The Architecture and Musical Logic of Cmix

Paul Lansky
Music Department
Princeton University
Princeton New Jersey
paul@princeton.edu

Abstract: **Cmix** is a toolkit for synthesis and analysis. It differs substantially from most synthesis packages in that it has no scheduler and accumulates mainly by mixing to disk. It has a scorefile language written in C-like syntax which dispatches commands to the Cmix kernel and user functions written in C, and receives values in return. The basic argument of the system is that it provides a relatively low-level, musically neutral interface.

Cmix was written mainly to satisfy my needs and not as a general purpose system which would have universal appeal. I had worked with many music programs, beginning with **Music4B,** written in **BEFAP,** back in the mid '60s, and gradually evolved a working method which was musically satisfying, given my hardware resources, my compositional interests, and my programming abilities. **Cmix** is a distillation of these methods, and though I didn't write it with any kind of distribution in mind, it has been adopted by others, and there seems to be general confusion about what it is and what it does. Therefore, I'd like to take this opportunity to explain what it is, and the musical logic behind its architecture.

Cmix is essentially a toolkit. It is closer in spirit and architecture to the **NeXT MusicKit, Fugue** and the **Carnegie Mellon Midi Toolkit,** than it is to **Csound, Cmusic, Music4,5,** etc[1]. There are several design principles behind **Cmix** which distinguish it from other systems. The first is that its architecture encourages the working methods of the user to emulate the protocols of a rehearsal rather than of a performance. The guiding principle of a good rehearsal is not to waste valuable time rehearsing things which don't need work, and to spend time on individual parts, finally combining the whole ensemble only when all the component parts are in shape, i.e. mixing. To this end **Cmix** merely executes each score command as it is occurs (there is no sorting), and it normally mixes its output to disk, although an option for destructive writes is provided. Any part of a soundfile may be addressed at any time. A consequence of the adoption of the mix/rehearse model is that a scheduler is not useful, and instead random-access to a soundfile is provided, analogous to a rehearsal, in which one wants to be able to jump to arbitrary spots. **Cmix** is therefore ill-suited to any kind of real time synthesis system. Instead, its intended domain is the non-realtime creation and processing of soundfiles. Rehearse intelligently until you get it right. Also implicit in these assumptions is the view that no matter how fast computers get, our musical demands will always exceed their normal capacities and therefore CPU time will always be relatively expensive. (This assumption is undoubtedly false but it has kept me busy for ten years.)

As an extension of the rehearsal model, **Cmix** can also be thought to adopt a multi-track recording studio model. Any number of tracks can be created, although these may not be channels in the

audio sense, but rather individual parts, giving a soundfile something of the attributes of a data-structure, or even a musical score. (I have worked with as many as 20 channels.) One of the main virtues of this architecture is that it is easy to start, stop and undo without losing data. At a recording session it is customary to avoid re-recording an entire track, piece, or movement just because a few errors occurred. In **Cmix** it is easy to accumulate results, selectively redo, and even emulate the format of a multi-track recording session in which only at the final mixdown is everything heard at once.

The second design principle is to minimize the use of special purpose syntax, maximize the availability of **Unix** tools. There is no synthesis or patching language as such, only a library of C routines, unit-generators and utilities to make it easy to get samples from soundfiles or invent them, do something with them, and mix them out to the same or different soundfiles. All that is necessary to write a **Cmix** application is a basic knowledge of C. All signal processing applications essentially just need an initial call to position pointers on input and output files, a loop through the samples with some form of i/o, and a cleanup call to update the soundfile headers. A disadvantage to this is that the amount of time between making changes in instrument design and hearing results is greater than in the case of a language such as **Csound**, for example, which makes the test cycle much shorter. (In fact, when I want to do this sort of thing, I use **Csound**.) The advantage of this is that the specificity with which **Cmix** arbitrates musical decisions is much lower than in most music synthesis packages, and comes closer to the sense of a language compiler such as C, which itself is basically 90% of the architecture of **Cmix**. A template for a simple Cmix instrument is:

```
double dosomething(p,n_args)
float *p;  /* array of values passed from score */
{
        samplecount=setnote(start,dur,filenum): /* position pointers */
        for(i=0; i<samplecount; i++) {
                do something....
                ADDOUT(something,filenum):  /* write to soundfile */
        }
        endnote(filenum);  /* update headers */
        return(whatever); /* return a value to score */
}
```

The scorefile will simply address this routine directly by name, cause it to be invoked, pass it arguments, and receive a possible return value. **Cmix** thus tends to encourage the development of applications rather than continuously evolving instruments. I have written a powerful mixing and editing program and a large number of special-purpose filter and sound-processing routines.

The third important design aspect is that the data specification language, the score in other words, also uses C-like syntax and can be thought of as a kind of interpreted C. **Cmix** functions are called by by name from a scorefile and return values to the scorefile for subsequent interpretation if necessary. As a result, it is possible to embed logic normally contained in instruments directly in the scorefile: trivially, time and pitch conversions, and less trivially, the consequences of operations. The language is a subset of C and is called **Minc** (for **Minc Is Not C**). It was written

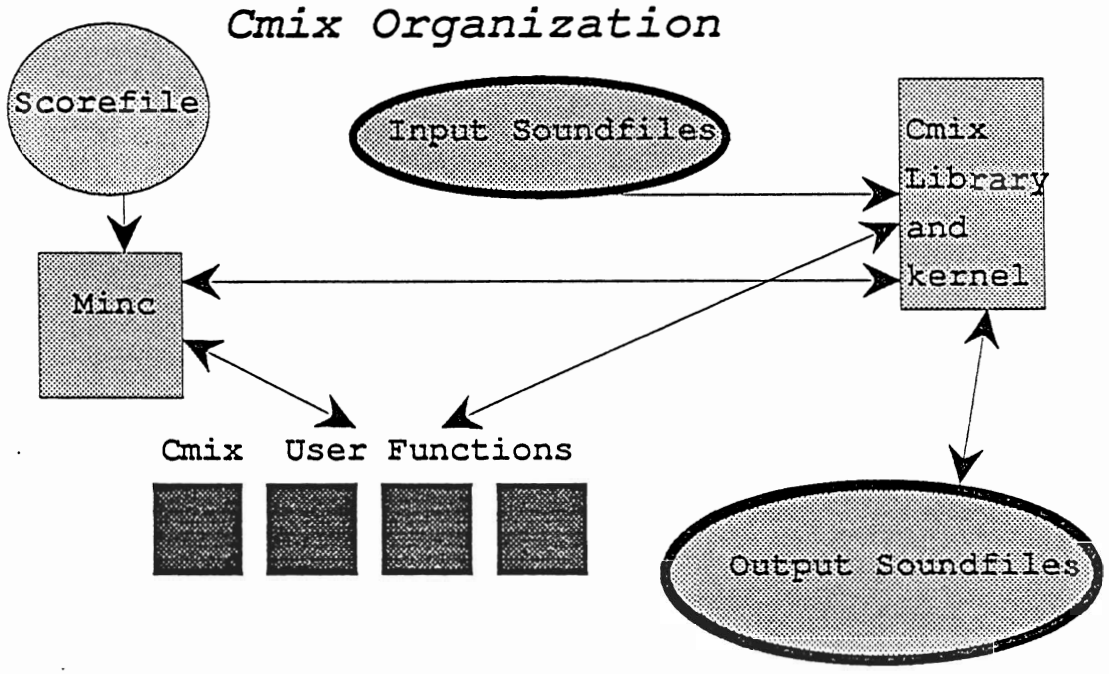
by Lars Graf. Here is an example scorefile from a hypothetical instrument which will process a soundfile with some sort of filter and return the resultant peak amplitude for subsequent processing by another routine.

```
input("yawn")                          /* open input soundfile as file 0 */
output("cough")                        /* open output soundfile as file 1 */
float start,peak,loop,limit,freq       /* define variables */
freq=100
limit = 16000                          /* test and loop */
if((peak = grunt(start=0,dur=3)) > limit)   {
   for(start=3; start < 21; start=start+1)     {
        peak = beep(start,dur(0),peak/2,pchcps(freq))
         freq = freq + 100
    }
}
```

In this case *grunt*() and *beep*() are user-written C functions, which will do some operation on the soundfile called "yawn" and write the output, probably by mixing, to the soundfile "cough". *Input*(), *output*(), *dur*() and *pchcps*() are built-in Cmix functions. *Input*() opens an input file with internal file id 0, *output*() opens an output file with file id 1, *dur*() returns the length of the input file, and *pchcps*() converts hz to 8ve.pc form. There are currently 36 functions built into the Cmix kernel which are recognized by Minc, and it is simple to add more. Since there is no distinction, as far as Minc is concerned, between one C function and another, the user may write functions whose sole purpose is to process signals, to return values to Minc, or even just tell the time of day. The close relation between score and instrument reduces the independence of these two components. A score itself may take on the characteristics of an instrument, and an instrument may include score-like features such as the generation of notelists.

The real heart of the Cmix architecture thus lies in the collaboration between Minc and user-written C functions. There is, moreover, no particular reason why the Cmix kernel and library have to be called by user-written functions. In fact I have written some applications in which i/o is done using the NeXT Soundkit. Furthermore, since user functions have no requisite kernel support it is easy to adapt the program in a number of ways. At Columbia, Mara Helmuth has written a graphical instrument design program under X and I have also started writing a library of Objective C functions. What I have found most useful about Cmix is the power gained by the Minc interface, the generality of its low-level approach, and the fact that it is such a preposterously simple system.. There are a number of improvements which could be made, such as the use of shared libraries, an optional scheduler, a graphical interface, etc I also suspect that realtime hardware coming in the next year or two is going to argue for new approaches, but my inclination is to let someone else fuss with these, I'd rather write music at this point.

Cmix is available by anonymous ftp from princeton.edu. Versions for Sun, Vax and NeXT are in pub/music.



Notes

[1]Dannenberg and Fraley, "Fugue: A Signal Manipulation System with Lazy Evaluation and Behavioral Abstraction", *Proceedings of the 1989 ICMC*, pp. 76-79

Dannenberg, Roger B. (1986) **The CMU MIDI Toolkit**, Carnegie Mellon University.

Jaffe, D. and Boynton, L, 1989. " An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal* 13(2):48-55

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

Vercoe, B. 1986. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. MIT Media Lab.