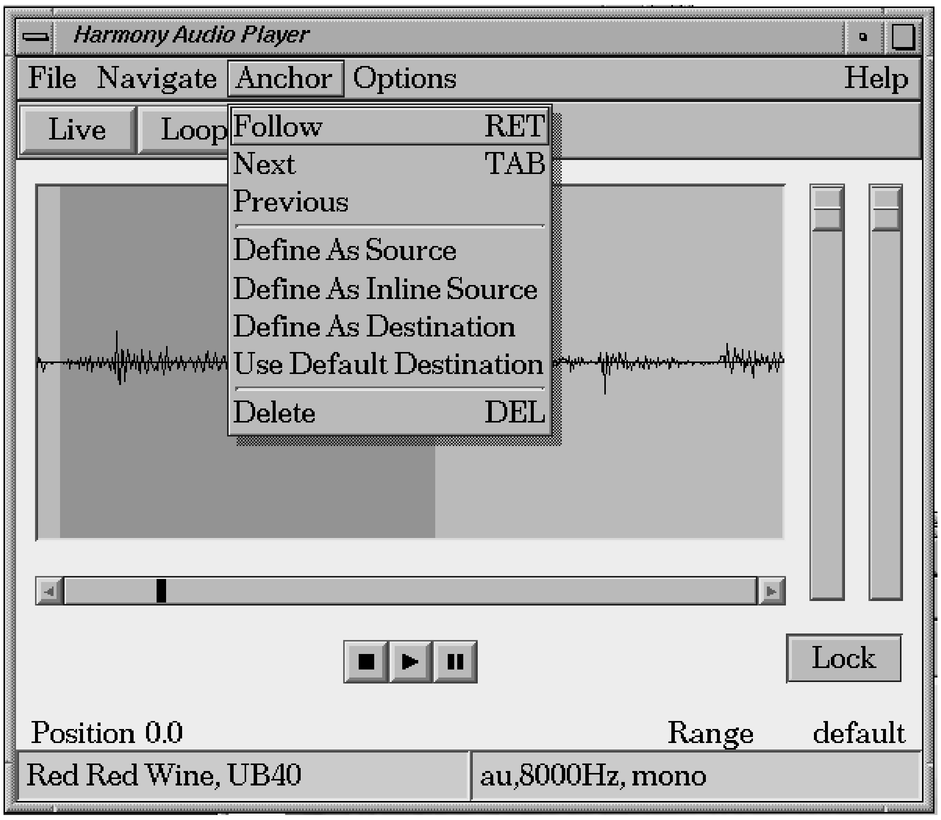


Figure A.3: Marking an area in the Harmony Audio Player.