Machine Tongues XVIII: A Child’s Garden of Sound File Formats

Stephen Travis Pope* and Guido van Rossum†
*_Computer Music Journal_, CNMAT
P. O. Box 9496
Berkeley, California, 94709 USA
stp@CNMAT.Berkeley.edu
†Multimedia Kernel Systems Group, CST
Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, P. O. Box 94079
Amsterdam, NL-1090 GB The Netherlands
Guido@CWI.nl

Practitioners of computer music often need to store sound as sampled digital data on their computer’s hard disks, digital audio tapes (DATs), or compact disks (CDs). The data formats used to represent digitally sampled sound determine the audio quality of the sound capture, the amount of disk or tape needed to store the data, and the ease with which application programs can be developed to manipulate sound data.

Sound file systems are software packages that provide standardized data structures for sampled sounds in order to enable real-time recording and playback, to ease the development of sound-processing software tools, and to increase the interoperability of related utility programs. Sound file systems differ in what kind of support they offer for compact and efficient sample storage on disk, real-time sound I/O, data interchange between programs, and sound manipulation and annotation. The components of such a system include the disk-based storage format, in-memory data structures, function libraries for processing sound data (the application programming interface or API), and some collection of end-user utility programs, for example to play and record sounds.

This article introduces a few of the many ways that sound data can be stored in computer files, and describes several of the file formats that are in common use for this purpose. This text is an expanded and edited version of a “frequently asked questions” (FAQ) document that is updated regularly by one of the authors (van Rossum). Extensive references are given here to printed and network-accessible machine-readable documentation and source code resources. The FAQ document is regularly posted to the USENET electronic news groups alt.binaries.sounds and comp.dsp for maximal coverage of people interested in digital audio, and to comp.answers, for easy reference. It is available by anonymous Internet file transfer from the server ftp.cwi.nl in the two files pub/audio/AudioFormats.{part1, part2}, and can also be found in the archives named rtfm.mit.edu:/pub/usenet/news.answers/audio-fmts/part[12], and in the *Computer Music Journal* ftp archives as mitpress.mit.edu:/pub/Computer-Music-Journal/Documents/SoundFiles/AudioFormats*.t.

The Design of Sound Storage Systems

It is very simple to store digital audio data on a computer’s hard disk; one simply needs to take the stream of binary numbers that come out of an analog-to-digital convertor (ADC)—or out of a software sound synthesis program—and write them in sequence to a disk file (Mathews 1969; Moore 1990). If one desires any flexibility, however, such as the possibility of storing sound files with different formats—a mix of mono- and stereophonic files, for example—it is necessary to store some auxiliary information along with the “raw” sample data.

The simplest way to do this is to attach a descriptive data structure to the file that gives the relevant format data—sample rate, sample format, number of channels, etc. This data structure is referred to a sound file “header,” and a program that uses the sound files (e.g., the “play” program sends samples to the digital-to-analog convertor [DAC]) can read this header and configure its hardware interface (e.g., the DAC) appropriately. This is, in fact, how most sound file systems work—they define a (fixed- or variable-format) header data structure and provide some (generally C-language) API functions for handling sound in memory and accessing disk and DAC/ADC I/O.

Some sound file systems are designed to handle large, single-data-block file storage, such as would be required for digital recording, mixing, or software sound synthesis systems. They often support several data formats (compressed, high-resolution, etc.), and allow users to attach text comments and some manner of marker or cue pointers to sound files. Other systems offer sound files that can contain multiple sound data “chunks” and store explicit MIDI voice data together with them. These formats better support sample data dump management for sample-based synthesizer hardware, but often do not support a wide enough selection of sample rates and formats for use on workstation-based systems.

The design of sound file systems is closely related to a number of other fields in music technology and knowledge representation. There is significant “overlap” with the design of general music representation systems (Dannenberg 1993; Wiggins et al. 1993), of storage formats for MIDI songs, digital mixer data, and signal processing algorithms, with the interchange formats used in electronic mail systems and the world-wide hypertext web, and with the specification of object types and databases for sound and music.

The most sophisticated sound processing applications being developed today place quite complex requirements on the underlying sound file system, and modern multimedia tools are hindered to a large extent by the weakness of the most popular sound file formats used on personal computers (see the system descriptions below).

Issues in Sound File Systems

The design of sound file systems is a rich topic, and designers must weigh a number of conflicting requirements: “medium-to-high” sustained throughput (1 to several MB/sec); “medium-to-large” files (1 to many MB per file); flexible mark-up and annotation; the need for markers, cue points, and links into a sound file’s data; and the poor compressibility of raw sound data. Decisions made in the construction of a sound file system have ramifications on its flexibility, scope, throughput, speed and space overhead, and the portability of the tools that use it. A system’s speed—its throughput for real-time multi-channel ADC/DAC performance—storage overhead—the storage efficiency for very large sound files—the support for different sound file formats—what data is stored in the header and how is it used by programs—and API flexibility (what extra features are supported, see below) are important criteria for comparison of sound file structures.

The most basic design question is whether the operating system’s “native” file system is sufficient for sound file storage and performance, and, if so, how to encode and annotate sound sample data for storage in standard disk files and usage by sound-processing programs. In the case that a special sound file system is used (rare on general-purpose computers today), the design of this system presents complex and multi-variate problems.

Simple sound file systems provide fixed-format headers with data fields such as those mentioned above, giving the sound file’s sample rate, sample format and size, number of channels, and possibly other descriptive data. If one wishes to allow more flexible manipulation of sound files, it soon becomes apparent that more complex and extensible sound file headers are desirable. One might, for example, wish to store a text comment or annotation together with a sound file. This text might be a transcription of the spoken text of a sound file, a descriptive comment, or the program text that was used to generate the sound file. Several formats store the maximum amplitude (per channel) with the sound file; this can be extremely useful in sound file processing. One can also store a (derived or applied) envelope function with the sample data. The system might allow “links” between sound files, so that one file can contain the information that it was derived from another one.

Some formats allow users to place “cue points” in sound files, so that one can encode, for example, the starting sample of a given musical phrase or spoken word. This facility allows for the creation of “virtual” sound files (Roth, Kendall, and Decker 1985), which are sound files that contain no sample data at all, but rather have “pointers” into other sound files using their cue points rather than copying their samples—e.g., a sound file that represents the third note of another file.

The decision of whether or not the sample data of a sound file must be stored in a contiguous disk file, or even all on the same disk, varies among systems. File formats that do not allow fragmented sample storage limit the user’s ability to create a sound file that is larger than the available free space on any given disk.

Sample-management-oriented systems include data in the sound file header that describes the suggested usage of the sample on a RAM-based MIDI synthesizer: the sample’s basic pitch and gain factor, its useful transposition and dynamic range, etc. These formats often allow whole families of samples to be stored in a single file together with their voice mapping data.

In future systems we will likely require powerful system-wide version management and flexible hypertext database index information (as in the HyperText Mark-up Language HTML) to be included in sound files as well.

The Components of a Sound File System

We introduced the concept of a sound file header above; it is typically a data structure that encodes the basic properties of the sound file, and that can be read and overwritten by programs that manipulate the sound file and its sample data. The description of this structure—typically given in a shared C-language header file—can be used to write low-level accessing functions for sound data—the system’s API, often provided by the hardware manufacturer—and with these functions to write end-user digital sound processing programs. There are several utility programs that are common to most sound file systems; these are described briefly below.

The most common operation on sound files is listening to them. This requires a program to “play” a sound file by sending its sample data to a computer’s DAC in real time. This program will normally read the sound file’s header, set up the required memory buffers, and set up the DAC for the proper sample rate and format. There are many other possible options included in the more sophisticated play programs, such as real-time mixing of several sound files, synchronization with external events (e.g., triggering sound files from MIDI), or real-time sample format conversion (e.g., expanding compressed sample blocks, or playing floating-point sample data on integer DACs).

For systems that support real-time sound recording, a “record” program will be necessary to read sample data from the ADC and write it to a sound file with the proper header. Record programs typically have options for setting the rate, number of channels, and other properties of the file; they use these settings to configure the ADC before recording and then write these settings into the sound file header.

If sound files are to be freely manipulated and processed by application programs, stand-alone utility programs must be provided that read and write sound files with the proper headers. On UNIX systems, programs called “fromsound” and “tosound” (or “soundin” and “soundout” on older systems) can be used in shell “pipes” connecting several programs together with the output of one serving as the input of another (written `command1 | command2`). Programs that provide the tosound functionality generally have options that determine the characteristics (rate, format, etc.) of the output sound file (Moore 1990). This allows one to say, for example,

```
sound_compiler orch_file score_file | tosound -channels 4 output.snd
```

or

```
fromsound sound_3 | reverberator | tosound -rate 48000 sound_3.rev
```

Supporting a sound file system involves providing a programming library (API) with functions for sound file creation, reading, writing, and some transformation of sounds in memory. The API allows users to create their own programs that have built-in interfaces to the sound file system. These are typically provided in the form of C-language specification and implementation files. Depending on the application focus of the sound file system, the API may be range from giving only very simple file I/O functions to sophisticated sampler data format translators or hypermedia sound managers.

Sound File Systems Throughout History

Early computer sound synthesis systems (Mathews 1969; Thieberger 1995) used programs to generate streams of binary numbers (“samples”—generally 12-bit integers at the time), and then wrote these number streams to multi-track digital magnetic tape. The “performance” of these tapes then took place on a special computer that was connected to a digital-to-analog convertor (DAC). This was done as a two-step process because the machines on which the samples were computed were generally large mainframes without the facilities for the kind of sustained high-bandwidth real-time I/O necessary for sound conversion. The DAC machines were often real-time mini-computers or laboratory systems.

As computers increased in speed and decreased in expense, and time-sharing operating systems were developed, it became practical to perform the sample computation and the D/A conversion on the same machine. The Digital Equipment Corp. (DEC) PDP-11 series of computers was popular for this starting in the mid-1970s. As a ramification of these developments, it was desirable to store the sample data on the computer’s local magnetic disk drives; the topic of just how to do this efficiently and flexibly arose, leading to the plethora of sound file systems in use today.

Several of the computer music platforms of the 1970s used the DEC RT-11 or RSX-11M operating systems, and made use of the PDP-11 system’s own file system for sound storage. The growing popularity of the UNIX operating system—accelerated by the introduction of the DEC VAX-11/780 computer in 1977, and of AT&T UNIX version 7 in 1978—motivated many computer music studios to adopt UNIX as their operating system. One major problem was that the original UNIX v7 file system could generally not support real-time sound file performance because of its limited throughput and poor control of file fragmentation. The data of a single file could be spread in blocks all over the disk, making high-speed sequential access unpredictable, and the (space and speed) overhead of its small-file-oriented addressing scheme was significant for huge sound files. Other problems were the lack of support for multi-volume file systems (i.e., having one file spread over several disk partitions or even several disks), and lack of standard facilities for separating annotation from data (i.e., storing information about a sound file separate from the sound’s sample data using “resource forks” or some manner of hyper-text links).

In response to these short-comings of the UNIX v7 file system, several alternatives were developed in the late 1970s and early 1980s. The most widely used one of these was D. Gareth Loy’s “csound” file system (Loy 1982)—not to be confused with the later-developed software sound synthesis language of the same name. Csound part of the CARL software package written at the Computer Audio Research Laboratory (CARL) at the University of California in San Diego (Moore 1982; Loy 1993). For the smaller PDP-11 machines, Robert Gross at Eastman School of Music (later at the University of California, Berkeley), developed his Cylinder-Contiguous Sound file System (CCSS), based on csound (Gross 1982). These systems are discussed in more detail below; the main features they share are the separation of sound files into one part that was stored in a standard UNIX file and contains the information about the format of the sound file (the header), and another part that is stored on a special disk partition—not managed by the UNIX file system—that provides much larger disk allocation blocks (such as one disk cylinder), and therefore better throughput and less data fragmentation. Because these packages implement their own file systems, special utility programs are required to manage them. Even such simple tools as the UNIX copy command (`cp`) have to be re-implemented to handle sound files properly (i.e., a `cpsf` command for sound files). The csound system, in fact, implements the whole suite of file management utilities—dump/restore, copy/rename/remove, file system consistency and fragmentation checking, etc.—for these special sound file systems.

A number of systems have been developed since csound—some that use simple sound file headers and the host’s native file system, and others based on special proprietary file systems. The csound idea is still very much alive in the IRCAM, BICSF, and EBICSF sound file systems (described below), though the storage policies of these newer systems are different than in csound in that they all use the host’s native file system, the Berkeley UNIX “fast” file system or the Macintosh’s HFS.

During the 1908s, as UNIX-based systems increased in sophistication and moved onto workstation-class computers, digital audio signal processing also moved onto personal computers with the advent of (primarily) Macintosh-based integrated hardware/software systems such as Digidesign’s Sound Tools and Pro Tools, Studer/Editech’s Dyaxis using MacMix, and Symbolic Sound Corp.’s Kyma system. These systems generally share three features: a graphical user interface that made command-line-oriented UNIX systems look quite out-moded; a special-purpose hardware digital signal processor; and a proprietary sound file system. Some go as far as requiring special dedicated disks (that are not manipulable by other Macintosh programs) for sound file storage. Recently, these systems have become more “interoperable” with increasing support for portable sound file formats such as AIFF, which is widely supported on several platforms.

MIDI, and the wide use of MIDI-based sampling synthesizers, also had an effect on sound file systems. The MIDI sample dump format is—strictly taken—only an interchange format, but it has served as the basis of several storage systems, and influenced several others.

More recently, several new systems have appeared, either based on existing formats—such as RIFF being derived from AIFF, or BICSF from csound—or new in design—such as SPHERE. We will introduce and compare these systems below.

Sampling and Digital Sound Representation

Sampled audio data can be characterized by the parameters of the ADC used when the sound was recorded (or the settings of the synthesis program that created it); the same settings must be used to configure the DAC to play the data back. These basic characteristics include the sound’s sampling rate (the frequency in Hz of the convertors), the format and precision of the binary numbers used for the samples (e.g., 16-bit integer, or 64-bit floating-point), the sample compression scheme (e.g., none, µ-law, or Huffman coding), and number of channels in the sound. There is a debate as to whether sampling rates are to be quoted in Hz (or kHz), or whether it is more politically correct to use the term “samples per second” (samples/sec). For simplicity, we will use Hz in this article.

Sampling Rates

For various historical reasons, some sampling rates are more popular than others. Some recording hardware is restricted to (approximations of) a few of these rates, while some playback hardware has support for many of them. We list the most common sample rates in Table 1 along with the devices with which they are most-frequently used. (The popularity of divisors of common rates can be explained by the simplicity of clock frequency dividing circuits).

Rate in Hz	Description
5500	One fourth of 22050 (the Macintosh’s default sampling rate)
7333	One third of 22050
8000	Telephony standard used with µ-law and A-law encoding (see below)

8012.8210513	NeXT workstation—the rate used by their Telco CODECs
11025	Half of 22050
16000	G.722 telephony standard, or half of 32000
16384	(= 16 * 1024) csound/BICSF
16726.8	NTSC television rate C-3 period 214 (= 7159090.5 / (214 * 2))
18900	CD-ROM/XA standard
22050	Either 22050, half the CD sampling rate, or the default Macintosh rate
22254.5454...	Derived from the horizontal scan rate of the original 128k Macintosh monitor
32000	Digital radio, NICAM (Nearly-Instantaneous Companded Audio Multiplex [IBA/BREMA/BBC]) and other TV work; also on long-play DAT and HDTV
32768	(= 32 * 1024) csound/BICSF
37800	CD-ROM/XA standard for higher quality
44056	This rate is used by some professional audio equipment to fit an integral number of samples in an NTSC video frame
44100	The CD sampling rate
48000	The DAT sampling rate for domestic use
49152	(= 48 * 1024) csound/BICSF
> 50000	Higher rates are sometimes used in software DSP, but are generally not supported by DAC or ADC hardware (with some interesting exceptions)

Table 1: Common sound sampling rates

While musicians generally find it egregious, most people do not seem to notice if recorded sound is played back at a slightly different rate, transposed by 1 or 2 percent (tens of cents of transposition). On the other hand, if recorded data is being fed into a playback device in real-time (e.g., over a network), even the smallest difference in sampling rate will eventually either flood or drain the data buffering scheme.

There is an emerging tendency to standardize on a small number of sampling rates and encoding formats, even if the file formats used to store the data may differ. The most popular sample rates and formats in current use are shown in Table 2.

Rate in Hz	Precision and format
8000	8-bit µ-law-scaled mono
22050	8- or 16-bit linear unsigned integer mono and stereo
44100	16-bit linear signed integer mono and stereo

Table 2: “Standard” sampling rates and sample formats

Sample Size and Convertor Resolution

Along with the sampling rate, the numerical accuracy of the data samples has a major impact on the audio quality and storage compactness of digital audio. As a rule, each bit of (linear) resolution provides for 6 dB of signal-to-noise ratio (SNR) (Moore 1990). The most common sample format for modern systems is signed 16-bit linear integers (sample values ±32768), providing a theoretical maximum SNR of 96 dB. Higher-resolution convertors are becoming more cost effective, however, and several sound file systems support both larger-size (e.g., 24-bit) integers and floating-point samples.

It should be noted that it is possible to have A/D and D/A convertors that support the same sampling rate and format, yet differ greatly in audio quality—one need only to compare low-end and high-end CD players to hear this. There are several important features of analog audio interfaces that are independent of the sample rate and resolution. The most important among these (Moore 1990; Pohlmann 1992) are the monotonicity of the convertor (i.e., are the convertors of high enough quality that the lowest-order bits actually matter), and the quality of the analog section to which it is attached (i.e., are the I/O filters and preamplifiers good enough to maintain the 96 dB quality of the conversion). This article is not an introduction to digital audio interfaces, but it should be noted that two devices that both support “CD quality” I/O can still differ greatly in the quality of their A/D or D/A conversion.

Handling Multi-channel Sounds

For multi-channel sounds, one speaks of a sample “frame” meaning a group of samples that are to be played simultaneously, one-per-channel. Sample streams are generally interleaved on a frame-by-frame basis; if there are N channels, each frame will contain N samples, one for each channel. The sampling rate is really the number of *frames* per second, so for stereo data recorded at 8 kHz, there are actually 16000 samples per second. For stereo frames, the left channel usually comes before the right. Several formats standardize sample orders for higher channel numbers (four, six, eight, or more); samples of quadraphonic sound, for example, may appear as [LF, RF, LR, RR], as [L, R, Center, Surround], or in some other order.

Sound Compression Schemes

Strange though it seems, audio data is remarkably hard to compress effectively. The general rule is that a compression technique can be good at maintaining the quality of the original sound (i.e., it can be side-effect-free or “inaudible”), or it can be executable in real time using a standard CPU (i.e., “do-able”), or it can achieve a good compression factor (i.e., “effective”), but not all three. There are several well-standardized schemes for sound data compression that we will discuss below. These are based on three basic methods: one can use a compact numerical representation (such as 8-bit floating-point) for sample values, one can compact blocks of samples using standard data compression techniques (e.g., Huffman or LRZ coding), or one can perform some kind of analysis (e.g., spectral) of the sound data to get a more compact representation (as in FFT-based or LPC vocoders). Each of these techniques has relative advantages and disadvantages with respect to the three criteria mentioned above. The second type—using standard compression algorithms on raw sample data—is inaudible, but achieves rather poor compression ratios and is too slow to support real-time expansion on many platforms.

One of the most popular methods of the first kind is called µ-law or u-law encoding—pronounced “mu-law” (µ is the Greek letter mu). Standard µ-law samples are logarithmically encoded in 8 bits—like tiny floating-point numbers—so that their dynamic range is that of 12-bit linear integers. Because the compression uses a simple mathematical function of each data value, one can easily expand µ-law samples in real time. Using this technique to compress 16-bit samples is clearly audible, though, as a decrease of approximately 24 dB in the dynamic range of the signal. Another encoding similar to µ-law is called A-law, and is used as a European telephony standard. There is less support for it on personal computers and UNIX workstations. The exact formulae for µ-law and A-law encoding can be found in several of the references and in the FAQ on which this article is based; C-language source code for converting to/from µ-law (written by Jef Poskanzer) is distributed as part of the SOX software package mentioned below. Some extensions to this define µ-law compression with extra silence detection—a more analysis-based technique.

Public standards for voice compression are slowly gaining popularity, e.g., CCITT (the international standards organization for telecommunications) standards G.721 (Adaptive Delta Pulse Code Modulation [ADPCM] at 32 kBaud) and G.723 (ADPCM at 24 and 40 kBaud). Sun Microsystems, Inc. has placed the source code of a portable implementation of these algorithms (as well as G.711, which defines A-law and µ-law) in the public domain. One place from which to retrieve this source code this source code from is mitpress.mit.edu:/pub/Computer-Music-Journal/Code/ccitt-adpcm.tar.Z. (Needless to say, their proprietary implementation distributed in binary form with their Solaris operating system is better.) Source code for another 32 kBaud ADPCM implementation, assumed to be compatible with Intel’s DVI audio format, can be retrieved from ftp.cwi.nl:/pub/audio/adpcm.shar.

The words “LPC” and “CELP” are sometimes used interchangeably to describe two related spectral-model-based compression schemes. They are different, however, in the compression ratios and audio quality they offer. An LPC (Linear Predictive Coding) coder analyzes sound using a simple, “source/filter” model based on the human vocal tract; it is a “subtractive” technique that analyzes the sound into a time-varying source signal (like the glottal pulses from the throat) that is passed through a dynamic filter (like the mouth and nasal cavities). The LPC coder then discards the original sound and stores the (compact) parameters of a “best-fit” source/filter model (to some given filter length and driver characterization). An LPC decoder uses these parameters to generate synthetic sound that is “similar” to the original—depending on the precision and update rate of the filters and accuracy of the driver. LPC coding can achieve very significant compression ratios (down to very low bit rates), and the re-synthesized sound is generally intelligible (for speech) but often sounds "like a machine is talking." LPC is generally not useful on complex, multi-timbral sounds.

A CELP (Code Excited Linear Prediction) coder does the same LPC version, but then computes the errors between the original signal and the synthetic model and transmits both the LPC model parameters and a very compressed representation of the errors. The compressed representation is an index into a "code book" function array shared between coder and decoder—this is why it is called "code excited." A CELP coder does much more work than an LPC coder—usually about an order of magnitude more computation—but the result is a much higher quality resynthesis of the original sound. A CELP decoder ran be written to be quite efficient and run in real time on "standard" hardware (Campbell, Tremain, and Welch 1990). The implementation of the FIPS-1016 standard CELP system we are working on at CWI offers essentially the same quality as the 32 kBaud ADPCM coder but uses only 4.8 kBaud (the same as a speech LPC coder). A free implementation of a CELP coder (in C and FORTRAN) can be found in the Internet-accessible file super-org:/put/celp_3.2a.tar.Z.

Apple has an Audio Compression/Expansion scheme called ACE (used on the Apple II/GS) or MACE (on the Macintosh) (Apple 1986). This is a "lossy" scheme that attempts to predict where the wave will go on the next sample. There is very little quality change on 8:4 compression, somewhat more for 8:3. It does guarantee 50 or 62.5 percent compression, though.

The GSM 06.10 standard is a speech encoding in use in Europe that compresses 160 13-bit samples into 260 bits (33 bytes), i.e., 1650 Bytes/sec (13200 Baud at 8000 samples/sec). A free implementation of this technique in C can be retrieved from tub.cs.tu-berlin.de:/pub/tubmik/gsm-1.0.tar.Z.

For 8-bit delta-encoded sample data, a Huffman encoding has been shown to be relatively successful at achieving good compression ratios.

For 16-bit data, companies such Sony, Philips, and NeXT have invested large sums of money in the development of proprietary compression schemes. Information about PASC (Philips' scheme) can be found in *Advanced Digital Audio* (Pohlmann 1992).

Tony Robinson (ajr@eng.cam.ac.uk) has written a relatively "good/fast/loss-less" compression tool for several different audio formats (particularly good for WAV and MOD files). The software is available by anonymous ftp from svr-ftp.eng.cam.ac.uk:/misc/shorten-1.08.tar.Z.

The FAQ document for the USENET news group comp.compression has a description of the 6:1 audio compression scheme used by MPEG (for Motion Pictures

Expert Group—a video compression standard). It is very interesting to note that video compression schemes reach much higher ratios (on the order of 26:1) than for audio (a topic in itself). The comp.compression FAQ can be retrieved from the two files rtfm.mit.edu:/pub/usenet/news.answers/compression-faq/{part1, part2}. Comp.compression also carries a regular posting called “How to uncompress anything” by David Lemson (lemson@uiuc.edu), which (tersely) hints on which program you need to uncompress a file from a wide variety of format. This document can be retrieved as ftp.cso.uiuc.edu:/doc/pcnet/compression.

Compact-disc-interactive (CD-I) machines form a special category in terms of their compression schemes. The following formats are used: PCM 44.1 kHz 16-bit integer CD format; ADPCM Level A, 37.8 kHz 8-bit; Level B, 37.8 kHz 4-bit; and Level C, 18.9 kHz 4-bit.

There are other formats being developed daily, so this list is sure to grow.

A Child’s Garden of Sound File Formats

Sound file formats are a separate issue from DAC/ADC hardware characteristics. The Appendix describes the hardware support for digital audio found on a number of popular workstations and personal computers.

Many proprietary or system-dependent file formats have been developed for audio data, and a few have found wider, portable application. Because of the similarity of these “mainstream” systems, it is generally possible to define conversions between any pair of sound file formats—sometimes losing information, however. In this section, we will discuss the concrete sound file systems we find in common use today, evaluating their header formats as well as the software API and utility applications provided for users of their “native” machines.

We will distinguish two types of file formats: self-describing formats, where the sample parameters and encoding are made explicit in some form of file header (this is greatly preferred); and “raw” or header-less formats, where the device parameters and encoding are fixed or assumed (generally a bad idea).

Self-describing file formats often support several different data encodings, where the fields of the sound file header indicate a particular format/rate/encoding variant. “Headerless” formats define a single encoding and allow no variation in device parameters (except sometimes sampling rate, which can be difficult to determine out by any other means than listening to the samples). They are generally only used on systems where the file type can be determined from some type resource—as on the Macintosh—or strictly interpreted from the file name—as on the Sun workstation’s “.au” files.

Self-describing File Formats

As introduced above, the headers of self-describing sound files contain a data structure with the parameters of the sampling device and sometimes other information (e.g., a human-readable description of the sound or a copyright notice). In some cases (e.g., NeXT/Sun, AVR), this is a fixed data structure of known size and format that can be easily read from the file and interpreted by programs. The standard fields are the sound’s sample rate, number of channels, sample format and/or precision, sample compression switches, and optional comment strings.

Some formats do not define a fixed header format, but have a small “basic” header and allow files to contain a number of “chunks” of encoding information, annotation data, or other fields intermingled with chunks of sample data. In these cases (e.g., AIFF, WAV, BICSF), each chunk generally has its own header that has a type or code tag field and also gives the size of the chunk’s data. The rest of the chunk data can be interpreted according to the chunk’s type or tag (or the chunk can be skipped given the size found in its header). Chunked file formats—also called “tagged” files—may support vastly different facilities depending on the standard chunk types they provide, and whether or not they allow multiple sample data chunks in one file. The programmer’s API to chunked file formats will involve more complicated functions to assemble variable-size arrays of file chunks and interpret them into a set of in-memory data structures.

Most headers begin with a “magic number”—a pre-defined integer that identifies the file as being a sound file. (This habit comes from a kluge used in UNIX applications because the file system supports only very primitive file properties. More advanced file systems such as the Macintosh’s use a file’s creator/type fields to identify it as a sound file.)

The data encoding given in sound file headers defines how the actual samples are stored in the file, e.g., as signed or unsigned numbers, as bytes or short integers, in “little-endian” or “big-endian” byte order, etc. Strictly taken, the channel interleaving is also part of the encoding, although there is little variation among formats in this area.

Table 3 presents an overview of some of the most widely used sound file formats. The first column gives the name of the format; the second the default file name extension (e.g., NeXT sound files generally have file names that end in “.snd”). The third column tells with which hardware platforms or software tools the format is commonly used. The most important fields of the header are shown in the last column, as are the default format and encoding in the case that they are fixed. These formats are described in detail below.

Format Name	FileName Extension	Origin/Use	Description
NeXT/Sun	.snd, .au	NeXT, Sun, DEC	Variable-size, fixed-format header with sample rate, # of channels, encoding, comment

			string; supports many data formats.
AVR	none	Atari/Apple	Fixed-size, fixed-format header with sample rate, mono/stereo, 8- or 16-bit format, MIDI sample loop and MIDI application information.
SPHERE	none	NIST/ARPA	Fixed-size, variable-format header, with rate, etc. and version, ID, and database key info.
Sound Des. I	None	Digidesign	Fixed-size, fixed-format header with sample rate, integer sample resolution, and graphical display information; monophonic files only.
Sound Des. II.	None	Digidesign	Variable-size, variable-format header using the Macintosh file system's "resource fork;" includes the "standard" properties as well as SMPTE and measure/beat information.
IFF	.iff	Amiga	"Chunked" format with variable-size, variable-format header—rate, # of channels, instrument info., envelope function chunks, supports 8-bit sample data only.
AIFF	.aif(f)	Apple, SGI, etc.	Chunked format with fields for rate, # of channels, sample width (8-24-bit integers), text and MIDI chunks, numbered marker points.
AIFC	.aif(c)	Apple, SGI	Extension of AIFF supporting µ-law compression
WAV	.wav	Microsoft/IBM	Non-compatible version of AIFC; header with rate, # of channels, sample width, MIDI and text chunks; supports IBM compression formats.
(E)BICSF	.sf, .snd	CARL/UNIX/IRCAM	Variable-size, variable-format header with rate, # of channels, encoding; supports inter-file links, virtual, and composite (mix) files, named marker regions, lots of annotation chunk types.
VOC	.voc	PC Soundblaster	Chunked format with sample, loop, silence, repeat, and other blocks, supports at most 8-bit samples (down to 2 bits).
MOD	.mod, .nst	Amiga	Sample dump format (multiple sounds per file); 8 bits; header with rate, MIDI voice data.
MIME	none	Internet mail	For including speech in electronic mail; supports compressed mono 8 kHz 8-bit µ-law data formatted as ASCII strings.
MIDI SDS	none	MIDI Manufacturers	Fixed-size, fixed-format header with sample period, sample width (integers), and loop info.; multiple sounds per file.

Table 3: Self-describing sound file formats

Note that the filename extension ".snd" is ambiguous; it can be either the self-describing NeXT or EBICSF format, or the header-less Mac/PC format, or even a header-less Amiga format (see below).

Headerless File Formats

A sound file system that intends to provide any flexibility or portability will use a self-describing sound file format. Because most recorded sounds start with a bit of silence containing small amounts of background noise, however, it is usually easy to distinguish 8-bit and 16-bit signed and unsigned formats by “reading” the beginning of the data as a number stream (UNIX users would say `od -b < file | head`). The signed formats will have an abundance of bytes with values 0376, 0377, 0, 1, 2, while the unsigned formats will have 0176, 0177, 0200, 0201, 0202 instead. (Using the UNIX shell command `od -c` will show any headers that are put in front of the file.)

Table 4 shows a few of the (increasingly rare) header-less formats, along with their applications and properties.

Name ext.	Origin/Use	File properties
.snd, .fssd	Mac, PC	Variable rate, 1 channel, 8-bit unsigned integers
.ul	US telephony	8 kHz, 1 channel, 8-bit µ-law encoding
.snd	Amiga	Variable rate, 1 channel, 8-bit signed integers
none (HCOM)	Macintosh	8 bits, 16 kHz, 1 ch; uses Huffman compression
.au	Sun	8 kHz, 1 channel, 8-bit µ-law encoding; may or may not include a NeXT/Sun file header

Table 4: Headerless sound file formats

Sound File Format Descriptions

Having presented the design issues that are important to sound file system performance, and introduced the systems to be discussed, we will now explore them in detail and give examples of their use. They are presented in approximate order of increasing complexity (and sophistication), with those using simple, fixed-format headers preceding the more complex and variable “chunked” formats. We will restrict ourselves to self-describing formats that support at least “CD-quality” data formats (i.e., 16-bit samples).

The NeXT/Sun Sound File Format

The NeXT computer’s sound support software (which is also provided with the NeXTSTEP environment on other hardware platforms) uses a self-describing sound file format based on the “SNDSoundStruct” data structure. Formats identical to this are also used on the Sun Microsystems, Inc. SPARCstations and DEC DECstations. The header structure is included at the beginning of every sound file and is also used to describe audio data in memory as part of an application. The NeXT Sound Kit includes an Objective C object-oriented “Sound” class (Jaffe and Boynton 1989) whose instances are used to encapsulate the SNDSoundStruct and provide methods (functions) to access and modify its attributes. The Sound class is the NeXT machine’s API to the sound file system. The other hardware platforms that use this format provide different APIs and utility programs (see below).

The SNDSoundStruct Data Structure

The SNDSoundStruct data structure is used on-disk and in-memory as a header—its information describes the attributes of the sound. In on-disk sound files, it is followed by the data that represent the sound. This means that a sound file has a simple, fixed-format header, followed by exactly one sample data block. The data structure is defined in a global C-language header file as shown in Figure 1. The fields are described in more detail below.

```
/* NeXT/Sun Sound File Header Data Structure */
typedef struct {
   int magic;              /* Magic number for sound files (0x2e736e64) */
   int dataLocation;       /* Byte offset to the data (>= 28) */
   int dataSize;           /* Number of Bytes of data (without header) */
   int dataFormat;         /* Pre-defined sample format code */
   int samplingRate;       /* Sampling rate in Hz */
   int channelCount;       /* Number of channels (1, 2, or 4) */
   char info[4];           /* Optional text (comment), 4 Bytes */
                           /* (or larger if dataLocation > 28) */
} SNDSoundStruct;
```

Figure 1: The NeXT/Sun sound file header data structure

The first field—`magic`—is a file ID (or “magic number”) that is used to identify the structure as a SNDSoundStruct. Its value is pre-defined (as a `#define` macro in the header file) as hexadecimal 2e736e64 (which just happens to equal the 4-byte string “.snd”). In C this is written 0x2e736e64. All NeXT/Sun sound files start with these 32 bits.

A NeXT/Sun sound file contains a SNDSoundStruct header followed by sound data. The data itself is external to, although usually contiguous with, the header. (Nonetheless, it is often useful to speak of the SNDSoundStruct as the header and the data.) The header field `dataLocation` is used to point to the data. On-disk, this value is an offset (in bytes) from the beginning of the SNDSoundStruct to the first byte of sound data (typically 28—the size of the 7 fields of the header). The data, in this case, immediately follows the structure, so dataLocation can also be thought of as the size of the header. If the user wishes to have a longer comment (or anything else) at the end of the header, he/she need only to set dataLocation to a larger value. The other use of dataLocation—as an address that locates data that is not contiguous with the structure—is used for in-memory sounds, not for disk-based sounds.

The `dataSize`, `dataFormat`, `samplingRate`, and `channelCount` fields describe the size and format of the data samples. The dataSize (the total size of the files sample block) is given in bytes (not including the size of the SNDSoundStruct). The dataFormat is a pre-defined code that identifies the type of sound. For sampled sounds, this is the quantization format (e.g., 8-bit µ-law, 16-bit linear integers, or 32-bit floating-point), however, the data can also be instructions for synthesizing a sound on the DSP, or sound display data (see below). The samplingRate field gives the sampling rate in Hz (if the data is samples), and the channelCount is the number of channels of sampled sound (the frame size); one, two, and four channels are supported.

`Info` is a string that can be used to provide a comment or textual description of the sound. The size of the info field is set when the structure is created and (unfortunately) cannot be enlarged thereafter. It is at least four bytes long (even if unused). There is a utility program that adds space for a longer comment by increasing the dataLocation offset. There is no standard interpretation of the comment, but user programs are free to encode additional data in it.

Format Codes

A sound’s format is represented as a positive 32-bit integer. NeXT reserves the integers 0 through 255; you can define your own format and represent it with an integer greater than 255 (though none of the system-provided tools will know what to do with it). Most of the formats defined by NeXT describe the amplitude quantization of sampled sound data—e.g., 8-bit µ-law, 8-, 16-, 24-, or 32-bit integers, or 32- or 64-bit floating-point numbers. Various compression and emphasis modes are known, including ADPCM and µ-law with silence detection.

Sound files can contain altogether other kinds of data, such as sound envelope display data or DSP commands (in which case they are not really sound files; they just re-use the same header format. There is support for “indirect” header (where the data might not be contiguous with the header), and “nested” headers.

Fragmented Sound Data

Sound data is usually stored in a contiguous block of memory or a single linear file on disk. However, when sampled sound data is edited (such that a portion of the sound is deleted or a portion inserted), the data may become discontiguous, or fragmented. Each fragment of data is given its own SNDSoundStruct header; thus, each fragment becomes a separate SNDSoundStruct structure. Fragmentation serves one purpose—it avoids the high cost of moving data when the sound is edited. Recorded sound data that has not been edited is naturally unfragmented. Writing fragmented (edited) sound from memory to disk “expands” and linearizes it with the NeXT/Sun system.

Common Combinations

The most frequently used sound file formats on NeXT and Sun machines are shown in Table 5.

Usage	Rate	Channels	Format Code
Voice file	8012	1	8-bit μ-law
System beep	22050	1 or 2	16-bit linear
“CD-quality”	44100	2	16-bit linear
“DAT-quality”	48000	2 or 4	16- or 24-bit linear, or 32-bit floating-point

Table 5: Common NeXT/Sun sound file types

Example

Table 6 shows the contents of a NeXT/Sun header for a file that has a stereo, 44.1 kHz sample lasting 1.0 sec stored as 16-bit integer samples. The types of the fields can be determined from Figure 1. The length of each field is given in Bytes, leading to a 28-Byte header structure, hence the dataLocation value of 28. If one wanted to use the info field for a longer comment or other data structure, one would use a larger data location offset value.

Size	Name	Value	Comment
4	magic	0x2e736e64	Magic number = “.snd” as a integer
4	dataLocation	28	Offset to the data in Bytes
4	dataSize	176200	Number of bytes of data (= 44100 * 2 * 2)
4	dataFormat	3	Pre-defined code for 16-bit linear data
4	samplingRate	44100	Sample rate in Hz
4	channelCount	2	Number of channels (for stereo)
4	info	“”	Unused text information
176200	samples	. . .	Sound data array starts here

Table 6: An example of a NeXT/Sun sound file header

API and Support

Basic sound operations, such as playing, recording, and cut-and-paste editing, are most easily performed with the NeXT Music Kit’s Sound object. In many cases, this obviates the need for in-depth understanding of the SNDSoundStruct altogether. For example, if you simply want to incorporate sound effects into an application, or to provide a simple graphic sound editor (such as the one in the NeXTMail application), you need not be aware of the details of the header data structure. UNIX shell utility programs are provided as part of the NeXTSTEP system to add file headers to “raw” sample data files, to convert the format of a sound file, to list the properties of sound files, and for other operations. These are not, however, very well integrated into the UNIX shell environment. The command to list sound files is called “sndinfo” rather than “lsf” (or even better being integrated into the “ls” command). The NeXT file browser recognizes sound files, so that play programs or graphical editors can be started when the user “opens” a sound.

The Sun SPARCstations use a structure with the same members as SNDSoundStruct, with different names, and offer a simpler C-language API and play, record, and “raw2audio” (like “tosnd”) utilities. Other utilities include a dialog-based play/record program, and the native OpenLook file manager (finder) handles sound files correctly. There is another API called the Sun/COSE XAL Audio Library (Sun 1994) developed by a consortium of Sun, H-P, DEC, IBM, and other UNIX providers for the Common Operating System Environment (COSE). The XAL library provides a powerful object-oriented API to sampled sound processing and multi-format I/O. It supports the NeXT/Sun header, as well as AIFF, WAV, and other disk formats. Its in-memory format is opaque and is encapsulated by the API classes.

The DECstation 5000 workstations use a variant of the NeXT format that employs “little-endian” byte encoding and has a different magic number (0x0064732e in little-endian encoding).

API Functions for creating and reading NeXT/Sun file headers are also provided with the source code of several public domain software packages for sound format conversion(see below).

Discussion

The NeXT/Sun format is a simple self-describing format useful as a sound interchange format for most basic applications. It uses a fixed-format, variable-size header on disk and in memory and allows one sample block per file. Because the sound file header is a fixed, one-level data structure, writing a programming interface to read it from files or to generate it is quite easy.

The presence of the info field allows the user to add a text comment to a sound file, but as there is no standard as to how this string is interpreted by programs, very few applications use it for any “more interesting” annotation, such as storing the maximum amplitude or envelope as mentioned above.

The format supports several sample formats from 8-bit inteers to 64-bit floating-point numbers, but allows only μ-law or ADPCM compression as “standards.” It supports files with fragmented storage in memory only, and uses the UNIX file system to manage file fragmentation on disk. This brings with it the limitation that sound files may be no larger than the size of UNIX disk partitions, and may not be fragmented across several partitions.

There are pre-defined formats for a wide variety of sample formats, including 24-bit integer and 32- and 64-bit floating point numbers. Storing sounds as floating-point numbers is very useful when using “heavy duty” signal processing algorithms, such as higher-order digital filters or complex reverberators, though no commercial DACs support them (i.e., they must be converted into 16-bit integers before playing them). Several non-sample formats are also defined, such as DSP code or data blocks, display data, or “indirect” pointers.

Storing the sampling rate as an integer (rather than a long or float) may be considered a disadvantage, though this is really a rather minor point. Only AIFF and BICSF use floating-point numbers for sample rates. Allowing stereo or quadraphonic data is also a plus.

AVR File Format

One format that is quite popular (used by a number of commercial packages) on the Atari ST and Apple Macintosh platforms is the “AVR” format from Audio Visual Research. AVR format defines a fixed 128-byte file header that is described in the C-language data structure shown in Figure 2. Note that there are fields in the Figure that correspond roughly to all the fields of the NeXT/Sun format header, with the addition of several new fields, such as those that define loop points within a sampled sound and a MIDI note assignment. This format provides these to make it more powerful when used to store sound samples for use with a sampling synthesizer. A sound can be assigned to a given MIDI key, and can have two pointers into its sample data that determine where the sound is to loop in case it is sustained for longer than its recorded duration on a sampler.

```
/* AVR Format Header Data Structure--128 Bytes */
typedef struct {
   char magic[4];    /* Magic number = “2BIT” */
   char name[8];     /* Sample name (null-padded string) */
   short mono;       /* Number of channels--mono/stereo */
   short rez;        /* Sample resolution--8/16-bit */
   short sign;       /* Sample format--signed/unsigned */
   short loop;       /* Loop mode--on/off */
   short midi;       /* MIDI note assign or keyboard split */
   long rate;        /* Sample frequency in Hz */
   long size;        /* Data length in samples */
   long lbeg;        /* Offset to start of loop--0 if unused. */
   long lend;        /* Offset to end of loop--0 if unused. */
   short res1;       /* Reserved, MIDI keyboard split */
   short res2;       /* Reserved, sample compression */
   short res3;       /* Reserved */
   char ext[20];     /* Longer name space (if name[7] != 0) */
   char user[64];    /* User-defined; typically ASCII */
} AVRHeader
```

Figure 2: The AVR sound file header data structure

API and Support

There are C-language libraries for manipulating AVR format files on both the ST and Macintosh platforms that provide low-level header and data I/O functions. These have been used to construct a wide range of applications for sampler management and recorded sound processing.

Discussion

While the AVR header has more data fields than the NeXT/Sun’s, it is more restricted in its data format and less flexible in terms of not allowing any additional data in the header (other than the user-defined 64-Byte field), and supports fewer sample formats. AVR does, however, allow the inclusion in sound files of loop start and end points and MIDI keyboard splits, something of great use to users storing sound for use with a sample-based synthesizer. The default definition files define compression fields for µ- and A-law, ADPCM, and other schemes. Supporting only mono- or stereophonic samples is a problem for some users, as the limitation to integer samples of 8 or 16 bits. The “kluge” to allow long sound names (putting 7 characters at the front of the header and 20 more later on) is rather ugly. As with the NeXT/Sun format, the use of a fixed data structure makes the implementation of parsing/generating libraries easy. No high-level API (such as the NeXT’s) for AVR format files is known to the authors.

MIDI Sample Dump Standard

The Sample Dump Standard (SDS) adopted by the MIDI “powers that be” (the MIDI Manufacturers Association and the Japanese MIDI Standards Committee) defines the standard method for transfer of sound sample data between MIDI-equipped devices. Although it is primarily an interchange format, it can also be used as a sound file format of sorts, and supports high-resolution samples, so it warrants our attention here. More detailed descriptions are available in several MIDI-related references, and in the Internet file ftp.cs.ruu.nl:/pub/MIDI/DOC/mididump, from which the discussion below is derived.

The MIDI SDS defines a simple packet-oriented sample interchange protocol with which two devices can reliably share sample data. This protocol is described in terms of the command packets that one device (the “master”) sends to another (the “slave”) to initiate a transfer, how the slave device responds, and how the two devices manage the interaction.

The structure of the header that describes a MIDI sample dump is shown in Table 7. This structure is enclosed in a MIDI system exclusive (sysex) message by the first and last bytes, with the special values of hexadecimal f0 and f7. The rest of the data in the header gives the sample rate (actually its reciprocal the sample period, in nsec), the sample format (resolution in bits of the linear integer samples), and the starting and ending samples of the one loop region. As this format is mainly concerned with samples, each sound also has a MIDI channel number and a sample number; there may be several samples per channel.

Size	Field
1	Sysex header = f0
1	System type ID: universal non-real time = 7e
1	Channel number
1	Sub-ID: Header = 01
2	Sample number
1	Sample resolution
3	Sample period
3	Sample length
3	Sustain loop start point
3	Sustain loop end point
1	Loop type
1	End-of-Sysex (Eox) = f7

Table 7: MIDI sample dump header block

The linear format of this packet resembles the following (whereby all multi-Byte data are sent with the least-significant Byte [LSB] first)

```
F0 7E cc 01 ss ss ee ff ff ff gg gg gg hh hh hh ii ii ii jj F7
```

where

`cc` = MIDI channel number of the sample
`ss ss` = sample number (LSB first)
`ee` = sample resolution (number of significant bits; 8-28)
`ff ff ff` = sample period (1/sample rate) in nsec
`gg gg gg` = sample length, in Bytes
`hh hh hh` = sustain loop start point (Byte number)
`ii ii ii` = sustain loop end point (Byte number)
`jj` = loop type (00 = forwards only; 01 = alternating)

The structure of a MIDI SDS data packet is given in Table 8, where one sees the same MIDI sysex format, and 120 Bytes of sample data.

Size	Field
1	Sysex header
1	ID: Universal Non-Real Time
1	Channel Number
1	Sub ID: Data Packet

1	Packet Number
120	Sample Data (120 bytes)
1	Checksum
1	End-of-Sysex (Eox)

Table 8: MIDI sample dump data block

An example of the linear form of this packet is

```
F0 7E cc 02 kk <120 bytes> mm F7
```

where

`cc` = channel number

`kk` = running packet count (0-127)

`mm` = checksum (exclusive-or of 7E, `cc`, 02, `kk`, and the sample data)

The total size of a data packet is 127 bytes. This is to avoid overflow of the MIDI input buffer of a device that may want to receive an entire packet before processing it. The packet number begins at 00 and increments with each new packet. It resets to 00 after it reaches 7F, and continues counting. The packet number is used by the receiver to distinguish between a new data packet, or a resending of a previous packet. The packet number is followed by 120 bytes of data, which form 60, 40, or 30 data samples (most-significant Byte first for multi-Byte samples), depending on the length of a single data sample. Each data Byte holds seven bits, with the most-significant bit in each byte set to 0 in order to conform to the requirements of MIDI data transmission. Information is left justified within the 7-bit bytes, and unused bits are filled with 0. As an example, assume a 16-bit sample with the value 87E5 hexadecimal. In binary representation, that would be

1000 0111 1110 0101

and it would be encoded as the following MIDI data stream:

01000011 01111001 00100000

The checksum of the packet is the running exclusive-or (XOR) of all the data after the sysex byte, up to but not including the checksum itself.

For a file-based version of the MIDI SDS, an application may simply store the uninterpreted packets, or may “unpack” them, compacting the sample data and discarding the packet “wrappers” to store the data in some other format. The sample dump standard does not address file storage of sound, and the MIDI file format does not address sample dumps.

Discussion

The MIDI SDS format is very useful for up- and down-loading sample data from/to hardware synthesizers, but is rather weak as a sound file system. It supports only monophonic samples linear integer formats. Nevertheless, its role as an accepted standard interchange format makes it important to understand.

NIST SPHERE Format

A sound file format popular in the speech-processing community is the US National Institute for Standards and Technology’s SPeech HEader REsources (NIST SPHERE) standard. The most recent version of the SPHERE software package (in C and FORTRAN) is available from the Internet file jaguar.ncsl.nist.gov:/pub/sphere-v.tar.Z (where “v” is the version code, 2.3 at the time of this writing). The SPHERE header is a free-format 1024-byte ASCII data structure that is prepended to the waveform data. The header is composed of a fixed-format portion followed by collection of one-line attribute fields. The two-line fixed basic header consists of the ASCII strings

```
NIST_1A
1024
```

The first line specifies the file type (a magic number of sorts) and the second line gives the header length. Each of these lines are space-padded to be 8 Bytes long (including the new-line) and are structured to identify the header as well as allow those who do not wish to read the subsequent header information to programmatically skip over it.

The remaining variable-format portion is composed of lines that have a field name, a field format, and a data value. The field formats are pre-defined as string tokens that identify the value’s type and length in Bytes, e.g., “-i2” for a 16-bit integer or “-s8” for an 8-Byte string. Fields are defined for most of the common header data seen above, plus a few interesting additions. The single-token line “`end_head`” marks the end of the active header; the remaining header space (up to 1024 Bytes) is ignored.

Example

A sample NIST SPHERE header is shown in Table 9. Note the standard fields for the utterance identifier, database access keywords, version control, and the flexible sample formats.

Field Name	Format	Data String	Comment
NIST_1A			“Magic number”
1024			Header length
database_id	-s5	TIMIT	Keyword field
database_version	-s3	1.0	Version string
utterance_id	-s8	aks0_sa1	Word ID
channel_count	-i	1	mono
sample_count	-i	63488	Number of samples
sample_rate	-i	16000	Rate in Hz
sample_min	-i	-6967	Minimum sample value
sample_max	-i	7710	Maximum sample value
sample_n_bytes	-i	2	Bytes per sample

sample_byte_format	-s2	01	Pre-defined for linear integers
sample_sig_bits	-i	16	16-bit resolution
end_head			end-of-header token
(empty space)			Sound data starts at offset 1024

Table 9: NIST SPHERE format header example

There are a large number of other field types that can be used in SPHERE files. These include “recording_environment” and “microphone” (strings describing the conditions under which the sound was recorded), “recording_date,” “recording_time,” “sample_coding” (a string giving the sample format, compression, emphasis, etc.), “sample_checksum” (for consistency checking), and “sample_max” (per-channel).

API Support

The standard NIST SPHERE API library (available from the ftp site mentioned above) includes a comprehensive collection of platform-independent C-language functions that create and manipulate the fields of this file header, and the package includes utility programs such as `h_add`, which adds a header to a file of raw sample data, storing the result in a new file, and `h_strip`, which strips the header from a SPHERE file.

Discussion

The SPHERE format allows flexible annotation of sound files, and supports numerous sample formats, including several compression schemes. The inclusion of a version identifier, utterance identifier, and database keyword points to the intended uses of the format for large sound/speech databases. The fixed-size, variable-format header is unique among sound file formats, and poses interesting challenges to API developers (though the existing public domain API seems to alleviate this).

Sound Designer Formats

Digidesign Corp., of Palo Alto, California, USA—producers of the Sound Tools and Pro Tools hardware/software packages (among others)—have defined two sound file formats that they name Sound Designer I and Sound Designer II. Both of these formats are self-describing formats that define a header data structure in Macintosh-specific form, and are widely used by sound-processing applications on this platform. They are , however, very different.

Sound Designer I

Digidesign’s documentation of the Sound Designer I sound file format states that “it is widely used and supported, as evidenced by the many CD ROM discs with sound effects stored in this format. It is primarily used to store monophonic 16-bit short-duration (on the order of seconds) audio samples. We recommend that [developers] support this format for short sounds, but that [they] use Sound Designer II format as a primary format” (Digidesign 1990). Sound Designer I format uses a fixed-format 1336-byte file header that is described by a data structure in which most of the fields are reserved for use by Digidesign’s own applications (i.e., they are documented as “DO NOT USE” in the formal specification). There are the “standard” fields for sample rate, sound file size, loop points and up to ten named marker points (rather than marker regions), as well as various fields for graphical sound editors such as the scale, zoom factor, and offset of the sound within a sound view. The sound sample data is assumed to follow the header structure in the data fork of the sound file. The sample format is given only as the number of bits per sample, so that no compressed or floating-point sample formats are supported. The header has no user-defined or extensible fields other than a 255-byte “comment” field. There is also no header field for the number of channels, i.e., Sound Designer I format can only support monophonic sound files.

Sound Designer II

Digidesign’s Sound Designer II format is, in many ways, a vast improvement over Sound Designer I format, with the one major exception that it is even more Macintosh-specific and non-portable. The format is described in terms of Macintosh “resources,” meaning <ID, type/value, name> triplets that are stored in a special part of the file (the resource “fork”). The sample data is stored in the file’s “data fork.” This makes the files much easier to manipulate for the Macintosh C (or Pascal) programmer, but quite non-portable, as almost all file transfer programs ignore the resource fork and transfer only the data fork of files between a Macintosh and other platforms. Sound Designer II format provides a resource field for the number of channels (ID=1002; type = Pascal-format string, name = “channels”), so that multi-channel sound files are possible, and it supports a number of other new fields for documentation, SMPTE data storage, and measure/beat time data. The sample format is still given as bits-per-integer-sample. There is support for numbered marker points that include a name or comment text, and for various “region” and loop resources.

The use of a flexible header format (using Macintosh resources), means that user programs can extend the format by adding their own resource types (using the resource IDs not used by Digidesign).

The Digidesign documentation still recommends using multiple monophonic Sound Designer II format files for multi-channel operations, rather than a single multi-channel file. One can only assume that some programs can only process monophonic files.

Discussion

Both of the Digidesign sound file formats bear witness to the fact that they are designed to be Macintosh-specific, rather than portable, platform-independent, formats. For example, although the file header data structure for Sound Designer I format is defined in C, all strings in the header a Pascal-format strings (i.e., they are preceded by a “size” byte rather than being null-terminated). The fact that Sound Designer II format is only defined in Macintosh resource format (at least according to the documents this author received from Digidesign), also hinders the porting of these file formats. Nevertheless, these are both widely used formats that are supported by many software tools—Digidesign’s and those from other sources.

Sound Designer I has the interesting feature that it stores the sound file name in the sound file header—something that could make archival quite a bit easier.

The logical separation of the “header” from the sample data is also an important feature of Sound Designer II format, even if it is opaque to the user because of the Macintosh file system’s “fork” mechanism. Few other formats support this feature.

The exact description of the Sound Designer I and II sound file header structures is available upon request from Digidesign, Corp. (in printed, rather than machine-readable format, unfortunately) as (Digidesign 1990).

AIFF Format (Audio IFF) and AIFC (AIFF Compressed)

The interchange file format (IFF) format was developed by Electronic Arts, Inc. and extended to the standard IFF/8VSX by them, and then to audio IFF (AIFF) by Apple Computer, Inc. for storing high-resolution sound and MIDI instrument info (Electronic Arts 1985; Apple 1989). It is also used by Silicon Graphics, Inc. and several professional audio packages on various platforms. An extension called AIFF-C or AIFC supports several data compression schemes (see below). A version of the full AIFF/C specification (Apple 1989) is available from several Internet ftp sites, including ftp.cwi.nl:/pub/audio/AudioIFF1.3.hqx (a BinHex’ed MS-Word file), mitpress.mit.edu:/pub/Computer-Music-Journal/Documents/SoundFiles/AIFF-C.ps.Z (compressed PostScript), and ftp.sgi.com:/aiff-c.9.26.91.ps (PostScript). It can also be found in (Milne and Deatherage 1991). The IFF standard is (Electronic Arts 1985).

Like its relatives IFF and TIFF, AIFF is a “chunked” file format. Chunked files have a flexible format, rather than a fixed header followed by a single data block. There is a small (12-Byte) fixed file header, followed by a variable number of data or other “chunk” blocks in any order. Each chunk starts with a header that has a “type” field (a key or chunk ID—to identify the format of its contents), and a “size” word giving the size of its data (normally in Bytes). AIFF sample files have a “common” chunk (with the same information as a typical sound file header), one or more chunks describing cue points, text comments, copyright information, sampler setup information, or MIDI data, and one sound data chunk. The common chunk normally comes after the fixed header, followed by the other chunks, with the sample data chunk last, though one can write other chunks after the sample data if desired.

Figure 3 illustrates how an AIFF file is constructed. It starts with a header or “form” chunk that includes the file’s magic number (the 32-bit quantity representing the four-character string “FORM”). There is a common chunk that defines the sound sample data format, and one sound data chunk (though they need not be in the order shown).

```
Form Chunk
   Chunk ID = “FORM”
   File size (all chunks)
   Form Type = “AIFF”
Common Chunk
   Chunk ID = “COMM”
   Common chunk data size
   Sample format, rate, etc.
Data Chunk
   Chunk ID = “SSND”
   Sound chunk data size
   Array of sample data
```

Figure 3: Minimal AIFF File Structure

The rest of the file format is flexible and depends on the application that created the file. No more than one sample data chunk is allowed in a file, but any number of cue, text, and other kinds chunks can be used to annotate AIFF sounds. The IFF definition allows for amplitude contour chunks (e.g., attack or decay), and IFF files include four standard annotations: author, name, annotation, and copyright strings. Three chunk types greatly empower AIFF applications: sampler instrument data, raw MIDI data, and application-specific chunks. These will be discussed below.

The C-language data structures common to all AIFF chunks is shown in Figure 4. Every chunk starts with its type ID (32-bits representing an ASCII string of four letters) and 32-bit chunk size. Note the macros for unsigned data types in Figure 4, as well as the use of 80-bit extended-precision floating-point numbers (which are required by the API implementor). The Pascal-style string type is a character array preceded by its length (rather then being null-terminated as in C). This has the advantage of being faster to skip over (without having to scan it for its length).

```
/* Type definitions used in AIFF structures */
typedef char[4] ID;             /* Magic numbers are 4-char IDs */
typedef unsigned long ULong;    /* Used for large numbers */
typedef extended float XFloat;  /* 80-bit floats for sample rates */
typedef unsigned char UChar;     /* Used for raw data pointers */
typedef unsigned short UShort;   /* Used for long counters */
typedef short MarkerId;          /* Used for marker numbers */

   /* AIFF uses Pascal-style strings with their lengths prepended. */
typedef struct {
   UChar ckSize;     /* String length in Bytes (<= 255) */
   char  ckData[];   /* String data (not null-terminated */
} PString;

   /* General AIFF chunk structure */
typedef struct {
   ID    ckID;       /* Chunk ID string, 32-bits = 4 characters */
   long  ckSize;     /* Chunk data size in Bytes */
   char  ckData[];   /* Chunk data--format varies according to type */
} Chunk;
```

Figure 4: Basic AIFF declarations and chunk structure

The form header that is included at the beginning of all AIFF files is shown in Figure 5. The first ID identifies the file as some kind of IFF chunked file, the second gives the “local” format—AIFF or AIFC. The rest of the chunk data (`ckSize` Bytes) follows this header.

```
   /* AIFF/C FORM Chunk--Global File “Wrapper” */
typedef struct {
   ID    ckID;       /* ID = “FORM” */
   long  ckSize;     /* Size of file’s chunks in bytes */
   ID    formType;   /* ID = “AIFF” or “AIFC” */
   char  chunks [];  /* 1 or more chunks in any order */
} FORM_Header;
```

Figure 5: AIFF form chunk structure

The AIFF Common chunk corresponds to the sound file header of the formats described above. Its fields include the sample rate, sample format and precision, the number of channels, and the size of the sample data block. As shown in Figure 6, the chunk header is followed by these fields, using an integer to store the sample size in bits (8-24) of (assumed linear integer) samples, and an 80-bit float for the sample rate.

```
/* AIFF/C Common Data Chunk--Sound Data Properties */
typedef struct {
   ID ckID;                /* = “COMM” */
   long  ckSize;           /* = 18 Bytes in chunk*/
   short numChannels;      /* 1, 2, or more */
   ULong numSampleFrames;  /* number of full frames */
   short sampleSize;       /* size in bits / sample */
   XFloat sampleRate;      /* rate as an 80-bit float */
} CommonChunk;
```

Figure 6: AIFF Common Data Chunk

One sound data chunk is allowed per AIFF file; its format is defined in Figure 7, which shows the standard chunk header and variable-type sample data array. The offset and block size are normally 0, but can be used to indicate silence at the beginning of a sound or to optimize data buffering.

```
   /* AIFF/C Sound Data Chunk Format */
typedef struct {
   ID ckID;          /* = “SSND” */
   long  ckSize;     /* size of data */
   ULong offset;     /* start offset of block 1 in Bytes */
   ULong blockSize;  /* optional block size in Bytes */
   Void  soundData[];/* interpreted according to sample format */
} SoundDataChunk;
```

Figure 7: AIFF Sound Sample Data Chunk

The other chunks AIFF supports allow one to define named marker points into a file’s sample data, to add several kinds of comments to the file, or to attach a special data structure with MIDI instrument voicing data for a sampler. The data structure used for markers or cue points is defined in Figure 8, followed by the format for marker chunks. Note that a single marker chunk can hold on to many named markers.

```
      /* AIFF/C Data Structure for marker points */
typedef struct {
   MarkerId    id;         /* Marker number */
   ULong       position;   /* Sample offset */
   PString     markerName; /* Name string */
} Marker;

      /* Marker chunk with array of markers */
typedef struct {
   ID          ckID;       /* = “MARK” */
   long        ckSize;     /* Size of data in Bytes */
   UShortnumMarkers; /* How many markers in chunk */
   MarkerMarkers[];  /* Marker data structures */
} MarkerChunk;
```

Figure 8: AIFF Marker Structure and Chunk

One or more comment strings—that also keep a time-stamp and can be “associated” with a specific marker ID—can be encapsulated in a special comments chunk. The two data structures for this are defined in Figure 9. Note that this chunk type is only used for uninterpreted comments; other chunk types exist for specific annotations such as author, copyright, etc. (see below).

```
   /* AIFF/C Comment Data Structure--with Marker ID and time-stamp */
typedef struct {
   ULong timeStamp;        /* When was comment created */
                           /* (in sec since 1/1/1904) */
   MarkerID    marker;     /* Optional marker comment applies to */
   UShortcount;      /* Size in Bytes of text */
   char        text[];     /* Comment string */
} Comment;

   /* Comments Chunk--can hold onto many comments */
typedef struct {
   ID          ckID;       /* = “COMT” */
   long        ckSize;     /* Size of data in Bytes */
   UShortnumComm;    /* How many comment records */
   Comment     comments[]; /* Array of comments */
} CommentsChunk;
```

Figure 9: AIFF Comment Structure and Chunk

The MIDI voice information mentioned above is stored in an Instrument chunk, shown in Figure 10. It has fields for the “true” frequency of the sample (as a MIDI key number with de-tuning in cents), and the suggested high and low transposition range (as key numbers) and dynamics (as MIDI key velocities) of the sample. The default gain is a dB value by which to scale the sample to achieve full amplitude (very useful).

An instrument chunk can also define how a sample is to be extended in length if it is sustained beyond its stored duration. Begin and end points can be given for a section of the “steady-state” portion of the sample so that a sampling synthesizer can “loop” through it, repeating it as necessary before playing the “release” or “decay” portion of the sample. The data structure for loops is defined in Figure 10; it refers to two marker IDs (by number), assumed to be defined in a marker chunk.

```
/* AIFF/C Loop data structure--uses numerical marker IDs */
typedef struct {
   short       playMode;   /* Loop play mode */
   MarkerId    beginLoop;  /* Cue marker where loop starts */
   MarkerId    endLoop;    /* Cue marker where loop ends */
} Loop;

      /* MIDI Instrument Data Chunk--sampler voicing */
typedef struct {
   ID ckID;                /* = “INST” */
   long  ckSize;           /* = 20 Bytes */
   char  baseNote;         /* Frequency of sample as a MIDI Key */
   char  detune;           /* Fine-tuning in Cent */
   char  lowNote;          /* Lowest note to transpose sample to */
   char  highNote;         /* Highest note to transpose sample to */
   char  lowVelocity;      /* Lowest velocity to play sample at */
   char  highVelocity;     /* Highest velocity to play sample at */
   short gain;             /* Default gain in dB */
   Loop  sustainLoop;      /* Optional sustain loop record */
   Loop  releaseLoop;      /* Optional release (decay) loop record */
} InstrumentChunk;
```

Figure 10: AIFF Loop Structure and MIDI Instrument Chunk

Other chunk types are defined to support storing uninterpreted “raw” MIDI data, AES channel data for live recordings (e.g., emphasis information), four kinds of text annotation—name, author, copyright, annotation—and free-form “application-specific” data.

AIFC defines header formats for µ- and A-law, ADPCM, and other compression schemes.

Example

Table 10 shows an AIFF file structure that is “populated” with data; the field sizes in Bytes are given, followed by the field names, values, and a comment. This file has 2 sec of 44.1 kHz mono sound, and defines two markers named “start-loop” and “end-loop.” The MIDI Instrument chunk shows data that would allow this to be used by a digital sampler.

Size	Name	Value	Comment
4	ckID	“FORM”	“Magic number” token
4	ckSize	176518	Size of file in bytes
4	formType	“AIFF”	(or “AIFC”) Data type
4	ckID	“COMM”	Common chunk header

4	ckSize	18	Bytes in chunk
2	numChannels	2	Stereo
4	numSampFrames	88200	Number of full frames
2	sampleSize	16	Size of samples in bits / sample
10	sampleRate	44100.000000	Rate as an 80-bit floating-point number
4	ckID	“MARK”	Marker chunk header
4	ckSize	36	Size of data in Bytes
2	numMarkers	2	How many markers
34	Markers	(see below)	Marker data structures
2	id	1	Marker number
4	position	1000	Sample offset
12	markerName	10, “start loop”	Name string
2	id	2	Marker number
4	position	60000	Sample offset
10	markerName	8, “end loop”	Name string
4	ckID	“INST”	Instrument chunk header
4	ckSize	20	Bytes in chunk
1	baseNote	60	Frequency of sample as a MIDI Key
1	detune	0	Fine-tuning in Cent
1	lowNote	54	Lowest note to transpose sample to
1	highNote	68	Highest note to transpose sample to
1	lowVelocity	40	Lowest velocity to play sample at
1	highVelocity	80	Highest velocity to play sample at
2	gain	6	Default gain in dB
6	sustainLoop		Optional sustain loop record
6	releaseLoop		Optional release (decay) loop record
4	ckID	“SSND”	Sound data chunk header
4	ckSize	176408	Size of data chunk in Bytes
4	offset	0	Start offset of block 1 in Bytes
4	blockSize	0	Optional block size in Bytes
88200	soundData	. . .	Samples interpreted according to format

Table 10: AIFF/C File structure

API Support

There are simple AIFF API libraries available on several platforms. Functions for creating and reading AIFF headers are also provided with several public domain software packages (see below). On the Macintosh, AIFF files are handled properly by many system tools using the Apple Finder’s file type information rather than the file header. Most Macintosh-based sound processing tools support AIFF, e.g., Digidesign's SoundTools, ProTools and AudioMedia, Symbolic Sound's Kyma package, or Studer's Dyaxis system. Software is provided with the SGI Indigo to record and play AIFF/C files, and to convert from NeXT/Sun or raw data formats to AIFF; the SGI also provides an API to AIFF files as part of their multimedia library. The average level of these tools is "lower" than the support provided to support the NeXT/Sun format; for example, there is no standard tool on the Macintosh or SGI Indigo platform to add comments to sound files after their creation or to list them in a convenient format.

Discussion

One advantage of chunked formats is that they are easily used to describe in-memory storage as well as stream- or file-based interchange formats. The differences between the various chunked file formats is in how they represent sound data, which annotational chunk types they support, and whether or not there can be more than one data chunk in a file. AIFF/C allow only one data chunk, comments chunk, marker chunk, etc. per file, so that the added flexibility of letting these structure be variable size is largely taken away..

One disadvantage of chunked formats is that the functions to read them are generally more complicated than for fixed-header formats. A program must often read the entire file chunk-by-chunk (possibly skipping over the sample data) to find all relevant application data before beginning any processing. The ability to store annotation data after the sample data chunk is an advantage if one wants to add a small comment to a recorded sound without having to copy the sample data on-disk.

The flexibility and interoperability of AIFF—its use on samplers, Macintosh tools, SGI Indigo, and elsewhere—is its most important feature. It can be used to move sounds between Macintosh and Indigo applications, e.g., Kyma, Csound (the language), Dyaxis, and ProTools. Software packages exist that use AIFF for everything from sample management to sound archival.

AIFF allows only integer or μ-law sample formats, and no floating-point formats can be described with the standard, unfortunately (a major problem for some users).

It is a minor annoyance that, although markers do have names, they must be referred to by number in loops, e.g., in the MIDI instrument chunk.

AIFF also defines sample frame formats for 1-, 2-, 3-, 4-, and 6-channel sound.

RIFF WAVE (or WAV) file format

RIFF is a file interchange format developed by Microsoft and IBM; it is very similar in spirit and functionality to IFF, but is not compatible with it (and it uses a different byte ordering ["little-endian"]). WAVE (or WAV after the three-character file name extension its files use) is RIFF's equivalent of AIFF; its inclusion in

Microsoft Windows 3.1 has secured its place as a “standard” to be addressed by sound file system implementors. The official specification of WAVE can be found in (Microsoft 1992), which is also available as ftp.cwi.nl:/pub/audio/RIFF-format.

WAVE is described as a chunked file that has a header (AIFF’s form chunk), a format chunk (common chunk in AIFF), and can include a variety of other optional chunks as shown (in pseudo-BNF form with ANSI-C-format comments) in Figure 11. The chunk types of RIFF/WAVE correspond quite closely to those of AIFF, and the file structure is the same. Each file has a small header (the “RIFF/WAVE” tags), a format (common) chunk with sample attributes, and a wave (sound sample data) chunk. The other chunks are optional and generally come after the common chunk and before the sample data chunk. The parentheses used in the RIFF specification are used to denote the inclusion of one chunk within another; for example, the line “RIFF(“ means “place a RIFF chunk header here with an ID (tag) and the size of all the data included up to the matching ‘)’.” As in AIFF, the “outer wrapper” on a file (the AIFF form chunk or RIFF chunk header in Figure 11) “encapsulates” all the chunks within it, and these “inner” chunks can encapsulate structures of their own, as in the example of a marker chunk containing many marker records.

```
// RIFF/WAVE File Structure
<WAVE-form> ->
 RIFF(                    // File (FORM) header; type RIFF
                          // (includes size field as in AIFF/C
   “WAVE”           // Local header; type WAVE
      <fmt-ck>            // Format (common) chunk
      [<fact-ck>]         // Fact chunk
      [<cue-ck>]          // Cue (marker) points
      [<playlist-ck>      // Play list (MIDI instr.)
      [<assoc-data-list>] // Associated (annotation)
      <wave-data>         // Wave (sample) data
   )
```

Figure 11: RIFF WAVE Format File Layout

The WAVE format chunk specifies the format of the sample data chunk. The format chunk is defined as shown in Figure 12. Note that is contains a collection of “common” fields, and also allocates space in the format chunk for additional, format-specific data fields. The `wFormatTag` field is a sample format flag similar to the one used in the NeXT/Sun format introduced above. The legal values for this are defined in a header file, and include linear integers, μ- and A-law, ADPCM, and other schemes including “Microsoft PCM,” “IBM μ-law,” “IBM ADPCM,” etc. (??). The data types `WORD` and `DWORD` used here correspond to 16- and 32-bit integers (`short` and `long` in non-MicroSoft C). The type `FOURCC` is used for the 32-bit, 4-character identifier constants used in IFF/RIFF files.

```
// RIFF/WAVE Format chunk header
<fmt-ck> ->
   fmt(                          // Format chunk header and size
      <common-fields>            // Standard fields
      <format-specific-fields>   // “Other”
)

   // RIFF/WAVE format chunk common field data structure
<common-fields> ->
   struct {                      // Common fields chunk
      WORD     wFormatTag;       // Format flag
      WORD     wChannels;        // Number of channels
      DWORD    dwSamplesPerSec;  // Sampling rate
      DWORD    dwAvgBytesPerSec; // For buffer estimation
      WORD     wBlockAlign;      // Data block size
   }
```

Figure 12: WAVE format chunk

The <wave-data> contains the waveform sample data. It is defined in Figure 13, which shows that a data chunk may consist of a single data block, or of a list of data and silence blocks.

```
   // RIFF/WAVE wave data chunk--1 or more data blocks
<wave-data> -> { <data-ck> : <wave-list> }

   // Raw data chunk--header and data
<data-ck> ->
   data( <wave-data> )

   // Wave list chunk--header, data, and silence
<wave-list> ->
   LIST(“wavl” {                 // Wave list chunk header
      <data-ck>            // Wave samples
      <silence-ck>   }           // Silence
      ...                  // More data or silence
   )

   // Silence chunk--length in samples
<silence-ck> -> slnt( DWORD dwSamples )
```

Figure 13: RIFF/WAVE Wave data chunk format

The cue-points chunk identifies a series of positions in the waveform data stream. The cue chunk is just like an AIFF marker chunk, but has no name string, and the data structure used for a cue point is different. Its definition is given in Figure 14. The interpretation of the various sample, file, and block offsets depends on the disk and software being used, and can be rather complex (Microsoft 1992).

```
// Cue point data structure
<cue-point> ->
   struct {
      DWORD dwName;              // Cue point name
      DWORD dwPosition;          // Sample offset of cue point
      FOURCC fccChunk;           // ID of chunk with cue point data
      DWORD dwChunkStart;        // File offset of cue point
      DWORD dwBlockStart;        // Block offset of cue point
      DWORD dwSampleOffset;      // Sample offset within the block
   }
```

Figure 14: RIFF/WAVE cue point structure

The play list chunk specifies a play order for a series of cue points. It can be used to loop through selected sections of a sound various numbers of times. The play list contains a number of play segments, whose data structure is given in Figure 15.

```
// RIFF/WAVE play segment structure
<play-segment> ->
   struct {
      DWORD dwName;        // Cue point number
      DWORD dwLength;      // Loop length
      DWORD dwLoops;       // Number of repetitions
   }
```

Figure 15: RIFF/WAVE play segment structure

The associated data list provides the ability to attach information such labels to sections of the waveform data stream. It can also store miscellaneous annotational information like AIFF’s annotation chunk.

Examples

The first line in Figure 16 shows the compact notation for a WAVE file with 8-bit monophonic PCM samples at an 11025 Hz sampling rate. The compact notation used in Figure 16 shows each chunk as an expression of the form `type(contents)`. The header format of a 20-bit monophonic PCM file with 44.1 kHz sampling rate and some annotation information (the name of the sound in a special name chunk) is shown in the second example of Figure 16.

```
   // Simple example of a RIFF/WAVE file
RIFF( “WAVE”
   fmt(1, 1, 11025, 11025, 1, 8)
   data( <wave-data> ) )

   // RIFF/WAVE example with an info chunk for the song name
RIFF( “WAVE”
   info(name(“Ach, Du Lieber” Z))
   fmt(1, 1, 44100, 132300, 3, 20)
   data( <wave-data> ) )
```

Figure 16: RIFF/WAVE Examples

API Support

The MS-Windows support libraries for WAVE format include basic file I/O functions and support for reading and writing the mark-up chunks. No APIs that support the processing WAVE files on other platforms are known to the authors (other than those included in the sound file conversion packages described below).

Discussion

The RIFF/WAV format must be compared to AIFF in evaluating or discussing it. Although AIFF predates it, RIFF is not as flexible as AIFF, aside from the fact that WAVE supports a few more sample formats than AIFF. One should also note that RIFF/WAVE is really very similar to AIFF—a slight non-compatible modification by Microsoft of a well-known and stable multi-platform standard.

The provision of the play list chunk, which allows one to play a number of segments in any order repeating any number of times, is very interesting. The play list, however, uses its own descriptors for play segments, rather than cue point structures.

There is still no support for floating-point samples, nor for other compression schemes than those mentioned above (PCM, exponential laws, and ADPCM).

The csound, IRCAM, BICSF, and EBICSF Sound File Systems

Several chunk-oriented (or flexible-format, variable-size header, depending on how one defines “chunked”) sound file systems derived from D. Gareth Loy’s csound package evolved independently during the 1980s; they were then merged in the Berkeley/IRCAM/CARL sound file (BICSF) system (Loy et al. 1988). This system is also still seen referred to as the “IRCAM” sound file system. Unlike csound, BICSF uses the host computer local file system (typically the Berkeley UNIX fast file system or Macintosh HFS) for sample storage, rather than placing sound file headers and sample data in different files on separate disk partitions.

The “standard” BICSF system uses a fixed-size file header with magic number, sample rate (as a floating-point number), number of channels (1, 2, 4, or more), and sample format fields (supporting floating-point samples). This data structure is shown in Figure 17. Several optional “sound file code” fields are defined for encoding (e.g.,) the maximum amplitude per channel, the sound’s amplitude envelope, or a comment text. The structure of BICSF code fields is very similar to AIFF’s optional chunks—each has a header with its type and size, which is followed by a data structure that is interpreted according to the block’s type. A collection of these code fields may follow the BICSF header and precede the file’s single sample block, as in AIFF. Unlike AIFF, no code fields may follow the sample data chunk. The exact list of supported file codes has varied and expanded in recent BICSF implementations. There are standard chunks for annotations, as well as for storing the maximum amplitude per channel (with its sample offset), and for attaching amplitude and pitch envelopes to files.

```
   /* CSOUND/BICSF File Header--fixed fields + codes = 1024 Bytes */
typedef union {
   struct {
      int sf_magic;        /* magic number--may vary (see text) */
      float sf_srate;      /* Sample rate--32-bit floating-point */
      int sf_chans;        /* Number of channels--1, 2, or 4 supported */
      int sf_packmode;     /* Sample format--many supported */
      char sf_codes[];     /* variable number of sf-code “chunks” */
   } sfinfo;
   char  filler[1024];     /* standard header size = 1024 Bytes */
} SFHeader;
```

Figure 17: BICSF sound file header

The BICSF file header was “merged” with the NeXT/Sun format’s header structure by Paul Lansky in his CMix system (Pope 1993). This system and its variants are referred to here as Extended-BICSF or EBICSF. Versions of it are included in the CMix (Lansky 1994) and MODE (Pope 1993) software distributions (in C and Smalltalk-80, respectively). The “merge” of the BICSF and NeXT/Sun headers means simply that the first 28 Bytes of an EBICSF file hold a legal NeXT/Sun file header (defined in Figure 1 above), that has its dataLocation—the offset from the start of the header to the first data Byte—set to 1024 (or a multiple thereof). The data after the 28th Byte is ignored by NeXT/Sun applications, most of which correctly use the dataLocation offset to jump to the samples for their operations. This data after the first header is a BICSF-style header structure, with fixed fields for the magic number, rate, format, and number of channels, followed by a variable number of code blocks (chunks). Figure 18 shows the basic format of the EBICSF file header.

The NeXT/Sun SNDSoundStruct data structure includes the fields introduced above, and is followed by the five standard BICSF fields and some number of code fields.

```
      /* EBICSF Sound file header structure */
   typedef union {
      struct {
         SNDSoundStruct N_Header;   /* 28-byte NeXT/Sun header */
         int sf_magic;              /* BICSF magic != NeXT magic */
         float sf_srate;            /* Floating-point sample rate */
         int sf_chans;              /* Number of channels */
         int sf_packing;            /* Pre-defined storage format */
         char sf_codes[];           /* Any number of code fields */
      } sfinfo;
      char  filler[1024];           /* Default = 1024 Bytes; may be */
                                    /* larger (n * 1024) if needed */
   } SFHeader;
```

Figure 18: EBICSF Format File Header

One interesting feature of BICSF is that there are several magic numbers that are used to differentiate different kinds of sound files. A sound file can also be designated as a “link” or “virtual” file if it contains pointers into the sample data of another file rather than having its own samples. A file could, for example, represent the samples of another file between two specified cue points. A file that has the magic number denoting a “composite” file is actually like a mixing script; it can define several segments of other (virtual) files to be played simultaneously and/or sequentially. Note that this places special requirements on an EBICSF-compatible play program; it must also be a real-time disk-to-DAC mixer.

Note that the sample rate is stored as a 32-bit floating-point number—more accurate than the integer values used in the NeXT/Sun and RIFF/WAVE formats, but less than AIFF’s “exaggerated” 80-bit precision. The pre-defined sample formats include several sizes of integers, 32- and 64-bit floating-point numbers, μ- and A-law, and ADPCM. Note also that this header is at least 1024 Bytes long, even if there are no `sf_codes` data chunks included; it can indeed be much longer (up to about 32 kBytes in practice) if several envelope functions are encapsulated in the header.

Just like AIFF chunks, all BICSF code fields have an 8-Byte header that gives their type and data size. There are data structures defined for the various types of code field, several of which are shown in the following Figures. Note that the 8-Byte chunk header is not included in the code data structures; it is assumed to be discarded by the reading function.

The SFMaxAmp data structure defined in Figure 19 is used to denote the maximum amplitude per channel of a sound file. It holds onto two arrays of four values each (by default), which are the maximum absolute sample values for each channel, and the sample offsets where they were found. A time tag tells when the maxima were last measured (in case they are “stale”).

The cue region structure is also shown in Figure 19; it is composed of a name string and the starting and ending sample locations that define a region of the file’s sound data. This way, one cue region can serve as a loop record or a pointer to a “sub-sound” held onto in a related “virtual” sound file.

```
/* Max amp code--always has data for 4 channels */
typedef struct {
   float value[4];         /* Max sample value of channel */
   long  samploc[4];       /* Frame offset of sample position */
   long  timetag;          /* Time last updated */
} SFMaxAmp;

   /* Cue point (like an AIFF mark but has start/stop points) */
typedef struct {
   char  cuename[64];      /* the cue region’s name */
   long  beginsamp;        /* starting sample index */
   long  endsamp;          /* ending sample index */
} SFCue;
```

Figure 19: EBICSF Standard Code Formats

An important feature of EBICSF is its support for links between sound files—virtual and composite files. The data structures used for these files are shown in Figure 20. A sound file link is a file with a BICSF header and no samples; it has an `SFLink` code in it that points to a set of samples in another file. In the case of a link, the file is named, and the starting and ending samples are specified, as shown in the first data structure in Figure 20. A virtual sound file is like a link, with the exception that a virtual file code field gives the “target” file name and the name of a cue point in that file (assumed to be defined in the header of the target file). Some applications will want to use links (there the link file has the sample offsets) while others will prefer to use virtual files (where the “target” file defines the cue region).

Several kinds of codes exist in EBICSF for storing amplitude and pitch envelopes in sound files. The various envelope chunks have headers that define their type (e.g., an RMS amplitude derived with a 256-sample sliding window stored as 1024 short integers, or a pitch envelope reduced to a 32-point break-point envelope), and there are data types for envelope data arrays as well as break-point envelopes (with x- and y-values of points that will be interpolated between).

The last example in Figure 20 shows the record structure used for "composite" files. These are actually more like mixer data structures than sound files per se. A composite file has a number of `SFComposite` records that describe when to play a segment of another file, name the file and cue region, and provide an envelope function for fading in and out. As mentioned above, the play program that reads these is actually performing the function of a real-time mixer.

```
   /* Link file pointer with sample region in the "target" file */
typedef struct {
   char reality[64];       /* The "real" file's name */
   long startsamp;         /* Starting sample index */
   long endsamp;           /* Ending sample index */
} SFLink;

   /* Cue-based virtual sound file--points to a named cue region */
typedef struct {
   char reality[64];       /* The "real" file's name */
   char cuepoint[64];      /* Named cue point in the file */
} SFVirtual;

   /* RMS envelope function--one kind uses 1024 short integers */
typedef struct {
   char envname[64];       /* The envelope function's name */
   short envdata[1024];    /* Envelope data--1024 integers */
} SFRMSEnvelope;

   /* 2-D Cartesian Point Data Structure */
typedef struct {
   float x, y;             /* Floating-point X and Y coordinates */
} Point;

   /* Pitch envelope function--one kind uses 32 breakpoints */
typedef struct {
   char envname[64];       /* The envelope function's name */
   Point envdata[32];      /* Envelope data--32 breakpoints */
} SFPitchBPEnvelope;

   /* Composite Segment--Mix Record with file and envelope */
typedef struct {
   long start;             /* Sample at which to begin this element */
   SFVirtual target;       /* Where are the samples? */
   SFRMSBPEnvelope;        /* Breakpoint RMS envelope */
} SFComposite;
```

Figure 20: Somew of the EBICSF Standard Code Formats

EBICSF also provides several types of comment structures—roughly equivalent to IFF’s author, copyright, etc., chunks—as well as extended text annotations, such as the name of a sound file’s “parent” (the sound from which it was derived), its “script” (DSP program used to create it), one or more keywords, and a multi-level version number.

Example

Example EBICSF file headers are shown in Tables 11 and 12 and Figure 21. Table 11 shows a header that corresponds roughly to the file AIFF example header shown in Table 10. The first 28 bytes contain the same data structure as the NeXT/Sun file header shown in Figure 1. After this comes the BICSF version of the basic characteristics (with floating-point sample rate), and a number of code fields (data chunks) for the maximum amplitude arrays, a simple comment, two named cue regions, a pointer to the file’s “parent,” and even the moving-window RMS amplitude envelope of the sound. Assuming this is being read in from disk, the contiguous sample data starts at the end of this 3072-Byte header. Each chunk has (like with AIFF) a 32-bit ID (whose possible values are defined as macros on the EBICSF header file), and gives the size of the chunk’s data (not including the 8-byte header).

Size	Name	Value	Comment
28	SNDSoundStruct	See Figure 1	Standard NeXT/Sun header
			(NB: data offset is = 3072 for this file.)
4	sf_magic	107364	BICSF magic (not the same as NeXT magic)
4	sf_srate	49152.00	Floating-point sample rate—BICSF “hi-fi”
4	sf_chans	2	Number of channels
4	sf_packing	4	Pre-defined storage format for 32-bit floats
4	code	“MAXA”	MaxAmp chunk ID
4	bsize	36	Code chunk size in Bytes
4*4	values	[32767, ...]	Max sample values of 4 channels
4*4	samploc	[10000, ...]	Frame offsets of sample positions per channel
4	timetag	778065725	Time last updated (UNIX-style msec counter)
4	code	“COMM”	Comment chunk ID
4	bsize	512	Comment chunk size in Bytes
512	comment	“Comment . . .”	Comment of length 512 by default
4	code	“CUE”	Cue region chunk ID
4	bsize	72	Cue chunk size in Bytes
64	cuename	“Sustain Loop”	the cue region’s name

4	beginsamp	15000	Starting sample index
4	endsamp	33000	Ending sample index
4	code	"CUE"	Cue region chunk ID
4	bsize	72	Cue chunk size in Bytes
64	cuename	"Release Loop"	The cue region's name
4	beginsamp	33000	Starting sample index
4	endsamp	49152000	Ending sample index
4	code	"PRNT"	Parent file chunk ID
4	bsize	64	Parent chunk size in Bytes
64	parent	"Parent.snd"	The "parent" file's name
4	code	"ENV"	Envelope chunk ID
4	bsize	2112	Code chunk size in Bytes
64	envname	"RMS Env"	The envelope function's name
2048	envdata	(0, ...)	Envelope data—1024 short ints by default
(empty space up to 3072 Bytes)			
393216	data	. . .	Sample data starts after 3072 bytes

Table 11: A Filled-in Standard EBICSF Sound File Header

Table 12 illustrates a virtual sound file; this has the same header format as the previous example, but uses a different magic number (so that utility programs that cannot process it will ignore it). After the standard header comes a virtual file chunk that identifies another file by name ("words_11b.snd") and a named cue region within that file ("third word"). It is assumed that this "target" file has a cue region by that name.

Size	Name	Value	Comment
28	SNDSoundStruct	See Figure 1	Standard NeXT/Sun header
			(NB: data offset is = 1024 [but is irrelevant].)
4	sf_magic	107415	Special EBICSF magic for virtual files
4	sf_srate	44100.00	Floating-point sample rate
4	sf_chans	2	Number of channels
4	sf_packing	2	Pre-defined format for 16-bit integers
4	code	"VIRT"	Virtual chunk ID
4	bsize	128	Virtual chunk size in Bytes
64	reality	"words_11b.snd"	The "real" file's name
64	cuepoint	"third word"	Named cue point in the file

Table 12: An EBICSF Virtual Sound File Header

The text shown in Figure 21 is the MODE system’s print-out for an EBICSF sound file that includes several more of the extended EBICSF code chunks. Note that in Smalltalk, symbols are printed with a preceding hash-mark (#), strings are written within single quotes, and interval objects are printed as ranges, e.g., (271 to: 29740). The comments within double quotes were added after the fact.

```
name:    ‘snd/AllGatesAreOpen/Michael/slow_c/4a.snd’
rate:    44100.0
channels:1
format:  #linear16Bit                 “Symbol for format”
duration:1.42367 sec
maxAmp:  Dictionary (1->10700->23345)  “Ch 1; Value 10700; index 23345”
size:    126592 bytes
modified: 94 July 25 : 5:05:22 pm
text:    “Droem och vaka”             “Transcription”
version: 1.2.1.1                      “Multi-level version number”
comment: ‘Transposed down a minor third and slowed down 35%’
cueList: Dictionary (#droem -> (271 to: 29740),    “List of cue regions”
                     #och -> (31815 to: 41035),
                     #vaka -> (41036 to: 62755))
script:  ‘pv 44100 1024 8192 128 173 0.82 0 0 -i’  “Vocoder script”
parent:  ‘snd/AllGatesAreOpen/Michi_1/src/4a.snd’  “Name of parent file”
envelope: (an array of 1024 16-bit integers for RMS envelope)
samples: (sample data array)
```

Figure 21: Smalltalk format of an EBICSF Sound File Header

API Support

Both the Cmix (Pope 1993) and MODE (Pope 1992) packages provide API libraries for building programs that use EBICSF sound files. The Cmix version includes numerous UNIX shell utilities for manipulating sound files. Note that most of the native NeXT or Sun utilities will work with EBICSF just as with NeXT/Sun format files, so that no special play, record, list or other utilities are needed for “normal” EBICSF sample files. The Cmix utilities include programs, for example, to add comments to sound files.

The MODE system’s sound file library supports all the EBICSF facilities mentioned above (including virtual and composite files), and has graphical sound file browser/editors and a general-purpose object-oriented API for sound processing and I/O (in Smalltalk-80, making it portable among many platforms: UNIX, Macintosh, and IBM PC-Compatible).

Discussion

The BICSF allows for complex annotation with script, transcription, comment, version, envelopes, and other data attached to a sound file. Some users prefer this data to be stored in separate files (with related names, for example). The MODE version of EBICSF takes a rather extreme position on this point (putting everything—even multiple envelopes—in one file), partly because it uses these files as the storage format for an object-oriented hypermedia sound database.

The support for named cue regions (rather than numbered marker points), for virtual, and composite files, and for fragmented sample data (using composite files) is an important plus. These features place EBICSF somewhere between sound file and mixing script formats. There is little support for MIDI sampler data in csound, BICSF, or EBICSF.

EBICSF is upward-compatible to the native sound file system on platforms that use (or at least support) the NeXT/Sun system—of major importance on those platforms with good built-in support for it. The standard tools on these platforms can, however, not handle link, virtual, or composite files.

Other Formats

There are many other sound file storage formats in use for other applications that were deemed not suitable for use in computer music and have therefore not been included in our in-depth focus above. We will mention them here and give references to more detailed documentation. There are also numerous closely related areas that use very similar formats, e.g., for sequencer song storage, for mixing scripts, for sound database archival.

8-bit Formats

The Creative Voice (VOC) file format is widely used for sample storage, but supports only 8-bit samples. It is described in the FAQ file from which this article is excerpted. Apple HCOM files contain Huffman compressed data that yields 16 kHz 8-bit unsigned data after decompression.

Sampler-oriented formats

There are several formats designed for use storing sample banks from sampling synthesizers. These generally support a limited number of sample formats and rates, and allow several (possibly many) sounds to be stored in the same file.

The Amiga MOD Format is used by many samplers and sample management utilities on the Atari ST and Macintosh platforms. It offers good flexibility for use with MIDI samplers, but is not really intended as a general-purpose sound file system. A detailed description of it can be found in the Internet file ftp.cwi.nl:/pub/audio/MOD-info.

Sound Interchange via Electronic Mail

The standard interchange format used for electronic mail distribution has been extended to allow the portable inclusion of sounds in mail messages. The proposed MIME (Multimedia Internet Mail Extension) standard includes an interesting “interchange format” for audio data; it describes a family of transport encodings and structuring devices for electronic mail. This is an extensible format, and initially standardizes a type of audio data dubbed “audio/basic,” which is 8-bit μ-law data sampled at 8 kHz.

Proprietary formats

Many hardware/software packages define their own formats. Digidesign SoundDesigner, Studer Dyaxis MacMix, and other tools incorporate proprietary sound file formats.

Comparison

According to the data encoded in the sound file header, and where and how the sample data is stored on the disk, one can make a taxonomy of sound file systems according to the criteria shown in Table 13 below. One could easily fill in this table for each of the sound file systems presented above.

- Native or proprietary disk partitions
- Headerless or self-describing files
- Fixed- or flexible-format header
- One or multiple sample data chunks
- Loops and Cues
    - Marker points or named cue regions
- MIDI sampler data
    - What data: Freq., lowPitch, hiPitch, gain, lowVel, hiVel, etc.
- Annotation
    - Comment string
    - Annotation string(s)
        - Transcription, name, author, (c), DSP script, etc.
- Inter-file links
    - Parent/hierarchy
    - Sample groups
- Virtual files
    - Marker or named cue link files
    - Composite (mix) files
    - Gaps or silence markers in files
- Sample formats
    - Fixed or variable format
    - Which formats: linear, floating-point, exponential, μ-law, etc.

Resolution: 8-128 bits seen in the literature

Separate compression/emphasis data

Table 13: Criteria for comparison of Sound File Formats

Sound File Conversion Software

The topic of software to convert between sound file formats warrants special consideration here. One can easily write a program to do simplistic sample rate conversion (using interpolation/decimation), or sample format conversion (using simple numerical processing), or data compression (using standard compression algorithms), but to do this well, and support many file formats is quite a task. Several of the systems described above provide some sort of conversion utility, e.g., the NeXT’s `sndconvert` or the CARL system’s `conv` program. The two best-known public domain programs for this are SOX and SoundHack.

SOX

The most versatile “free-ware” tool for converting between various audio formats is SOX (“Sound Exchange”), written by Lance Norskog (thinman@netcom.com). It can read and write various types of audio files, and optionally applies some special effects. It is known to run on most UNIX workstations, the Macintosh, IBM PC-compatible PCs, and the Amiga.

SOX is a “batch” program that generally reads one sound file and writes another—possibly in a different format, and possibly with some effect processing applied, depending on the settings of its command-line options. It can read all of the sound file formats described here (whereby BICSF is still referred to as the older “IRCAM” format), and it recognizes all filename extensions listed in Table 3 above except “.snd,” which is quite ambiguous, and “.wav.” SOX can do a number of types of sound processing, such as sample rate and format conversion, gain modification, echo processing, stereo-to-mono mix-down, vibrato, and various kinds of filters.

The C-language source code for SOX, and its API library SoundTools, has been posted to the USENET news groups alt.sources, and should be widely archived on various ftp sites (just ask archie). A compressed tar file containing the version 10 of SOX is available by anonymous ftp from ftp.cwi.nl:/pub/audio/sox10.tar.Z.

SoundHack

SoundHack by Tom Erbe (tom@mills.edu) is a public domain Macintosh program that can read/write several file formats using 8-bit µ-law, 8-bit linear, 32-bit floating-point and 16-bit linear data encoding. It supports NeXT/Sun, AIFF/C, SoundDesigner, BICSF, and other formats, and can also read (but not write) raw data files. It also implements such useful processing operations as sound file convolution, a phase vocoder, a binaural filter, and an amplitude analysis and gain change module. SoundHack is also widely archived on the Internet, and can be found in many Macintosh software archives.

Internet Sound Archives

An electronic publication with information about digitized sound and sound formats—albeit mostly on IBM-compatible PCs—is the *Sound Site Newsletter*, maintained by David Komatsu (davek@uhunix.uhcc.hawaii.edu). It has recently expanded its charter to include commercial products and appears about once a month. There is also a sound site network of ftp servers, bulletin boards and authors. The Sound Site Newsletter has its own ftp site—sound.usach.cl.

The Sound Newsletter is posted to: comp.sys.ibm.pc.soundcard, comp.sys.ibm.pc.misc, rec.games.misc and is archived on the ftp servers sound.usach.cl:/pub/sound/Newsletters, oak.oakland.edu:/misc/sound, and garbo.uwasa.fi:/pc/sound.

Conclusions

In this article, we introduced the topic of sound file systems and discussed some of the issues involved in their design. We also presented a rough framework for comparison of sound file systems, and a detailed discussion of a sampling of current systems. The references below point to numerous paper and electronic resources for interested readers, who are also referred to the FAQ document mentioned in the introduction.

Acknowledgments

Many too many people have contributed information to this document to list them here individually. The FAQ document on which this text is based contains many references to the providers of the data collected above. The authors are grateful to D. Gareth Loy, F. R. Moore, Curtis Roads, Lance Norskog (etc.—more to come) for their comments on this text.

References

Apple Computer, Inc. 1986. *Inside Macintosh, Volume II*. Menlo Park, California: Addison-Wesley.

Apple Computer, Inc. 1989. *Audio Interchange File Format “AIFF”—A Standard for Sampled Sound Files, Version 1.3*. Electronic document available from ftp.apple.com, ftp.sgi.com, or mitpress.mit.edu:/pub/Computer-Music-Journal/Documents/SoundFiles/AIFF-C.ps.Z.

Audio Engineering Society (AES). 1990. *Recommended Practice for Digital Audio Engineering: Serial Transmission Format for Linearly Represented Digital Audio Data*. New York: AES.

Campbell, J. P., T. E. Tremain, and V. C. Welch. 1990. “The Proposed Federal Standard 1016 4800 bps Voice Coder.” *Speech Technology Magazine*, April/May, 1990: 40-49.

Dannenberg, R. B. 1993. “Music Representation Issues, Techniques, and Systems.”

Computer Music Journal 17(3): 20-30.

Electronic Arts, Inc. 1985. *EA IFF 85 Standard for Interchange Format Files* and *8SVX IFF 8-bit Sampled Voice Format*. Electronic Arts, Inc.

Fichera, S, 1991. "Machine Tongues XIII: Real-time Audio Conversion under a Time-sharing Operating System." *Computer Music Journal* 15(3): 27-40.

Gross, R. 1982. *The CCSS Cylinder-Contiguous Sound File System*. Rochester, New York: Eastman School of Music.

International MIDI Association. *MIDI 1.0 Specification*. Los Angeles: International MIDI Association.

Jaffe, D. A., and L. Boynton. 1989. "An Overview of The Sound and Music Kits for the NeXT Computer." *Computer Music Journal* 13(2): 48-55. reprinted in S. T. Pope, ed. 1990. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, Massachusetts: MIT Press, pp. 107-115.

Loy, D. G. 1982. "A Sound File System for UNIX." *Proceedings of the 1982 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 162-171.

Loy, D. G., et al. 1988. The Berkeley/IRCAM/CARL Sound File System. Internet Document mitpress.mit.edu:/pub/Computer-Music-Journal/Documents/SoundFiles/BICSF.t.

Loy, D. G. 1993. "Tools for Music Processing: The CARL System at Ten Years." in G. Haus, ed. 1993. *Music Processing*. Madison, Wisconsin: A-R Editions, pp. 267-300.

Mathews, M. V. 1969. *The Technology of Computer Music.* Cambridge, Massachusetts: MIT Press.

Microsoft Corp. 1992. *Multimedia Programming Interface and Data Specification v1.0.* Electronic document RIFFMCI.RTF available from ftp.microsoft.com.

Milne, S., and M. Deatherage. 1991. "AIFF File Specification." in *Commodore-Amiga, Inc. AMIGA ROM Kernel Reference Manual: Devices (3rd Edition).* Menlo Park, California: Addison-Wesley, pp. 435 ff.

Moore, F. R. 1982. "The Computer Audio Research Laboratory at UCSD." *Computer Music Journal* 16(1): 18-29.

Moore, F. R. 1990. *Elements of Computer Music.* Englewood Cliffs, New Jersey: Prentice-Hall.

Pohlmann, K. C. 1992. *Advanced Digital Audio*

Pope, S. T. 1992. "The Interim DynaPiano: A Computer Tool and Instrument for Composers." *Computer Music Journal* 16(3): 73-91.

Pope, S. T. 1993. "Three Systems for Software Sound Synthesis." *Computer Music Journal* 17(3): 32-55.

Roth, J. M., G. S. Kendall, and S. L. Decker. 1985. "A Network Sound System for UNIX." *Proceedings of the 1985 International Computer Music Conference*. San

Francisco: International Computer Music Association.
Sun Microsystems, Inc. 1994b. *The XAL Audio Foundation Library*. Document available via the World-Wide Web from http://www.sun.com.
Thieberger, E. M. 1995. “An Interview with Charles Dodge.” *Computer Music Journal* 19(1): 00-00.
Wiggins, G. et al. 1993. “A Framework for the Evaluation of Music Representation Systems.” *Computer Music Journal* 17(3): 31-42.

Appendix: Support for Sound I/O on Current Computers

Many current personal computers and workstations include (or can be augmented with) hardware to play back and (sometimes) record audio data in several formats and sample rates. We list a few of those in common 1994 usage below along with their characteristics. Table 14 lists the built-in capabilities of many systems; Table 15 shows some of the most widely used add-on equipment in use. For many systems you can also buy third-party products that provide “professional-quality” audio interfaces (≥ 44.1 k 16-bit stereo with high-quality convertors and analog subsystems). The characteristics listed here are a rough estimate of the capabilities of the “standard” hardware. Note that in Table 14 the bit resolution field includes logarithmic encoding; most systems that support 8-bit samples allow μ-law and/or A-law encoding. “4(st)” means “four channels from disk, stereo output.” For systems with different recording and playback rates, they are shown as “xx/yy.”

Machine	Bits	Max Rate	#Channels
Amiga	8	~29k	4(st)
Apple Macintosh (all)	8	22k	1
Apple Macintosh (recent)	16	64k	4(st)
Atari ST	8	22k	1
Atari STE,TT	8	50k	2
Atari Falcon 030	16	50k	8(st)
Sun SPARCstation (all)	8	8k	1
Sun SPARCstation 10/20	8, 16	48k	1(st)
NeXT	8, 16	44.1k	1(st)
SGI Indigo	8, 16	48k	4(st)
SGI Indigo2, Indy	8, 16	48k	16(st, 4-channel)
DEC 3000/300-500	8	8k	1
DEC 5000/20-25	8	8k	1
HP9000/705,710,425e	8, 16	8k	1
HP9000/715,725,735	8, 16	48k	1(st)

Table 14: Sound I/O Support on Popular Computers

Machine	Bits	Max Rate	#Channels

PC/Soundblaster pro	8	22k/2 44.1k/1	1(st)
PC/Soundblaster	16	44.1k	1(st)
PC/Pas	8	44.1k/2, 88.2k/1	1(st)
PC/Pas-16	16	44.1k/2, 88.2k /1	1(st)
PC/Turtle Beach Multisound	16	44.1k	1(st)
PC/Cards with aria chipset	16	44.1k	1(st)
PC/Roland rap-10	16	44.1k	1(st)
PC/Gravis ultrasound	8, 16	44.1k	14-32(st)
Macintosh/Digidesign ProTools	16	48k	1-16(st, 4-channel)
Macintosh/Studer Dyaxis	16	48k	1-16(st, 4-channel)
Sun/NeXT/Mac/PC/or other with Ariel ProPort	16	96k	1-8(st, 4-channel)
NeXT/MetaSound Digital Ears	16	48k	1(st,4)

Table 15: Popular Add-ons for Sound I/O

All these machines (except for “off-the-shelf” IBM PC-compatible machines) can play back sound files of 8 kHz 8-bit µ-law quality or better without additional hardware, although the needed software is not always provided with the standard operating system release. Some machines need external hardware to record sound (or to record at high quality—like the NeXT machine, which had a 16-bit 44.1 kHz DAC but whose built-in ADC only supported 8 kHz 8-bit µ-law format). A few of the newer systems (e.g., the SGI Indigo, a Sun, NeXT, or Macintosh with Ariel’s DATPort, or the Digidesign ProTools system for the Macintosh) provide digital audio I/O in the form of S/P-DIF or AES/EBU ports. With these, one can use third-party external ADC/DAC HW (such as from Ariel Corp. or Apogee, Inc.), or record from disk to DAT (or reverse) in the digital domain. Several CD-ROM interfaces also allow one to record CD audio data direct-to-disk over the SCSI bus.

Data such as that given in these tables is bound to go out-of-date rather quickly—interested readers should consult an up-to-date version of the FAQ document on which this article is based (see above). In particular, there is a separate USENET news group—called comp.sys.ibm.pc.soundcard—devoted to IBM PC-compatible sound cards; it includes an FAQ document of its own (also posted to the group comp.answers).