Behavior Research Methods, Instruments, & Computers
1989, 21 (2), 126-129

Standards for PCM files

JOHN MERTUS
Brown University, Providence, Rhode Island

One problem facing the speech research community is how to share complex programs and A/D sampled data. This paper is a general discussion of different approaches to sharing and the advantages, disadvantages, and implications of each method. The focus is on setting a standard for PCM (Pulse Code Modulated) data files and how this could be accomplished.

At the Department of Linguistic and Cognitive Sciences at Brown University, we have five different types of computers on our local ethernet: VAXes, PC/AT equivalents, Macintoshes, a Celerity, and a TRACE Multiflow minisuper. Our feeling is that no single system or even a single vendor can provide all we need. Rather, the network is our computer system. The VAXes are used for speech analysis and waveform editing or program development, the ATs are used as subject testing stations, the Celerity is used for graduate student program development, the TRACE is used for research number-crunching, and the Macintoshes are used for word processing.

For this reason, we try to write analysis programs so that data files, most notably the files that contain sampled audio (PCM, or Pulse Coded Modulated) data, can be transferred directly to each machine and used without conversion. Thus, most users ship files back and forth without thought of compatibility. But our laboratory is isolated, and the types of files supported are limited. Programs that cannot be converted to read our files are doomed to a short life. Likewise, our programs are not used in many places other than Brown. Furthermore, I find myself unable to fulfill users' needs for more modern programs and different types of analyses.

For example, recently I received a program that runs digital spectrograms on an IBM PC/AT. It was a nice program, with good displays and menu commands. It also ended up in my trash can. Why? Because the cost of supporting it was too high for the benefit. I call these programs "black boxes" because neither the source code nor a detailed explanation of how they work is available.

We at Brown are not at all unusual; most researchers find themselves in the same position. If the speech-research community defines a set of standards that are carefully followed, it would allow all of us to share programs, even black-box software, and use them productively. In order to make that spectrogram program useful to Brown, it would have to read a standard file format and be able to play out what is currently being displayed.

Below is a discussion of how I see this could be done. It is in no way intended as a proposal, but rather as something to open up further discussions. Furthermore, I am an AT and VAX programmer; I do not understand Macintoshes and I warn that this discussion will be slanted toward the AT and VAX machines and their operating systems. These ideas come from discussions about such issues with various people, most significantly Terry Neary of the University of Alberta, and Philip Rubin and Lance Maverick of Haskins Laboratories.

Correspondence may be addressed to John Mertus, Department of Cognitive and Linguistic Sciences, Brown University, Box 1978, Providence, RI 02912.

For various reasons, we cannot look to large software companies, for example ILS, to lead the way with standards. First, as a group we do not spend enough money to affect their sales. Second, since these companies have committed vast resources to their proprietary formats, changing them would cost too much, both in money and in compatibility. But most of all, the more they keep their software different, the more they force the user to continue buying their systems.

It has been shown time and again (UNIX as the prime example, KERMIT as another) that a public standard, even a mediocre one, is better than no standard at all. Furthermore, a standard may become de facto if it is fairly and reasonably done and fulfills a need.

FILE PORTABILITY

We would like a standard sampled format for audio data. However, this begs the more fundamental problem of the desired approach to file portability. For now, I will mention the three approaches that I use. I am sure there are other approaches.

First, the software should employ a binary standard. This means that files can be shipped from one machine to another and read directly by the programs on the machines. This standard is very difficult to define and requires good, careful programming.

Second, there should be a conversion standard. This means that files generated on one machine can be converted into the host machine format semiautomatically. This is more difficult than it sounds because one has to identify the type of file, the format of the data, and so forth.

Third, software should conform to ASCII standards. Here, the header and the data are converted into ASCII, which can be read by all machines (even EBDIC systems).

Of course, there are no sharp lines dividing these standards, and no standard is best in all situations.

I use the binary standard for all sampled audio data, the conversion standard when files are created in floating point on the Multiflow and shipped to be displayed on the VAX, and the ASCII standard for the files that I might want to read and modify.

The Binary Standard

This may be the most difficult standard, but it offers the best rewards. It is impossible to deal with all computer architecture, but most of the traditional machines such as VAXes, ATs, Macintoshes, and most RISC machines have a standard 8-bit, byte-oriented architecture.

Assuming only an 8-bit architecture, it seems possible to define a binary standard, that is, a standard that allows both for transparently shipping data files between machines and for running black-box programs on the same machine. The latter requirement means that we must agree on a PCM format that is general enough to support different hardware. The former requirement will necessitate some sort of internal translation. This should not be too hard because on these machines the subroutines for reading and writing the data can be made to behave in the same manner. Thus, as part of the standard, header definitions and basic I/O subroutines could be distributed, making the standard tighter and easier to use.

But even assuming an 8-bit architecture, there may be a problem of which byte has more significance. This can be dealt with in at least two ways.

First, a byte in the header defines which is the high byte or bytes. But whenever the data is read in, and is not in the correct intrinsic format for that machine, the bytes are swapped. The main disadvantage becomes the overhead of this type of conversion. If the data is read just to be analyzed, the overhead of a single subroutine that transparently gets data from the file can be tolerated. But in the real-time case, usually machine-dependent routines are necessary for fast reading, and byte swapping may not be possible due to time constraints.

Second, if the format does not agree with the intrinsic machine format, when the file is first referenced, the program converts the entire file in place and changes the format byte in the header. From then on, the file can be referenced without any conversion overhead. The major disadvantage of such an approach is that write access is necessary to a file that has not been converted. Thus, one user cannot simply look at the data of another. Strictly speaking, this is not a binary standard but, as mentioned above, the line demarcating the two is fuzzy. Because the data size and position in the files does not change, I choose to define this type of internal conversion as belonging to the binary and not conversion standard. Sometimes I call this a translation standard.

A combination of both methods could also be used, especially if general purpose routines for that purpose are written and are well documented.

The same problem occurs, but to a much larger extent, for floating point. Here, although there are IEEE standards, nothing pleases everyone, so most systems use different floating-point representations. Again, assuming an 8-bit byte architecture, then floating-point words usually occupy 4 or 8 bytes. To ship floating-point files back and forth, it becomes essential to deal with this incompatibility. Both of the integer solutions can be used. This area needs much more exploration.

A major disadvantage of the binary standard is the tight computer architecture to which it is tied, and the fact that programs must be rewritten to deal with the different types of byte significance and floating points.

The Conversion Standard

At first glance, conversion seems to avoid reprogramming by making the data translation occur outside the program rather than inside it. However, this has major implications for transparent use.

It is not hard to write a program that converts data from one machine to another or translates files for one program type into files for another. However, bookkeeping becomes more difficult. For instance, files from two or more systems may result. A conversion program should, at minimum, (1) support wildcard file names, (2) not convert files that have already been converted, and (3) know from the file how to convert it. The latter two requirements imply that files from different machines or different programs need to be marked differently (i.e., global information about where the data came from and what programs use them). From past experience, the amount of global information should be kept to a minimum. Thus, careful definitions for the header are required; in fact, I would claim that these definitions should be as careful as the binary standard.

Besides reprogramming, an obvious advantage of the conversion standard is that it supports different computer architectures and can deal with wildly different file formats. But a binary standard for 8-bit machines can also be a conversion standard for all machines.

Still, for transparent use, having different file formats is bad. For example, one could have several black-box programs for the same machine written by many people with many different file formats. Unless file extension naming conventions are strictly enforced, the user will quickly lose track of which program takes which type of file, and so forth.

One of our IBM ATs is DECNetted to our VAXes; that is, a program on the AT can open a file on the VAX and read it. With a binary standard, it is possible to open that file transparently. With a conversion standard, one must copy the file, convert it, and then work with it. Rather than considering this a special case, such sharing is becoming more and more commonplace.

I think the conversion standard is fine for some data, but I would argue that it is not acceptable for PCM files. If the conversion route is followed, there should be one PCM format for each operating system/machine. Thus, reprogramming is still necessary. It would be just as simple to build a binary standard. Also, a good binary definition will allow conversion programs to be written

for machines, such as supercomputers, that do not use 8-bit architecture.

The ASCII Standard

Certainly the ANSI ASCII standard is one of the most universal in the entire computer industry—only IBM remains a holdout on its high-end systems. It is also very easy to dump data in ASCII format and read it back. But it has major problems.

There are at least two ways to represent different types of data within an ASCII file. First, by position; that is, the first word is the sampling rate, the second is the size of the file, and so forth. The second method is by keyword information; that is, the file contains lines such as

Sampling Rate=20000
File Size=12448.

However, for large files, such a representation consumes a great deal of disk space and the cost of reading in the data is very high. Even in this day of gigabyte disks and 5-mips machines, disk space and central-processing-unit power are and will remain at a premium. I like ASCII for files that may be read or edited by people, but it is wholly unsuitable for the real-time PCM files.

PCM FILES

Now we come to the crux of the matter—how to represent PCM files. I am not married to the ideas represented here and am not proposing them as the standard. They are just the first steps in defining a header.

I really think we should go for a binary standard. I believe it is workable. The first question is where the header should reside—at the beginning of the data file or in a completely separate file. I strongly support placing it at the beginning of a file. Joanne Miller of Northeastern University encountered problems when her lab separated the header from the file. Having two files around caused all sorts of problems.

Next, how should the PCM data be represented? There can be bytes in the header telling the IEEE type of the floating point and the integer type. Furthermore, another byte can tell the format of the PCM data (i.e., if it is floating point or integer, and how large). Conceivably, one could represent the PCM data in a variety of ways. But is there any reason for not using only integers? Or for using only 2-byte integers? This is another area to study.

Should the data be only PCM data or should it perhaps allow other sampled types, such as differential or μ-companded? Let me digress a moment before addressing this issue.

A Universal Audio Command

One of the most useful features of a speech-analysis program is the ability to hear a section of the speech waveform while within that program. At first glance, a standard that supports the hundreds of A/D and D/A boards of the real world seems impossible to define. However, one solution is to require all programs to have the ability to run another program. Thus, a command line can be passed to the operating system for playing out a section of the file. The individual system can have such a program tailored to the specific hardware.

For MS-DOS, UNIX, or VMS, the command could look like:

```
PLAYOUT  file=junk.pcm  repts=4  start=1200  stop=4800
```

This command is driven by keywords. If a keyword is not specified, a standard default would be assumed. For example, the default for "start" could be 0 and for "stop" the end of file. Another similar idea can be used on the Macintoshes, which do not easily support command lines.

Of course, this does not prohibit specialized programs such as waveform editors, where—in order to have a reasonable response—the audio is directly tied to the program. This standard just requires that the program be able to pass the PLAYOUT command line to the operating system, and in most operating systems that is trivial.

PCM Files Continued

I discuss a "universal" audio command at this point because it greatly affects the data format of a PCM file. The current class of machines is hard-pressed to output dichotic files at 20K per channel from disk. Thus, one cannot afford a large overhead in reading in the file. Even if we are to agree that all data is in 16-bit PCM format, this will not be enough. The reason is hardware. Do we try to dictate 12-bit bipolar? I think at minimum we must support different bit resolutions and the different formats of unipolar and bipolar.

The question becomes, How can this be done? I do not have a totally satisfactory solution, but below are three alternatives.

First, the PLAYOUT program could create a new file in the proper format and play that out. This could be very time consuming.

Second, the program calling the PLAYOUT program could convert the file before requesting a playout. Because the calling program may have been written for different hardware, the formats acceptable to the PLAYOUT program must somehow be coded on the current machine and not hardwired into the calling program.

Third, the PLAYOUT program could convert the file once and only once. Then the calling program must reread the file header so that proper data translation will take place.

In each of these cases, the calling program must be able to support multiple file formats. As mentioned above, that is not hard at all. For example, a routine called "get_FP_buffer" in C and Pascal, or "GetFPB" in FORTRAN, could be written that has the file header as one of its arguments and returns the PCM data as a floating-point buffer with zero voltage corresponding to 0. Distributing such subroutines now makes it easier for programmers to follow these standards. In fact, it makes writing programs easier because the I/O has already been done.

The PCM Header

Another area that needs careful examination is exactly what is in the header. That, of course, depends upon the user community. Striking a balance between a large and unwieldy universal header and one that is too simple is essential and is more of an art than a science.

My preference is for all data files, PCM or otherwise, to have a few bytes in the beginning that mean exactly the same thing. For example,

1. The first 4 bytes are an identifier usually coded in ASCII, but not necessarily so. This identifier could be something like "PCM1," standing for a PCM file, version 1.
2. A word giving the offset into the file where the data starts.
3. A word or byte giving the size of the header in bytes.
4. Some bytes giving the representation of the floating point and integer within the file (e.g., the IEEE floating-point standard assumed, order of significance, two's or one's complement, etc.).
5. A byte giving the format of the data in the file (e.g., floating point, double precision, byte, short integer, etc.).
6. A link word to another header in the file. This is zero if there is not another header.

The rest of the header, such as the size, sampling rate, number of channels, and cursor marks, is something the user community needs to hash out. Furthermore, we could distribute simple subroutines for manipulating headers. For example, these could be called "Make_PCM_Header," "Write_PCM_Header," and "Read_PCM_Header." This makes the programmer's job easy.

The idea of the link word is very important. The word is zero, which means that nothing follows the data or the offset into a file where another file header starts. This is useful for taking a PCM file and attaching to it information such as a mark file.

Some people would like a section of the header they could define. However, using a link word avoids the problem by reserving a section of the header for this purpose. The link words go to another header, and this header might link to another. Each header has an identification. Thus, a program can search out the specific header, and if one does not exist it can be added at the end of the file.

My waveform editor does this; when it reads in a large file that takes a lot of processing in order to create the display of the entire waveform, it first checks if such a display is stored. If so, it reads it in; if not, it creates it and stores it so loading the next time is quite fast. Another person who reads my audio files would just skip this data altogether.

REPRESENTING STANDARDS

How can we go about achieving such goals? That is a tough question, but I have a few ideas.

1. There should be a good written standard of the header contents.
2. There should be header files that define the PCM header in the standard languages of FORTRAN, C, and Pascal.
3. A few general-purpose subroutines for data manipulations written in FORTRAN, Pascal, and C should be distributed. We may or may not want this code to meet the ANSI standard definitions for those languages.
4. A template for the PLAYOUT program should be distributed, which would make writing this machine-dependent program easier.
5. Most of all, there should be an agreement by as many labs as possible that they will support the common standard.

The final question remains, how much support are people willing to give? Should we try to get funding to develop a standard? In the long run, that money would be returned quickly. On the other hand, should we rely on established laboratories to talk and try to reach an accord?

CONCLUSION

Everyone agrees that a PCM standard would be very useful and desirable. Initially, the cost will be very high, and converting to a standard will cause much pain and local incompatibility. However, in the long run the return will be enormous. Imagine reading a paper about some result using a new type of measurement, realizing that it might shed some light on one of your own problems, calling that person, finding out that he or she has an AT, and receiving an executable image over arpanet that you can run right away on your data.

I think that goal is attainable, but to reach it each of us will have to give a little. If we can come up with a good standard, I pledge that the Linguistics Laboratory at Brown will spend the necessary time supporting and using it.