8. Introduction to the Eastman Csound Library

(This section last updated August 2004)

The *Eastman Csound Library*, the topic of this section of the *Users' Guide*, is a collection of front-end programs, "instrument" algorithms, score file preparation programs and assorted utilities designed to simplify ECMC users' introduction to, and use of, the *Csound* music compiler. *Csound* is a widely used software audio synthesis and signal processing package, available at no cost under the LGPL (Lesser General Public License) from the Csound Home Page web site (*http://csounds.com*) in versions Linux (and other flavors of Unix), Macintosh and Windows.

Csound initially was developed on Unix systems during the 1980s by a team led by Barry Vercoe at the Massachusetts Institute of Technology and employed a commands issued from shell windows. During the 1990s the Csound programs were ported to Macintosh, Windows and other platforms. Today, cross-platform development of the "canonical" (official) Csound source code continues by a world-wide group of Csound users, coordinated by John ffitch, who update "canonical" (officially supported) versions of the language.

Like other software music compilers (such as *Cmix* and *cmusic*) that have been written at academic computer music centers, *Csound* has its roots in the very first music compilers, called *Music I* through *Music V*, written by Max Mathews at Bell Labs beginning around 1957. In most of these compilers, the fundamental means of creating or modifying sounds is a user-defined synthesis or signal processing algorithm, called an *instrument* in *Csound*. The algorithm consists of a step-by-step sequence of mathematical and logical operations, coded in the *Csound* compiler language, and then executed by the CPU. Often, these operations are very similar to the types of operations that take place within the electrical circuits of hardware synthesizers, amplifiers, mixing consoles and other types of audio gear. A particular *instrument algorithm*, which generates a particular type of audio signal or timbre, might be likened to a particular "patch" available on a hardware synthesizer or sampler. Alternatively, an algorithm might add reverberation, or echos, to a previously created soundfile, or to the audio signals being generated simultaneously by some other *instrument* within the same compile job.

Orchestra and score files

Csound is completely modular, providing the user with several hundred basic building blocks (called *unit generators*) that perform specific types of operations. The inputs to, and outputs from, these unit generators are combined (or "patched together") to form the complete algorithm. In turn, several of these *instrument algorithms* can be "playing" simultaneously within a Csound *orchestra*.

This "orchestra" requires performance input — something to play. Traditionally, academically-devloped software music compilers, which pre-date the introduction of MIDI by about 25 years, have employed a *score file* to provide this performance data. When viewed, the data within a *score* file is similar in some respects to the data within a spreadsheet, or within a *MIDI file* that one creates with a software sequencer or with a program such as *MAX*. Historically, however, this note and event list has been created not by playing MIDI keyboard controllers, but by the more tedious method of typing all of this data into a *score file*.[1] To make sound with *Csound*, then, at least two files are needed: an *orchestra* file to provide one or more "instruments" (sound generating or processing algorithms), and a *score* file to "play" these instruments. To identify the functions of Csound files, a *.orc* extension is generally appended to the names of orchestra files, while a *.sco* extension is appended to the names of score files.

Real time input and output

During the past fifteen years *Csound* has been expanded by additional modules and coding that provide real-time playback and, in addition to the traditional score file, MIDI file and/or real-time MIDI controller input. [2] Even on powerful computer systems, however, MIDI input capabilities have not rendered traditional file-based score preparation procedures obsolete. In fact, most ECMC users who have worked both

[1] By contrast, almost all recently developed software synthesizers, such as *Super Collider* and *Reaktor*, employ real-time MIDI input.

[2] Actually, even with MIDI input, a score file is still required, but typically will be reduced to a skeletal shell, containing only some necessary input data that cannot be provided by MIDI controllers.

with MIDI and score file input, and with GUI as well as text-based interfaces to Csound, prefer to use score files created with text-based programs. Certain types of musical material, such as melodic lines, often are more easily or successfully realized by means of MIDI input; but other types of musical gestures and concepts, such as algorithmic compositional procedures and certain types of rhythmic and textural procedures, frequently are achieved more easily or successfully using the traditional score file input method. Most importantly, in music with *lots* of note-to-note variables and nuances, one soon runs out of MIDI controllers, or of hands to work these controllers, whereas score files can provide a virtually unlimited number of parameters to control these variables.

Similarly, real-time playback, while offering the abundant advantages of immediate feedback, can introduce constraints as well, and sometimes is not desirable. System throughput (getting those samples out in time) becomes a paramount concern, often limiting the number and complexity of signal processing operations possible, and/or the available "polyphony" (number of simultaneous sounds), or requiring lower sampling rates. Often there is no easy method to audition *Csound*'s output in real-time and, simultaneously, to write these samples into a soundfile.

When using the real-time playback and/or MIDI input capabilities of *Csound*, you frequently will be aware of such trade-offs and constraints. Hardware synthesizers, used in conjunction with sequencers, provide a guaranteed number of multi-timbral "voices," but often severely limit the programmability of these voices. With software-based synthesis, these advantages and disadvantages often are reversed.

Front-end programs and applications that run Csound

Although *Csound* provides powerful and highly extensible procedures for sound generation and modification, many new users initially find the syntax of the *Csound* language "complicated", "old-fashioned" or "unfriendly," its learning curve uncomfortably steep, and the initial results of their efforts musically disappointing. As with many comprehensive software packages, a certain critical mass of procedural and technical information is required before the user can accomplish *anything* worth listening to, and musically satisfying results often come only after some experimentation, head scratching, fiddling and, perhaps, cursing.

Many types of front-end programs and applcations, built on top of the canonical Csound source code, have been written and distributed in attempts to simplify the usage of Csound. Most of these front-ends employ graphical user interfaces in place of the "canonical" command line interface, and many also incorporate working procedures or features that are unique to a particular operating system platform.[3] Some notable examples of these front-end applications include *Cecilia* (Linux, Macintosh and Windows), *CsoundVST* and *CsoundAV* (Windows), *MacCsound* (can you guess which platform?), and *Blue* and *HPKComposerCsound* (java).

The *Eastman Csound Library*, too, attempts to address some of these initial hurdles, providing various plug-and-play modules that allow one to begin using *Csound*, and experimenting with some of its resources, more quickly, without immediately becoming immersed in details. This ECMC Library, which has been in use and under development for over twenty years, enables ECMC users new to Csound to begin working near the top, rather than bottom, of this modular sound synthesis process, selecting from among a collection of pre-defined *instruments* to be included in his/her *orchestra* file, and providing templates to produce score files for these isntruments by means of a program called *score11*. Additional utilities also are provided to simplify and speed up many common tasks associated with using and running Csound.

Around 1980 Aleck Brinkman wrote the initial version *score11*, a front-end *preprocessor* that enables users to create many types of *Csound* score files more quickly, easily and intuitively.[4] Today, several score file preprocessors for *Csound* are available, and many expert Csound users write their own pre-processors to generate score files (and sometimes orchestra files as well). However, in my experience, *score11* — whatever its faults — remains the most powerful score file generating program for Csound.

Currently *score11* is only available for Linux and certain other Unix-based latforms. A few years ago Mikel Kuehn, an ECMC alumnus now teaching at Bowling Green State University, wrote a similar score preprocessor called *ngen*, available for Windows and Macintosh and well as Linux systems, that also is

[3] *score11* originally was written for *Music11*, the predecessor of *Csound* which ran on Digital Equipment Corporation *PDP-11* computers.

available on *madking*. *score11* and *ngen* share similar capabilities as well as a similar syntax. If you know one of these programs you can learn the other fairly quickly. Most of us who know both programs, however, tend to use *score11*, which has a few powerful features (particuarly with regard to tempo warping) that are not currently available in *ngen*.

Most of the remaining pages in this section are devoted to basic procedures for using the "pre-set" Eastman instrument algorithms, to preparing *Csound* compilation jobs and to alternative ways of running these jobs. More complete documentation on particular instrument algorithms and other resources introduced or mentioned here is available in the document *The Eastman Csound Library*, available in rooms 52 and 53.

8.1. Score-based library instrument algorithms

Two basic types of orchestra file *instrument algorithms* are available within the Eastman Csound Library:

(1) those that require performance input from a score file, and

(2) algorithms that require performance input from MIDI controllers, or from a MIDI file

For a variety of reasons, these two types of algorithms tend to be mutually exclusive. However, a few instrument algorithms with similar names (such as *samp* and *midisamp*) exist in alternative versions, one requiring score parameter field input, the other MIDI input. This subsection covers the *score-based* algorithms.

To obtain a list of the *score-based* instruments and processing modules available on the machine on which you are working, type

lsins

The following resources are available for each of these instrument algorithms:

1) an online and hardcopy *manual page*

To display one of these documents in a shell window, type

man instrumentname

(where *instrumentname* is the name of the instrument algorithm, e.g. *marimba*).

Hardcopy of all of these man pages, and of available *score templates* and *example scores* (see below) is included in the studio document *The Eastman Csound Library*. A selected sampling of these *man* pages, score templates and example scores for a few of the most frequently used library algorithms are included later in this *User's Guide* section.

2) a *score11 template*

A *score11* template provides all of the necessary (and some optional) *functions* and commented *parameter field (p-field)* input arguments necessary to prepare a score file for a particular instrument.

These *p-field* arguments provide an instrument algorithm with values that vary from note to note, such as pitch. Score *p-fields* thus function somewhat like *note-on, key number, velocity sensitivity* and other MIDI controller signals. Note, however, that *p-field* values are note *initialization* arguments that remain fixed throughout the duration of a note (like MIDI *note number* signals, but unlike the *continuous controller* data from a MIDI modulation wheel or pitch wheel).

A *function* is a table of numbers. The numbers within a function table may define an audio waveshape (such as a sine wave), or a control shape (such as a linear or exponential amplitude envelope shape), or simply a series of values that are used or processed in some manner by the instrument algorithm. Function definitions included within *score11* files begin with an asterisk, and often look rather cryptic to new users, since they tell function generating subroutines within *Csound* how to create a desired function. One of the first things that *Csound* must do, before it can compute sound samples, is to create and load into RAM a table for each function specified in a score file. If you run the *csound* command with the (default) display option, the compiler also will display these functions graphically on your monitor.

The command **lstp sc**
("*l*i*s*t *t*em*p*lates for *sc*ore11") will print a list of available templates. There is one template for each instrument.

The command **sctp**
("*sc*ore11 *t*em*p*late") followed by the names of one or more instrument algorithms, will display in your

shell window a score template for the requested instruments. To use a template to create a score by "filling in the blanks," redirect the output into a file

sctp marimba > filename

Then edit this file with a text editor.

3) one or more *example scores*

These example scores illustrate typical (if often didactic and bland) usages of an instrument, or special features, as well as some of the resources of *score11*.

To see a list of available example scores, type

lsex

To display one or more of these examples scores on your monitor, type

getex file(s)

The examples scores can also be redirected into a file, for printing or more leisurely viewing :

getex marimba1 > filename

4) *compiled soundfiles of the example scores*

The example scores have been compiled into soundfiles that are located within the */sflib/x* soundfile directory. To list the current soundfiles in this example directory, type

lsfsflib x or **lsfl x**

To display the soundfile *header* information for one of these soundfiles, type:

sflibinfo x/filename or *sfli x/filename*

To play one or more of these soundfiles from a shell window, type:

playsflib filename(s) or **psfl filename(s)**

Finally, should you would wish to see a master list of these various types of *Csound Library (cslib)* files for score-based instrument algorithms, type :

lscslib or **lscsl**

To view this rather lengthy listing one screenful at a time, pipe this command into a window paging program such as *less* or *more*:

lscslib | less

8.2. Procedures for creating soundfiles with the score-based Library instruments

There are several things that you must do before you can create a soundfile using the score-based Library instrument algorithms:

1) Create a *Csound* **orchestra file** that includes one or more of the Library algorithms:

A script called *mko* ("**m***a***k***e* **o***rchestra file*") (discussed below) can be used to automate all or most of this process. This step might be likened to pushing some buttons on a hardware MIDI synthesizer or sampler in order to select one or more particular timbral "patches" (or "programs") for use.

2) Create a **score file** for this orchestra to play:
• First, using the command *sctp*, obtain a *score11* score template for one or more instruments in your orchestra, capturing these templates into a file; then
• edit the template(s) in this score file, typing in the *p-field* values you want; then
• run this edited score file through *score11* to produce the actual score file, in *Csound* format, that *Csound* will use to compute the soundfile.

This step could be likened to recording some performance data with MIDI controllers into a hardware or software sequencer, but not yet being able to hear the results of your playing.

3) Use the executable **csound** command to compile your orchestra and score files, compute the samples and write these samples into a soundfile in your current working soundfile directory on the *snd* disk.

Alternatively, you can use the command *csoundplay* to run *Csound* in real-time mode. Instead of writing the samples into a soundfile, *Csound* will pass them to the system audio hardware for digital-to-analog conversion and instant playback.

The three numbered steps above are covered in the following three subsections.

8.2.1. Preparing orchestra files

There are two parts to most Csound orchestra files:

1) a short **header**, which specifies certain required global variables, used by all of the instruments in your orchestra. These variables determine important format characteristics of the soundfile, as well as how it will be computed.

2) definitions for one or more instrument algorithms

Three global variables for the orchestra are set in the header:

• *sampling rate* (the Csound global variable *sr*) : Csound can generate output sound samples at any sampling rate that is usable on the sound card of a system. The most common sampling rates used in the ECMC studios are 44100 and 96000. The sound cards on the ECMC production systems support abitrary sampling rates up to 96k. Sampling rates lower than 44100, such as 32000 or 22050, occasionally are useful to avoid glitches when employing complex algorithms with real time output, or for initial tests, to speed things up when the highest possible output quality is not required.

• *control rate* (the global variable *kr*) : We don't need to compute 44100 or 96000 values per second to accurately represent a vibrato at 5 hertz. Computing this vibrato signal at a much lower rate, perhaps somewhere between 1000 and 5000 times per second, will provide a perfectly acceptable result and save a little computation time. *Control* signals are not heard as audio signals themselves, but rather are used to impose a time-varying envelope on some parameter of an audio signal such as vibrato (periodic time varying pitch changes), glissandi, or amplitude fade-ins and fade-outs.
Control rates are often set quite low —perhaps as low as 1/100 of the sampling rate —for initial tests, and then can be increased for production runs. In setting the control rate, there is only one restriction: The *kr* must be evenly divisible into the sampling rate. For a 44100 sampling rate, control rates of 2205, 4410 and 8820 are common. When the sampling rate is set to 96000, typical control rates are 2400, 4800 or 9600.

• number of *output channels* (the global variable *nchnls*) : Csound can create an arbitrary number of output channels and pan audio signals between these channels. When working with the 4 channel audio system in room 52, you will set this variable to 1 (mono), 2 (stereo) or 4 (quad).

The higher the sampling rate, the control rate and the number of output channels, the more load the Csound compile job will put on a system. However, with today's computers, this usually is not an issue.

Creating orchestra files with mko

Unix programs that begin with the character string *mk* ("**m***a***k***e*") usually automate a series of several commands or operations to produce a usable output. Using the **mko** ("**m***a***k***e* **o***rchestra file*") script we can specify the header values and instrument algorithm(s) we wish to use as arguments on a single command line. With these input arguments, the script will

(1) create a scratch macro file named *temp.orc*. A *macro* is a short character string that is used either as an abbreviation for a much longer string of characters, or else as a pointer to particularly files or other system resources.
(2) Next, *mko* will run this source file through a Library program called *m4orch* to expand the macros into a usable *Csound* orchestra file, called *orch.orc* (which can be abbreviated *orc*) in your current working Unix directory.

The command line syntax of *mko* has one required argument and several optional arguments that can be included to override defaults:

mko [- or -nh] [SR] [kr KR] [NCHNLS] INSTRUMENT(S) [-O scratchfile]

where

• *SR* (optional) sets the sampling rate. The default is 44100.

Based on the sampling rate you provide, *mko* will select an appropriate *KR* (control rate) argument automatically. Usually this will be 1/10 of the sampling rate. In order to override this default control rate:

• include the flag *k2* followed by a space and the control rate you want.
• *NCHNLS* (optional) specifies the number of output channels, and usually should be set either to *1*, the default, for *mono*, or else to *2* for *stereo* ;
• *INSTRUMENT(S)* is a list of one or more names, in CAPS, of any Library instrument algorithms

you wish to include, such as *MARIMBA* or *GRAN*.

Additional command line options for advanced users:
If the first argument on your command line is a minus sign (**-**), this flag instructs *mko* NOT to expand the source file into an *orch.orc* file.
This can be useful if you wish to edit and modify this source file too add further signal processing, and then run their edited version of the file through *m4orch* to create the Csound orchestra file.
A **-O** (capital *O*, not a zero) flag, followed by a file name, instructs *mko* to write the source file into *filename* rather than into the default (and frequently overwritten) *temp.orc* file. Again, this is generally most useful if you wish to edit the file before expanding it, and don't want this edited file to be overwritten.

Unless your command line also includes the **-** ("no expand") option, *mko* will expand this alternative source file into a Csound *orch.orc* file. This file can be abbreviated *orc*.

Example 1:	Example 2:	Example 3:
mko MARIMBA	*mko 96000 2 GRAN*	*mko 48000 kr 9600 2 SAMP TSAMP*

Example 1: An Csound orchestra file named *orch.orc* (a.k.a. *orc*) is created for Library instrument *marimba*. All defaults are used for the header. The sampling rate will be 44100 and the orchestra output will be mono.
Example 2: A stereo orchestra file with a sampling rate of 96k is created for Library instrument *gran*.
Example 3: A stereo orchestra file that includes Library instrument algorithmns *samp* and *tsamp* is created. The sampling rate is set to 48000. Normally this would set the control rate to 4800. To achieve slightly higher resolution for all control rate signals we have overridden this default and have raised the *kr* rate to 9600.

For the curious: How mko works

When you type the command in Example 1 above, *mko* first parses your arguments, searching for arguments that make sense as sampling rate and number of channels arguments. If it does not find a sampling rate or a number of channels argument, it fills in default values. Next, it converts your arguments in macros and writes these macros to a temporary file called *temp.orc* that looks like this:

```
SETUP(44100,4410,1)
MARIMBA
```

Next, *mko* runs this scratch file through a Library script called *m4orch*, which also parses the file, searching for macros that are defined in Library configuration files. *SETUP* is one of these macros. It indicates that the three arguments that follow in parentheses, separated by commas, are respectively the sampling rate, control rate and number of channels to be used in the orchestra header. *m4orch* also recognizes the macro *MARIMBA*, which "points to" the file containing Csound code for instrument marimba. If *m4orch* encounters an argument that it does not recognize as a macro, it returns an error message. Otherwise it expands all of the macros into Csound code and writes this code to the file *orch.orc*.

8.2.2. Preparing score files

After (or, if you prefer, before) creating a usable Csound orchestra file (*orch.orc*), you'll want to give your band something to play. To accomplish this, you need
(1) to create an input file to *score11* for the instrument(s) in your orchestra,
(2) then edit this file to fill in parameter values, and then
(3) run the file through *score11* to produce a Csound score file.

To obtain a template for this score file, type:

sctp INSTRUMENT(S)

where *INSTRUMENT(S)* are the names of the Library instrument algorithms, in lower case, you wish to use.
Examples:

sctp marimba

will bring a score template for instrument algorithm *marimba* to your shell screen. To redirect this template into a file, instead of bringing it to your shell window, use the Unix redirect symbol > followed by a file name.

```
sctp  samp tsamp  >  samptest1
```

Result: Templates for instruments *samp* and *tsamp* are written into a file named *samptest1*.

While you are learning to use particular Library instrument algorithms, I recommend that you generally limit your scores to a single instrument. Even one of these puppies may cause you quite enough aggravation during the first few attempts. Even most advanced users generally create most of their *Csound* soundfiles using only one or two instruments. Questions of balance and spatial placement generally are resolved more quickly and easily by mixing several source soundfiles together, rather than by trying to compute the whole thing correctly in a single monster *Csound* job.

Each instrument algorithm is different. Some require only a few parameters while others incorporate a great many parameters. However, I have tried to apply a consistent format in numbering the most common *p-fields*. For most of the sound generating instrument algorithms:

p3 determines the *start time* of each note
du (*duty factor*) determines the *duration* of each note
p4 determines *pitch*, which usually can be supplied either in *pch* format (using *notes* in *score11*) or else directly in hertz (*cps*)
p5 determines *amplitude* level, on a 16 bit integer scale of 0 to 32767 (maxamp) or, sometimes, a floating point scale of 0 to 1.0 (maxamp)
p6 determines *attack time* (the time, at the very beginning of a note, during which the amplitude rises from 0 to the *p5* value)
p7 determines *decay time* (the time at the very end of the note during which the amplitude decays to 0)

Several instrument algorithms that simulate percussive timbres, such as *marimba* and *drums*, have envelopes that consist entirely of short attacks and much longer decays. However, algorithms that simulate sustaining instrumental timbres, such as *bsn* (bassoon), include an additional *p-field* (usually *p8*), labeled *atss* ("**at***tenuation of* **s***teady* **s***tate*"). This *atss p-field*, usually a multiplier for *p5*, determines the amplitude level at the point just before the decay begins.
Two-note example:

p5	10000	< peak amplitude
p6	.1	< attack time
p7	nu .3/.5	< decay time
p8	nu .6/1.6	< atss

Result: For both of the two notes in this score, the amplitude will rise from 0 to 10000 (*p5*) over .1 second (*p6* attack time). For the first note, the amplitude will gradually decrease from 10000 to 6000 (*p8* * *p5*), and then will decay to 0 during the final .3 seconds of the note (*p7*). For the second note, the amplitude will <u>increase</u> from the initial peak of 10000 up to 16000 (possibly producing the effect of a crescendo), then decay to 0 during the final .8 seconds. The amplitude envelopes for these two notes would look like this:

amplitude envelope of first note

```
             --- ..___
                     ---- ___
         -                    ----  ____
                                         -----  ....__________
        -                                                     -
       -                                                       -
                                                                 -
   -                                                               -
```

rise	"peak"	"steady state"	atss	decay
time:	amplitude:	portion	value:	time:
.1	10000	of note	.6 = 6000	.3 seconds

A sample score template for Library algorithm carillon

A score template for Library algorithm *carillon* is reproduced below. We have added line numbers (which do not appear in the actual template) here to facilitate the discussion that follows.

```
 1   <   Score file for Library algorithm  ### carillon ###
 2   < Functions needed: 61 100
 3   * f61 0 65 5 1. 64 .01;    <exponential decay
 4   * f100 0 1024 10 1.;       < sine wave

 5   CARILLON
 6   rd
 7   p3
 8   du 306.2;
 9     < p4 = pitch : if > 13. = cps, else pch
10   p4 no
11     < p5 = amplitude
12   p5
13     < p6 = attack time (normal range .005 - .02)
14   p6
15     < p7 = attack hardness  (1. = ord. ,  normal range .7 - 1.5)
16   p7
17     < p8 = brightness ( 1. = ord. , normal range .4 - 1.5)
18   p8
19     < p9 = percent tremolo (normal range .04 - .09,  use 0 for no tremolo)
20   p9
21     < p10 = tremolo rate (normal range 3. - 7.)
22   p10
23   p11          < a detuning p-field generally used only in chorused scores
24   end;   <<< End of ###  carillon  ### score  >>>
```

Whenever *Score11* encounters the comment symbol **<** , as on the first two lines of this template, it ignores the rest of the line. Comments are shown in italics in the example above.

The function definitions on lines 3 and 4 define an exponential decay and a sine wave, the only functions required by this algorithm. All functions required by an instrument algorithm are provided near the top of its score template.

Lines 5 through 24 comprise the body of the score file, and generally alternate between explanatory comment lines and the parameter value lines (indicated in bold type above) that we must edit to create our score. On line 5 we must add two or three arguments to specify a *start* time for our score (generally *0*) and the number of beats, or else notes, that the instrument will "play."

On line 7 (*p3*) we must create a rhythm (starting times for each individual note) by employing a *score11* keyword routine such as *rhythm, move* or random selection. Line 6 enables us to add any amount of **r***andom* **d***eviation* to this rhythm. For a strictly metronomic (and possibly mechanical-sounding) performance, simply leave the *rd* line blank. For a more "human-sounding" performance add a small value such as .02, which will cause all notes except the first to deviate by somewhere between +20 milliseconds and -20milliseconds from a rhythmically "perfect" performance. Line 8 (**du***ty factor*) specifies a default maximum duration for each note. The MAN page for this algorithm explains this default, and situations in which you would want to change it.

The commented lines that precede each remaining *p-field* identify which aspect(s) of the sound are controlled by the parameter. These comments generally provide a suggested range of values from which to choose, and, often, a default, "neutral" **ord***inary* value. Comment line 9 tells us that our pitch arguments in *p4* will be interpreted in *cycles per second* (hertz) whenever the value is greater than 13., otherwise in octave-pitch-class notation (if we employ the *Score11* keyword *notes*, which has been filled in on line 14 since it is the most common means to specify pitch). *p11* can be used to detune pitches specified in *p4*, but this generally is done only in chorused scores, and is discussed in the MAN page to the *chorus* utility.

To help us get a handle on these *p-fields*, and to hear the results of various *p-field* values and *Score11* keyword operations before we begin the task of creating our own score, we can consult one or more *example scores* that are available for each instrument algorithm. If we type

lsex sc ("*list example scores*"),

we will find three example scores available for the *carillon* algorithm: *carillon1*, *carillon2* and *carillon3*. To view one of these tutorial example scores, type

getex filename.

To hear a soundfile compiled from this score, play the *sflib/x* soundfile with the same name as the example score.

Editing a score file and running this file through Score11

We should now be ready to edit the template(s) included in our *score11* input file with a text editor, filling in the *p-field* values we desire. When our score input file is ready, we must run it through *score11*:

score11 scorefilename

This will create a usable Csound score file called *sout* (short for "**s***core* **out***put*"). We should take a look at this *sout* file to make sure we are getting what you want in each *p-field*.

For advanced users: When we run *score11*, our input score file actually passes first through the Library *m4* preprocessor, which expands all macros, then passes this expanded version to *score11* for processing. (For example, the instrument name gets converted by the Library *m4* script into a particular instrument *number*.) Should you wish to see your score file exactly as *score11* will see it, type: **m4expandsc filename**

8.2.3. Running csound

Creating a soundfile with Csound

After creating a *Csound* orchestra file (*orch.orc*) and a *Csound* score file (*sout*), we are (finally!) ready to lay down some sound with the **csound** command. The basic form of the this command is:

csound orch.orc sout

If you prefer (and most users do), you can use **cs** as an abbreviation for **csound**, and **orc** as an abbreviation for **orch.orc**:

cs orc sout

Result: *csound* will compile orchestra file *orch.orc* and score file *sout* into machine code; it will then create all of the functions specified in your score, loading them into memory and displaying them on your monitor. (On Linux systems, the function display by default appears in a small high resolution graphic window. You must click in this window to terminate the display.) Csound then will create a soundfile with the default name of *test*, compute the samples, and write these samples into soundfile *test*.

The *csound* command has a great many options, which are discussed in the MAN page for this command (available online and in hardcopy in the *Linux DOCS* binder, as well as in the *Csound Manual*). The most frequently used of these options are:

-o —specifies a name for the soundfile, in place of the default name *test*

-d —suppresses displays of the functions (if you don't wish to see yet another sine wave)

These options can be combined in your command line.

Example:

csound -d -o explosion.wav orc sout

Result: The functions will not be displayed, and the soundfile will be named *explosion.wav*.

The amount of computation time that *Csound* will require to create a complete soundfile depends upon several factors, including sampling rate and control rate, the length and complexity of your score, and the complexity of the instrument algorithm(s) you have called. Some simple *csound* jobs compute almsot instantly, but complicated instrument algorithms or textures may require considerably more number crunching.

While *Csound* is compiling a soundfile, it will send various diagnostic messages to your terminal. These diagnostics are called the *standard error message* (abbreviated *sterr*). In addition to actual error messages (such as amplitude clipping or bad *p-field* values that may cause a score note to be deleted), the *sterr* output provides other useful information. By watching the *sterr* output, for example, you can tell how much of your score has been compiled into the soundfile so far, and the maximum amplitude computed for each note. When all of the samples are written into the soundfile (the *Csound* job is completed and terminated), you can play the soundfile.

For advanced users: Various types of *job control*, discussed and illustrated in section 2.7, can be applied to *csound* or *mkcsound* jobs, allowing us to run such jobs in the background and redirect the *sterr* message into a file, or to run a series of jobs (*batch jobs*) in succession.

Running Csound in real-time mode: the csoundplay command

Csound has the ability to redirect the output samples so that instead of being written to a soundfile, they are passed directly to the soundcard for digital-to-analog conversion and listening in real time. The ECMC utility **csoundplay** (which can be abbreviated **csplay** or simply **csp**) can be used to automate this operation. The command syntax is

csoundplay [OrchestraFile] [ScoreFile]

The default orchestra file is *orch.orc*, and the default score file is *sout*.

As one would expect, real-time playback works best with fairly simple instrument algorithms and with scores that do not include many simultaneous notes. Sometimes, especially with instruments such as *marimba* and *carillon* that calculate sounds from scratch with multiple oscillators, it may be necessary to lower the sampling rate to avoid glitches in the sound. Once we have the score working to our satisfaction, we can reset the orchestra file sampling rate to 44100 or 96000 and run *csound* in the normal manner to create a soundfile. Consult the *csoundplay* MAN page for full details and bugs.

8.3. Using *cecilia*

Cecilia is a front-end "environment" for running Csound jobs. To open *Cecilia*, type *cecilia* (or else the abbreviation *cec*) in a shell window. If you don't want the application to tie up this shell window while it is running you can follow this command with an ampersand:

cec &

Cecilia enables each user to customize his/her working environment within the application. Individual user preferences can be set by selecting *Preferences* under the *File* menu, then editing values in the window that opens. Your personal preferences are stored in a file called *.ceciliarc* in your home directory.

We have added some resources developed here at Eastman to the standard *Cecilia* distribution. In order to access these Eastman resources, and also your home soundfile directory while working within the application, however, your *.celprefs* must include certain definitions and pointers to these Eastman additions. To create a *.celprefs* file that includes these items, type

mkcecprefs

in a shell window. After creating this file, you can edit it at any time while you are working within *cecilia* by selecting *Preferences* under the *File* menu. In the *Modules* window advanced users can add the names of directories that include their own Csound orchestra and score files. Clicking on the *Set Utilities* button allows us to add our own choice of utility programs to one's working environment. Clicking on the *Set Service Applications* button allows one to set soundfile play, info or editor preferences, but it is unlikely that you will ever want to change the default *Service Application* choices we have established.

Eastman modules added to Cecilia

A *cecilia* "module" consists of an orc/score pair —an orchestra file and a companion score file to "play this orchestra." Note that the score file must be a *Csound* score file, such as the *sout* output files created by *score11*), and not an input file to *Score11*. The directories (groups of modules) that we have added to Cecilia include

(1) modules for the *Eastman Csound Library*, which includes score p-field based instrument algorithms such as *marimba* and *samp*, as well as MIDI-based *midiins* algorithms such as *midisamp*;

(2) modules for all of the orchestra/score file examples in the *Eastman Csound Tutorial* document.

To access one of these groups of modules:

- Click on *File* in the main *cecilia* window, then on *New*.
- In the list of available modules that appears, you should see *ESMtutorial* and *ESMlibrary* near the bottom of the list.
- Clicking on *ESMtutorial* or *ESMlibrary* will pop open a listing of all of the "modules" available within this directory, as well as
    any subdirectories with additional modules, and
    helpful *README* files
- Click on a module to select it and load the appropriate files into Cecilia for editing or compilation. *README* files are opened in the same manner. Click on the *Info* button to display the contents of a *README* file, or information about using a module file.
  You can make all of these selections in quick succession by holding the mouse button down continuously until you have found the module you want.

A *CEC editor* window will open, which will include (1) an orchestra file (either mono or stereo, or both, if the algorithm is set up to create either mono or stereo output) and (2) a score file.

The *ESMlibrary* modules
The modules within the main *ESMlibrary* directory include score p-field based instrument algorithms from the *Eastman Csound Library*, such as *samp* and *marimba*. If you select one of these algorithms in the main *ESMlibrary* directory, the orchestra code for the algorithm will be loaded into the orchestra file section of the cecilia *editor* window, but there will be no score file. You must create your own score file in a shell window. Then, in the *editor* window, select *Module*, then *Open Score File*, and then, in the window that opens, the name of the score file (usually *sout*, assuming that this file was created by means of *Score11*).

You now are ready to compile this orc/score pair in the main *cecilia* window, using either

- the *Preview* button to send Csound's output samples directly to the DACs for listening, or
- the *Write* button to write the samples into a soundfile.

If you choose to write a new soundfile, name of this soundfile is determined in the *name* box in the main cecilia window. Since the default names are ugly (the name of the "module" followed by *.AIFF*), you probably will want to change this name to something simpler.

The *EXAMPLE* subdirectory:

The *ESMlibrary* menu includes a subdirectory called *EXAMPLES*. If you select this directory, you will see a list of available orc/score pairs, used to create example soundfiles in the *sflib/x* directory, that illustrate typical usage of the instrument algorithm. These examples, with names such as *marimba1*, *samp2* and *tsamp3*, are identical do those that can be accessed with such commands as *lsex* and *getex*, except that the input file to *Score11* has already been converted into a *sout* file. Clicking on one of these modules loads it into the cecilia's editor, where it is ready for immediate compilation.

The *MIDI* subdirectory:
Modules within the *MIDI* subdirectory include MIDI-based (*midiins*) algorithms from the Eastman Csound LIbrary, such as *midisamp* and *midiwave*. Default score files have been provided for each of these algorithms, so they are immediately playable (unlike the score-based algorithms in the main *ESMlibrary* directory). A *MORE_EX* ("more examples") subdirectory includes additional examples for some of the *midiins* algorithms. Be sure to read the *README* file within the *MIDI* subdirectory before trying to use these modules, and, after loading one of the *midiins* modules into Cecilia, click on the *Info* box for a quick usage summary.

Currently, the *MIDI* modules are useful only for playing through the DACs. do not try to write the Csound output into a soundfile.

The *ESMtutorial* modules
This group of modules includes all of the orchestra and companion score files presented in the *Eastman Csound Tutorial* document (Schindler). These modules are loaded into Cecilia in the same manner described above. Select

File -> *New* -> *ESMtutorial* -> (the module you want).

If you wish, you can edit the orchestra and/or score data within Cecilia's window window. Alternatively, you can create new scores in a shell window and read them into Cecilia with the *Open Score File* option under the *Module* menu in the Cecilia editor window.

Eastman Utilities

The Eastman utility programs we have added to those built-in to Cecilia are accessed under the *Utilities* menu in the main window. Currently, these utilities include the following:

mktutsffuncs : many of the modules in section 5 of the *ESMtutorial* group (such as *ex5-1* and *ex5-3*) require link pointers to particular soundfiles that are read in from the *sflib* directories. You can create all of these necessary links, without which some of these jobs will not run, by clicking on the *mktutsflinks* button. This will create a series of files in your current working soundfile directory called *soundin.1, soundin.2* and so on.

rmtutsffuncs to remove all of the link files in your soundfile directory that were created by the *mktutsflinks* command, when you no longer will be using modules such as *ex5-1* and *ex5-3*, click on this button.

8.4. Using the MIDI-based Library instrument algorithms

The instrument algorithms that accept note and "event" performance data from MIDI controllers (or from a MIDI file), rather than from *p-fields* within a score file, are contained within the *midiins* subdirectory of the Eastman Csound Library. To see a list of the currently available *midiins* algorithms, type

lsmidiins

For information, beyond that presented here, on the capabilities of these algorithms and on how to use them, consult the local help file *midiins*, or the final section of the hardcopy *Eastman Csound Library* binder in rooms 52 and 53.

As with the *score p-field* based instruments already discussed, some of the *midiins* algorithms generate purely synthetic audio signals, "from scratch," while others read existing soundfiles into RAM and then process the samples of these input sounds. In order to generate audio signals, or to process soundfile samples, the algorithm must be supplied with **function** definitions within a score file. Many of these *functions* are identical to those included within the score templates for the *score-based* algorithms, enable *Csound* to compute tables of numbers that represent one cycle of a synthetic wave form, or the sample data of a soundfile, or various types of curves or trajectories. However, the *midifuncs* files also include functions for *keymapping* (a table that determines which of the available input soundfiles is to be used in response to note-on signals from each of the 88 Clavinola keys) and for mapping MIDI controller data such as *velocity* to such processing operations as output amplitude.

All of the *midiins* algorithms require a set of functions, and with most of these algorithms one can choose from among alternative pre-defined sets, each of which will create a unique timbre or combination of timbres. Each of these pre-defined sets of function definitions is contained within a file in the *midifuncs* subdirectory of the Library. To see a list of currently available *midifuncs* files, type

lsmidifuncs

For information on the contents of these files, consult the local help file *midifuncs*.

By default, most of the *midiins* algorithms are *mono-in* and *mono-out*: they create a monophonic source audio signal, or else process monophonic input soundfiles, and also output a mono signal. However, this default monophonic output can be changed to stereo out, and various types of sound localization subroutines can be applied to determine the left-right spatial placement of each output sound.

A few instruments, however, are set up for *stereo-in/stereo-out* processing. The names of these "instruments" begin with the characters *ST*, like the algorithm *STmidisamp*. Generally, these stereo algorithms require stereo input soundfiles as well. The names of *midifuncs* files that incorporate groups of stereo input soundfiles, usable by an algorithm such as *STmidisamp*, also begin with the characters *ST*, such as the file *STwinds*.

MIDI controller data

Presently, the *midiins* algorithms have only been programmed to respond to controller input data that is available on the *Clavinola* and *MCS3*.[5] This likely will change in the future, so consult the *midiins* helpfile for information on further MIDI input possibilities that have been added since this writing. The MIDI

[4] It is possible to bring your own MIDI controllers into the studio and connect them to *MIDI in* jacks on the Studio MIDI interface. However, if an instrument algorithm has not been programmed to respond to a particular type of MIDI controller input, such as *aftertouch* or *breath control*, this input will have no effect.

signals currently recognized by the algorithms include:

• *note on* : A note is initiated when you depress a key on the *Clavinola*, or whenever a note event is encountered within a MIDI file. These *note-on* messages are accompanied by *note number* and *velocity* messages.

• *note number* : Each key on a MIDI keyboard (and additional possible "keys" below and above the range of an 88 key controller) is assigned a number between 1 and 127. The "middle C" key is number 60, which most often (but not always) is mapped to the pitch *c4* ("middle C," or 261.6 hertz).

• *velocity* : A the *velocity sensitivity* controller within the *Clavinola* or some other MIDI performance device measures the *quickness* (NOT the force or weight) with which a "key" is depressed.

• *note off* : A note currently sounding is terminated when you release the key on the *Clavinola* that initiated this note, or, in a MIDI file, when a "note off" signal is encountered for a currently "active" note

• The three *Clavinola* foot pedals:

☞ The *RIGHT* pedal (MIDI controller number 64) usually functions as a *"sustain"* pedal

This is a *note initialization* controller which sends out a single value at the onset of each note. The range of values should be from 0 to 127. However, the resolution of the data created by the right pedal of the *Clavinola* is so poor that this pedal essentially functions like a two-position *on/off* switch.

With most of the *midiins* algorithms, if you depress this pedal *BEFORE* playing a note, then play the note, and then release the key (and, if you wish, the pedal), the note will sustain for up to 5 or 6 additional seconds (assuming that the input soundfile duration is this long) with a gradual decay in amplitude. It is not possible to sustain a note or input soundfile at its full original amplitude with the right pedal. Given the poor resolution of this *Clavinola* pedal, it makes relatively little difference whether the pedal is depressed all or only part way down.

☞ The *LEFT* and *MIDDLE* foot pedals on the *Clavinola* (respectively MIDI controller numbers 67 and 66) each send out an *on/off* note-initialization signal at the beginning of each note. The pedal is sensed as being either "down" or "up" at the onset of a note, and subsequent changes in pedal position during the note have no effect.

With most of the *midiins* algorithms, these two pedals have been programmed to affect the articulation of notes, with the *left* pedal producing sharper, more staccato-like attacks and decays than normal and the *middle* pedal producing a smoother, more legato-like articulation. The resulting change in articulation quality is more apparent, and more useful, with some types of sound sources than with other types. See the *midiins* helpfile for more details.

8.4.1. Performing in real-time: mkcsoundmidiplay

The simplest way to use the *midiins* algorithms along with the *midifuncs* files is by running the program **mkcsoundmidiplay**. Using a series of arguments provided on a single command line, this script automates a series of operations and commands that cause *Csound* to

• accept real-time MIDI input (MIDI files cannot be used), and

• run in real-time playback mode, so that you instantly hear the results of your playing. Alas, no soundfile is written, so currently the only way to record what you play is to a DAT tape running in record mode in the Otari DAT deck.

The command line syntax for the script is:

mkcsoundmidiplay [SR] [CHAN] [GAIN##} [DUR##] FUNCTION INSTRUMENT

where :

☞ the [optional] *SR* argument determines the sampling rate (default = 44100)

☞ the [optional] *CHAN* argument determines the number of output channels (mono, the default, or else stereo)

Additionally, if soundfile inputs are used, the *CHAN* argument also determines

the number of input channels (default = mono), and

if *mono in/stereo out* is specified, a particular procedure for panning the mono input soundfiles to left-right stereo output locations

☞ the [optional] word *GAIN*, followed immediately (no space intervening) by a floating point number, specifies an output gain (amplitude) multiplier

☞ the [optional] word *DUR* also must be followed immediately (no space intervening) by number; this number specifies the maximum number of seconds for a "performance," after which the job will terminate. The default value is *DUR300* (300 seconds, or 5 minutes).

☞ the *FUNCTION* argument specifies the names of one or more function definition files (generally only one file) in the *midifuncs* directory

and

☞ the *INSTRUMENT* argument supplies the name of an instrument algorithm from the *midiins* directory

Mercifully, the command name *mkcsoundmidiplay* can be abbreviated either *mkcsmidiplay* or *mkcsmp* on your command line. If you type the command name (or one of its two abbreviations) with no arguments, a usage synopsis will be displayed.

Simple examples:

(1) *midiins* algorithm *wave* generates a (frequently boring but occasionally useful) synthetic fixed wave form timbre. (The timbre does not change during the duration of a note.) Several *midifuncs* files, each specifying a particular audio wave shape and resulting timbre, can supply the table of numbers that the algorithm samples. If we wish to make *wave* play with a mellow triangular shaped waveform, provided by *midifuncs* file *triangle*, and we are content to use all of the default *mkcsoundmdiplay* option values (sampling rate = 44100, mono output, no amplitude gain or maximum duration modification), we can launch our performance by typing

mkcsoundmidiplay triangle wave

After the script and *Csound* have created the necessary orchestra and score files and loaded the functions into RAM, their diagnostic *sterr* output to our shell window will freeze. We now can play triangle waves for up to five minutes, the default maximum "performance time," experimenting with different key velocities and with the three Clavinola foot pedals. Should we wish to terminate the job before the full five minute time limit —a very likely possibility —we can type a $\boxed{\hat{}c}$ (*control c*) at any time to regain control of our shell window.

(2) The *midifuncs* file *strings* includes sustained (4-6 second) contrabass, cello, viola and violin tones with a keymapping covering almost the full 88 key range of the piano (although the very highest violin tones may not be to our liking). To play these soundfiles with the *midisamp* algorithm, we could type

mkcsmp string midisamp

Optional arguments to mkcsoundmidiplay :

Sometimes it is necessary or desirable to tweak the optional arguments provided by *mkcsoundmidiplay*. For example, the "polyphony" (number of simultaneous notes) we desire may exceed the throughput capacity of *Csound* or of the disks or audio hardware, resulting in hiccups or glitches in the sound, or even in the audio buffer becoming clogged with garbage, necessitating manual termination of the job. (See the *BUGS* section of the *mkcsoundmidiplay* MAN page.) If this happens (or, based on your past experiences, you anticipate that it *might* happen), you can change the default 44.1 k sampling rate to a lower rate supported by the audio hardware, such as 32000 or 22050. (Any of these possible *SR* arguments can be abbreviated by its first two integers, if you wish; the argument *16* will set the *SR* to 16000, the argument *22* to *22050*).

Note that the output sampling rate is totally independent of the sampling rate of any input soundfiles used. *Csound* will perform sample rate conversion on the fly.

If the results of our playing result in amplitude clipping, we can run the job again and include a *GAIN* factor, such as

mkcsmp GAIN.67 alto1 midisamp

In this example, all output samples will be multiplied by .67, hopefully correcting the problem. Notice that the amplitude multiplier argument must immediately follow the word *GAIN*, in CAPS, with no intervening spaces.

Conversely, if what we play results in signal levels lower than desirable for good audio quality, we can specify that all output samples be multiplied by a gain factor of, say, 1.8:

mkcsmp GAIN1.8 alto1 midisamp

The *CHAN* ("audio channels") argument provides us with several spatialization methods for localizing mono source soundfiles or synthetic signals at various left-right stereo output positions. With *CHAN* option *1-2A* ("*1* in, *2* out, option *A*"), the lower the Clavinola key (MIDI note number) that initiates a sound, the further it will be placed to the left, and the higher the key, the further to the right. Option *1-2C* randomizes left-right output locations. Options *1-2D* and *1-2E* also randomize the initial placement of each note, but, in addition, introduce random (*1-2D*) or quasi-periodic (*1-2E*) spatial movement of the note throughout its duration. All of these option arguments also can be provided with a lower case letter (*1-2a, 1-2b,* and so on). See the man page for further details.

There are two reasons why one might wish to include a *DUR* argument to alter the maximum five minute performance time:

- in the unlikely event that we wish to play for more than 5 minutes; or
- to check the maximum output amplitude value produced by what we play.

When *Csound* terminates "naturally," by reaching the maximum performance time specified and then shutting down, it will print out an *sterr* message with the peak output value produced. If this value is fairly low —perhaps around 5000 or so —we can run the job again and include a *GAIN* argument to create higher output amplitude values and thus better signal resolution. When *mkcsoundmidiplay* is terminated manually, with a *control c*, it will NOT print out the peak amplitude.

Additional examples:

(1) *mkcsmidiplay 32 2-2 GAIN1.6 STwinds STmidisamp*

Result: The stereo source soundfiles in the *STwinds* function file are "played" by algorithm *STmidisamp*. The sampling rate is set to 32000, and all output samples are multiplied by 1.6.

mkcsmp 44100 1-2d DUR25 sine midiplunk

Result: Timbres reminiscent of plucked strings, generated by instrument *midiplunk*, will be created at a sampling rate of 44100. The output is stereo, with the tones randomly distributed, and randomly meandering about, between the two speakers. The performance will terminate after 25 seconds (if we don't abort the job earilier), and the peak output amplitude value will be displayed.

8.4.2. Additional scripts for using the midiins algorithms

If we wish to edit (alter, or append to) the Csound code for one of the Library *midiins* algorithms, or to edit a *midifuncs* file, or to create our own MIDI-based orchestra and score files (which may include function definitions for some of our own soundfiles), we cannot use the *mkcsoundmidiplay* command. Rather, we must perform several operations, similar to those required for use of the *score p-fields-based* Library instrument algorithms, before we can run Csound:

(1) Create an orchestra file that sets header values and then includes one or more *midiins* algorithms and/or our own MIDI-based instrument algorithm(s).

(2) create a score file for this orchestra that includes all of the function definitions required by the instrument algorithm(s) and a dummy function that sets a maximum performance time.

(3) Run the *csound* command, with appropriate flag options. If we wish to run *Csound* in real time, with MIDI controller input, the script *csoundmidiplay* can simplify the task.

Creating an orchestra file: mkmidiorch

A script called **mkmidiorch** (which can be abbreviated *mkmidio* or *mkmidiorc*) can be used to create a Csound orchestra file that includes an instrument algorithm (or, rarely, more than one algorithm) from the *midiins* directory. Note that this script functions much like the *mko* script, but enables us to fetch files from the *midiins* subdirectory, rather than from the *score p-field*-based *ins* subdirectory, of the Eastman Csound Library. The command line syntax of *mkmidiorch*, however, is very similar to that of the command *mkcsoundmidiplay*:

mkmidiorch [SR] [CHAN] [GAIN##} INSTRUMENT(S) [-O filename]

The *SR, CHAN, GAIN* and *INSTRUMENT* arguments function in exactly the same manner, and with exactly the same options and defaults, as in the *mkcsoundmidiplay* command. The *INSTRUMENT* file must exist in the *midiins* subdirectory; the default *SR* is 44100, the default *CHAN* value mono in and out, and the default

GAIN multiplier is *1*. Also by default, however, *mkmidiorch* will write its output Csound code into the file *orch.orc* (which can be abbreviated *orc*). This file frequently is overwritten. If we intend to edit this orchestra file, and want to guard against inadvertently overwriting it with a *mkcsoundmidiplay, mko* or *mkmidiorch* command, we can use the *-O* option to specify an alternative output file name.

Example *mkmidiorch* command lines:

(1) *mkmidiorch midisamp*

Result: The orchestra file *orch.orc* will be created (or overwritten), and will contain the code for *midiins* algorithm *midisamp*. The sampling rate is set to 44100, and the number of output channels to *2*.

(2) *mkmidiorch 32 1-2e GAIN1.3 midiplunk -O mdiplunk.orc*

Result: Orchestra file *midiplunk.orc* will be created (or overwritten), and will include the Csound code for Library algorithm *midiplunk*. The sampling rate is set to 32000 and the output channels to stereo, with random spatial positioning followed by quasi-periodic movement (*CHAN* option *1-2E) of the output sounds. The orchestra file specifies that the output samples be multiplied by a GAIN* factor of 1.3.

For more details, consult the *mkmidiorch* MAN page.

After creating orchestra files in this manner, advanced users can edit them, if desired, to alter the Csound code and/or to include some additional Csound signal processing subroutines.

Obtaining Library score and/or function files for the midiins algorithms: getmidiscore and getmidifunc

The command **getmidiscore** (abbreviation **getmidisc**) duplicates another of the individual steps included in the *mkcsoundmidiplay* command: the creation of a score file that includes a file (or, rarely, multiple files) from the *midifuncs* directory, and appends a dummy function (*f0*, or "function number 0") that sets a maximum performance time, after which a Csound job will terminate. The command line syntax is

getmidiscore [DUR##] FUNCTION(S)

As with the *mkcsoundmidiplay* command, the optional *DUR* flag, followed immediately by a number, changes the default 300 second (5 minute) performance time to the number of seconds specified.

Example: *getmidiscore vln*

Result: *midifuncs* file *vln*, followed by the two lines

```
f0 300
e
```

(which set the maximum performance time to 300 seconds) will be displayed in your shell window. To capture this output to a file, use the Unix redirect sign **>** followed by a filename:

getmidiscore vln > vln.sco

The resulting output file *vln.sco* can now be used as a Csound score file input (in place of the *sout*), along with an orchestra file that includes an algorithm such as *mmidisamp*.

Should you wish to capture only *function definitions* from a *midifuncs* file, without appending the *f0* performance time limit required in score files for MIDI-based algorithms, use the command **getmidifunc** instead of *getmidiscore:*

getmidifunc vln > vln.funcs

The resulting file *vln.funcs*, lacking the *f0* definition, cannot be used as a Csound score file. Presumably, we will wish to alter this file, or add some of our own function defintions, before adding the *f0 300* and *e* lines manually to turn this file into a usable Csound score file.

8.5. The samp and tsamp Library algorithms

Two of the most frequently used score-based Library instrument algorithms are **samp** (" *sampler*") and **tsamp** ("**T***ransposing sampler*"). These are "sampler" algorithms that enable us to read in groups of *monophonic* soundfiles, mix these soundfiles and apply various types of signal process (sound modification) algorithms to them. Although the input soundfiles must be monophonic, the output of *tsamp* and *samp* can be either mono or stereo. Companion versions of these algorithms, respectively named *STsamp* and *STtsamp*, can be used to read in <u>stereo</u> input soundfiles, process them, and output the resulting samples to a stereo soundfile.

There is only one major difference between the *samp* and *tsamp* algorithms:

☞ With *samp* (and *STsamp*), one specifies particular output pitches, most often in pitch class notation (by using the *score11* keyword *notes*). Microtonal tunings, and alternative ways of specifyng the output pitch, also are available.

☞ With *tsamp* (and *STtsamp*) by contrast, we specify desired transpositions, in half steps and, optionally, additional microtonal divisions. With non-pitched and quasi-pitched sound sources such as speech, drums and ambient sounds like a waterfall, it often is more convenient or intuitive to think in terms of shifting a sound up a minor third, or down a tritone, rather than to specify a particular "output pitch," such as *c4*.

[A real-time *midiins* instrument algorithm similar to *samp* named *midisamp* enables us to transpose and process groups of soundfiles that Csound "plays" in response to triggers from MIDI controllers. All references to MIDI-based Library algorithms and utilities in the discussion that follows are printed in small font and are enclosed in square brackets. You can ignore these small font bracketed inserts unless you will be using MIDI-based Library algorithms.]

The *samp/tsamp* algorithms also allow us to

• alter the amplitude envelope of the source sounds, by adjusting "peak" and "steady state" levels and/or by adding fade-ins and fade-outs of arbitrary durations
• add time varying changes in timbral brightness within individual tones, or from note to note
• add *chorusing* ("fatten" the sound, or give the illusions of two or more "players," by reading in the soundfile(s) more than once, with pitch detunings, delayed start times and other parameter variables for these "echos")

Two other members of the *samp/tsmp* family, named *bigsamp* and *bigtsamp*, are respectively identical to *samp* and *tsamp*, but in addition include p-fields than enable us to add several types of time varying pitch inflections, such as glissandi, vibrato, random frequency deviation. amplitude tremolo and random amplitude deviations.

In order to read soundfiles into Csound for processing, the *samp/tsamp* and *gran* Library instruments employ function definitions. A function (table of numbers) must be defined within the score file for input soundfile to be used. Pre-defined function files that incorporate groups of *sflib* soundfile inputs are available within the *funcs* subdirectory of the Eastman Csound Library. To obtain a list of these functions, type

lsfunc (or else *lsfuncs*)

[The comparable command to display a list of available Library soundfile function definitions for MIDI based Library instruments is: *lsmidifunc*]

To display one of these function files, type : **getfunc functionname**
(where *functionname* is the name of a function file) in a shell window. Examples:

getfunc vln

will display function file *vln* (a group of 44.1k multisample violin tones from */sflib/string*)

getfunc zimb zith1

will display function files *zimb* (zimbalon multisamples) and *zith1* (a zither multisample collection)

[The comparable command to obtain a soundfile function definition file for MIDI based Library instruments is: *getmidifunc*]

To include one or more of these function files within a score template for an orchestra that includes *samp, tsamp, STsamp* or *STtsamp,* type

sctp f FunctionFile s samp (or *tsamp*)

The *f* flag tells the script that one or more function files, specified in the following argument(s), are to be included. The *s* flag indicates that the following argument(s) are *score11* template files. Example:

sctp f strings s samp

Result: The function file *strings* will be included within a score template for instrument algorithm *samp*.

Complete documentation on how to use these algorithms can be accessed by typing

man samp or *man tsamp*

A hardcopy printout of this *man* page, along with score templates and example scores, is included in the *Eastman Csound Library* document in rooms 52 and 53.

8.6. Understanding input soundfile and keymap functions

At some point, it is likely that you will wish to use some of your own soundfiles with *samp*, *tsamp* or *gran*. Or you might wish to create your own keymapped collection of *sflib* soundfiles, or perhaps a hybrid collection that includes both *sflib* soundfiles and one or more of your own soundfiles. In order to do this, you must create function definitions, comparable to those within the Library function definition files, for each input soundfile. If you will be using these functions with Library algorithm *samp* or *bigsamp*, you also will need to create additional functions for keymapping, which will tell *samp* which input soundfiles to use for which pitches. Normally, creating these function definitions would require familiarity with the syntax of Csound *Function Table* statements, and with some of Csound's function generating *GEN Routines*. These topics are beyond the scope of this tutorial (and also beyond the interests of many beginning users). However, even if you have little knowledge of the syntax or workings of Csound, you can employ some Library scripts to create files that include function definitions for input soundfiles, as well as keymapping and other ancillary functions. Even for beginners, though, is helpful to be able to interpret (and thus, if necessary, to be able to debug) the function definitions within the file you create.

If we type

getfunc xylo2

the following set of function definitions, suitable for inclusion in an input file to *score11*, will be displayed:

```
  < ** xylo2 (xylophone medium mallets) functions ***
* f1 0 131072 -1 "/sflib/perc/xylo2.a4.wav" 0 0 0 ; < dur = 1.94
* f2 0 131072 -1 "/sflib/perc/xylo2.cs5.wav" 0 0 0 ; < dur = 1.86
* f3 0 131072 -1 "/sflib/perc/xylo2.fs5.wav" 0 0 0 ; < dur = 1.49
* f4 0 131072 -1 "/sflib/perc/xylo2.b5.wav" 0 0 0 ; < dur = 1.75
* f5 0 65536 -1 "/sflib/perc/xylo2.e6.wav" 0 0 0 ; < dur = 1.42
* f6 0 131072 -1 "/sflib/perc/xylo2.a6.wav" 0 0 0 ; < dur = 1.56
* f7 0 65536 -1 "/sflib/perc/xylo2.d7.wav" 0 0 0 ; < dur = 1.08
* f8 0 65536 -1 "/sflib/perc/xylo2.g7.wav" 0 0 0 ; < dur = 1.02
* f9 0 32768 -1 "/sflib/perc/xylo2.c8.wav" 0 0 0 ; < dur = 0.635
 <; f99 = input soundfile function numbers & split points {in MIDI note numbers}
*f99 0 128 -17 0 1 71 2 75 3 81 4 86 5 91 6 96 7 101 8 105 9 ;
 <; f98 = base pitches of input soundfiles {expressed in MIDI note numbers}
*f98 0  16  -2 0 69 73 78 83 88 93 98 103 108 ;
<  **** End of XYLO2  functions  *****
```

Following the syntactical requirements of *score11*, comments begin with a < symbol, each function definition is preceded by an asterisk (*****) and each line of input to *score11* (including function definitions) must end with a semicolon, which specifies the end of a line.[6]

Within this display there are two types of function definitions:

1) Function numbers *1* through *9* (the lines beginning **f1* through **f9*) instruct Csound to create and load into RAM nine tables and, within each table, to read in all of the samples from one of the *sflib/perc/xylo2* soundfiles. Library instrument algorithms *tsamp*, *bigtsamp* and *gran* will use only these nine functions, ignoring function numbers 99 and 98, for which they have no use.

2) The function definitions at the bottom of the file —**f99* and **f98* —create additional tables of numbers that are used by the *samp* and *bigsamp* algorithms in addition to the nine soundfile function definitions. The tables created by functions 99 and 98 include *MIDI note number* data and, together, create a *keymap*, specifying which of the nine input soundfiles will be triggered by the *output pitch* you specify for each note.

The first four *p-fields* (or arguments) within any Csound function definition specify, respectively,

p1 : the function number,

p2 : the time within the score (normally zero, as in all of the functions above) at which the function

[5] A "line" of *score11* code can exceed 80 characters, can include carriage returns, and can extend over any number of physical lines, until terminated by a semicolon. A line of Csound orchestra or score file code, by contrast, can NOT include carriage return characters.

will be created,
p3 : the size of the table (in bytes) and
p4 : the Csound function generating subroutine that will create the function (*-1* in the examples above)

The actual data within the table is determined by the following *p-fi eld* arguments:

(function parameters 1-4)	(data within the table)	(comments)
* f1 0 131072 -1	"/sflib/perc/xylo2.a4.wav" 0 0 0	; < *dur = 1.94*
* f2 0 131072 -1	"/sflib/perc/xylo2.cs5.wav" 0 0 0	; < *dur = 1.86*
*f99 0 128 -17	0 1 71 2 75 3 81 4 86 5 91 6 96 7 101 8 105 9 ;	
*f98 0 16 -2	0 69 73 78 83 88 93 98 103 108 ;	

How keymap functions work:

Keymap functions make use of MIDI note numbers to represent pitch and split points, even (as is usually the case) when MIDI input is not being used. This might seem odd, but there is a simple reason: MIDI note numbers are consecutively numbered integers, a requirement if this kind of Csound function table is to work correctly. MIDI note numbers alsoare easier to deal with than fbating point hertz values, and they can be easily converted to hertz within the Csound code once they have served their purpose. *f99* contains the numbers of the soundfi le function defi nitions (1 through 9 in the *xylo2* example) alternating with split points for these functions, expressed in MIDI note numbers. The data arguments in this table alternate between

• *LOW* key values (the lowest MIDI note number that will trigger a soundfi le)
and
• the number (here abbreviated *NUM*) of the function to be triggered by these keys

	LOW	NUM	LOW	NUM	LOW	NUM	LOW	NUM	etc.
f99 0 128 -17	0	1	71	2	75	3	81	4	etc.

Thus, *f1* (the soundfi le *xylo2.a4.wav*) will be triggered by input from "MIDI keys" (converted to hertz) 0 through 70 (*bf4*)
f2 (*xylo2.cs5.wav*) by "MIDI keys" 71 (*b4*) through 74 (*d5*)
f3 (*xylo2.fs5.wav*) by "MIDI keys" 75 (*ef5*) through 80 (*af54*), and so on.

f98 specifi es the *BASE* MIDI note number for each of the nine soundfi les, again expressed in terms of MIDI note number. This *BASE* number is the MIDI key that will cause the soundfi le to be played at its original pitch level, without a downward or upward pitch transposition: 69 = *a4*, 73 = *cs5 78 = fs5*, and so on. Using this value, the instrument can calculate the resampling ratio required to transpose the soundfi le to any higher or lower pitch level.

The **midinote** script

To gain some familiarity with *MIDI note numbers*, and how these numbers usually are mapped to equal tempered pitches, you can experiment with a local script called *midinote*. Typing

midinote

with no arguments will display a table of *MIDI note numbers* and equivalent pitches, expressed in *score11 "notes"* format, in octave pitch-class format, and in hertz. To display conversions only for particular *MIDI note numbers* and/or particular pitches expressed in *score11 notes* format, include the appropriate arguments on your command line. Example:

midinote as0 a4 92 112

will display:

MIDI note:	score11 notes:	pch:	cps (hertz):	Comment
22	as0	4.10	29.135	
69	a4	8.09	440.0	"tuning A"
92	gs6	10.08	1661.219	
112	e8	12.04		

For more details, consult the *midinote* MAN page.

8.7. Creating your own soundfile and keymap functions

Either one or two steps are involved in creating your own collection of soundfile inputs for use with *p-field*-based algorithms such as *samp* and *tsamp* [or for use with with MIDI-based instruments such as *midisamp*] :

(1) Creating a function definition for each input soundfile. These definitions will resemble the *f1* through *f9* functions within the *xylo2* files discussed in the preceding pages. If you are creating function definitions for use with *tsamp* or *gran*, this is all you need.

(2) If instead you are creating function definitions for use with *samp* [or for use with a MIDI-based algorithm such as *midisamp*] you also will probablly want to create *keymapping* functions similar to the *f99* and *f98* functions within the *xylo2* examples.

The scripts available to simplify these two tasks include:

	score p-field-based algorithms (*samp*, *tsamp*,*gran*)	[MIDI-based algorithms] (*midisamp*)
(1) scripts to create input **soundfile function definitions**:	**mksffuncs** (required by *samp*, *tsamp*,*gran*)	[**mkmidisffuncs]** [required by *midisamp*]
(2) scripts to create **KEYMAP** functions for input soundfiles:	**mkkeymap** (often used with *samp*) (not used with *tsamp*, *gran*)	**[mkmidikeymap**] [required by *midisamp*]

Typing any of these four command names with no arguments will produce a usage synopsis.

Using mksffuncs [and **mkmidisffuncs**] :
Input to these scripts takes the form

mksffuncs soundfile1 soundfile2 ... soundfileN [> outputfile]

[or *mkmidisffuncs soundfile1 soundfile2 ... soundfileN [> outputfile]*]

where the "*soundfile*" arguments are the names of existing soundfiles within your own $SFDIR (current working soundfile directory) or within any of the *sflib* directories. Generally, these soundfile names should be typed in in order from the lowest pitched to the highest pitched. Each input soundfile you specify will be assigned a successively higher function number, beginning with *f1*. See the *mksffuncs* [or *mkmidisffuncs*] *man* pages for more options and details.

Example 1 : *mksffuncs bass1.b2.wav scream1.e3.wav scream2.a3.wav scream3.cs4.wav*

The result should look something like this::

```
* f1 0 524288 -1 "/sflib/voice/bass1.b2.wav" 0 0 0 ; < 8.37 sec
```

** f2 0 262144 -1 "scream1.e3.wav" 0 0 0 ; < 5.94 sec*
** f3 0 524288 -1 "scream2.a3.wav" 0 0 0 ; < 11.88 sec*
** f4 0 65536 -1 "scream3.cs4.wav" 0 0 0 ; < 1.09 sec*

If this looks good to you, redo the command, this time capturing the output in a file:

!! > functionfilename

Example 2: *mksffuncs cb.p.c1 sneeze SEC1/cry bamboo*

Result: *f1* will define soundfile *sflib/string/cb.p.c1.wav* (a contrabass pizzicato tone), *f2* will define your soundfile *sneeze.wav*, *f3* will point to the soundfile *cry.wav* in your *SEC1* soundfile subdirectory, and *f4* will define the *sflib/perc* soundfile *bamboo.wav*.

8.7.1. Using mkkeymap [and mkmidikeymap] :

After creating input soundfile definitions in the manner illustrated above, we are ready to create keymap functions for these soundfiles (should we need them) by using the ECMC *mkkeymap* script [or the *mkmidikeymap* script if we will be using the *midisamp* algorithm]. The command line syntax is:

mkkeymap NUM BASE LOW NUM BASE LOW ... ... NUM BASE LOW

[or *mkmidikeymap NUM BASE LOW NUM BASE LOW ... ... NUM BASE LOW*]

Each of our input soundfile functions receives three arguments:

(1) *NUM* specifies the function number (1 for *f1*, 2 for *f2*, and so on.)

(2) the *BASE* argument specifies either a base MIDI note number or a base pitch. When this BASE MIDI key is played, or this BASE pitch is specified in a *score11* file, the soundfile will sound at its original pitch.

(3) the *LOW* argument specifies a keymap split point. MIDI keys (or output pitches) above this split point, but lower than the next split point, will trigger this soundfile The *LOW* argument for the first soundfile (*f1*) normally should be a 0 .

The *BASE* and *LOW* arguments can be given either in MIDI note numbers or else in score11 "notes" format (e.g. *c4* , *ef5* etc.)

Example: Suppose we now wish to create keymap functions for the four soundfile function definitions we created in Example 1 above, which look like this:

```
* f1 0 524288 -1 "/sflib/voice/bass1.b2.wav" 0 0 0 ; < 8.37 sec
* f2 0 262144 -1 "scream1.e3.wav" 0 0 0 ; < 5.94 sec
* f3 0 524288 -1 "scream2.a3.wav" 0 0 0 ; < 11.88 sec
* f4 0 65536 -1 "scream3.cs4.wav" 0 0 0 ; < 1.09 sec
```

Our command line might look like this:

mkkeymap 1 b2 0 2 e3 d3 3 a3 g3 4 cs4 c4

The resulting keymap function definitions will be displayed in our shell window:

< f99 = soundfile function numbers & keymap split points
f99 0 128 -17 0* **1 50 2 55 3 60 4 ;
< f98 = midi note numbers for base key { soundfile at original pitch}
*f98 0 16 -2 0 **47 52 57 61** ;

The keymapping parameters within *f99* and *f98* are highlighted here in bold type.

f1 (soundfile *bass1.b2.wav*) will be triggered by any output pitch up to MIDI note number 49 (*cs3*). It will sound at its original pitch when the output note is *b2* (MIDI note number *47*), and will be transposed for all other pitches.

Soundfile *scream1.e3* (*f2*) will sound at its original pitch when the output pitch is *e3* (MIDI key number *52*); it also will be triggered by any output pitch between *d3 (50)* and *fs3 (54)*.

f3 (which defines soundfile *scream2.a3.wav*) will be triggered by output pitches between *g3 (55)* and *b3 (59)*, and will sound at its original pitch level when the pitch is *a3 (57)*.

Output pitches at middle C (*c4*, MIDI note number 60) and above will trigger soundfile *screan3.cs4.wav (f4)*, which will be transposed for all notes except *cs4*.

If the keymap functions above look good they are ready for inclusion in an input file to *score11*. Redo the command and either

- capture it to a file, then read or paste this file into a score11 input file; or else
- append the output of *mkkeymap* directly intothe end of the score11 input file:

```
!! >>  score11filename
```

8.7.2. [Making keymap functions for MIDI-based Library instruments]

[Skip ahead to section 8.8 unless you will be using MIDI-based Library instruments such as *midisamp*].

The command line syntax of the *mkmidisffuncs* and *mksffuncs* scripts is identical, and they produce exactly the same output, except that the *mkmidisffuncs* definitions are in Csound score file format while the *mksffuncs* output is is *score11* format. The *mkmidikeymap* and *mkkeymap* scripts likewise share a common input syntax, and produce similar outputs.

To create soundfile function definitions and companion keymap functions for inclusion in a Csound score file intended for use with a MIDI-based Library instrument, with the same hypothetical input soundfiles as in the examples above, we would follow these steps:

(1) Create soundfile function definitions with *mkmidisffuncs*:

```
mkmidisffuncs bass1.b2.wav scream1.e3.wav scream2.a3.wav scream3.cs4.wav
```

This will display the following:

```
f1 0 524288 -1 "/sflib/voice/bass1.b2.wav" 0 0 0
f2 0 262144 -1 "scream1.e3.wav" 0 0 0
f3 0 524288 -1 "scream2.a3.wav" 0 0 0
f4 0 65536 -1 "scream3.cs4.wav" 0 0 0  ]
```

If this looks good, redo the command, this time capturing the output into a file: *!! > voicesounds.sco*

(2) Create companion keymap functions: *mkkeymap 1 b2 0 2 e3 d3 3 a3 g3 4 cs4 c4*

This will display:

```
   ; f99 = soundfile function numbers & keymap split points
f99 0 128 -17 0 1 50 2 55 3 60 4
   ; f98 = midi note numbers for base key { soundfile at original pitch}
f98 0 16 -2 0 47 52 57 61

f97 0 128 5 1 128 33              ;  veloc to non-linear amplitude
f96 0 128 5 .005 120 1. 128 1.     ;  brightness scaling for midisampbright
f95 0 64 7 -1 32 1 32 -1  ; triangular moving pan for CHAN option 1-2E

f0 300
e
```

If the function definitions returned by *mkmidikeymap* look correct, we can redo this command, this time appending its output to the end of our *voicesounds.sco* file, like this:

```
!!  >>  voicesounds.sco
```

This file is now ready for use (as a Csound score file) with the *midisamp* algorithm. All that we need do is create a suitable orchestra file, by typing something like

```
mkmidio  44100  1-2b  midisamp
```

and then play our voice sounds with the command

```
csoundmidiplay  orc  voicesounds.sco
```

8.8. Summary list of Eastman Csound Library commands

1. Commands to **LIST** Library files:

These commands take no arguments. The output will be displayed in your shell window.

Score p-field based algorithms:

lsins : list available *p-field*-based instrument algorithms

sctp : list *score11* templates for these algorithms
lsex sc : list example *score11* files for these algorithms
lsfunc (lsfuncs) :list available function files for these algorithms

MIDI algorithms:
lsmidiins : list available *midiins* instrument algorithms
lsmidifuncs : list available *midifuncs* function files

lscslib, lscsl : provide master list of Library files available on this system

---

2. Commands to **GET a COPY** of one or more Library files

These commands require a file name argument. By default, the output of all of these scripts will be displayed in your shell window. To redirect this output into a file, type

commandname argument(s) > **filename**

For a usage synopsis of any of these commands, type the command name with no arguments.

Score p-field based algorithms:
sctp : copies one or more *score11* template files for the specified instrument algorithm(s)
getex : fetches one or more specified **ex**ample *score11* files
getfunc (getfuncs) : gets one or more files from the *funcs* directory
MIDI algorithms:
getmidiscore (getmidisc) : creates a Csound score file that includes the specified *midifuncs* files

getmidisc [DUR##] FUNCTION

getmidifunc (getmidifuncs) : copies a *midifuncs* function file

---

3. Commands to prepare an **ORCHESTRA** file :

For a usage synopsis of any of these commands, type the command name with no arguments.

Score p-field based algorithms:
mko (mkorch) : automates creation of a usable Csound orchestra file called *orch.orc (orc)*

mko [-] [SR] [NCHNLS] INSTRUMENT(s) [-O sourcefile]

For situations where *mko* is not suitable, as when using global Library instruments such as *delays*, type in a file that includes Eastman Csound Library macrios:

SETUP(44100,4410,2)
GLOBALS
TSAMP([DELAYS])

Then type: *m4orch filename* to create the Csound orchestra file *orch.orc*.
m4orch (which can be abbreviated **m4o**) expands a source orchestra file with Library macro definitions into a usable Csound orchestra file called *orch.orc (orc)*
m4expand : works like *m4orch*, but brings the output to your shell window rather than writing to it the file *orch.orc*
Note: The similar command **m4expandsc** expands Library macros within an input file to *score11* so you can view the file exactly as *score11* will "see" it

MIDI algorithms:
mkmidiorch (mkmidio) : creates a usable Csound orchestra file that includes a *midiins* algorithm (A *MAN* page for this command is available.)

mkmidio [SR] [CHAN] [GAIN###] INSTRUMENT(s) [-O filename]

---

4. Commands to create **FUNCTIONS** for **INPUT SOUNDFILES**

For a usage synopsis of any of these commands, type the command name with no arguments.
Score p-field based algorithms:
mksffuncs : creates user specified input soundfile functions for use within a *score11* file for Library isnttruments such as

samp, tsamp and *gran*.

mksffuncs soundfi le1 soundfi le2 ... soundfi leN

mkkeymap : creates keymapping functions for these input soundfi le functions

mkkeymap NUM BASE LOW NUM BASE LOW NUM BASE LOW etc.

MIDI algorithms:

mkmidisffuncs : creates user specifi ed input soundfi le functions for use with *midisamp*

mkmidisffuncs soundfi le1 soundfi le2 ... soundfi leN

mkmidikeymap : creates keymapping functions for these input soundfi le functions

mkmidikeymap NUM BASE LOW NUM BASE LOW NUM BASE LOW etc.

---

4. Commands to run **CSOUND**

For a usage synopsis of any of these commands, type the command name with no arguments.

Score p-fi eld based algorithms:

csound (cs) : computes output samples and, by default, writes them to a soundfi le named *test* (*MAN* page available)

csound [flag options] OrchestraFile ScoreFile

csoundplay (csp) : runs Csound in real-time playback mode (*MAN* page available)

csp [OrchestraFile] [ScoreFile]

With no arguments the command uses the default fi les *orc* and *sout*).

cecilia : graphical front-end application for running *Csound*

MIDI algorithms:

mkcsoundmidiplay (mkcsmp, mkcsmidiplay) : automates running Csound in real-time playback mode, with MIDI input, using a *midiins* algorithm and a *midifuncs* fi le (*MAN* page available)

mkcsmp [SR] [CHAN] [GAIN##] [DUR##] FUNCTION INSTRUMENT

csoundmidiplay (csmp, csmidiplay) : runs Csound in real-time playback mode, with MIDI input, using the specifi ed orchestra and score fi les (*MAN* page available)

csmp [OrchestraFile] [ScoreFile]

With no arguments the command uses the default fi les *orc* and *sout*).

---

5. Miscellaneous commands

chorus : choruses a suitable *score11* input fi le (*MAN* page available)

mkcaltones : automates creation of a soundfi le with calibration tones (*MAN* page available)

---

6. For advanced users: Commands to make soft **links** to soundfi les or to spectral analysis fi les on the *snd* disk

Score p-fi eld based algorithms:

sflnk : creates a fi le called *soundin.#* in your *snd* directory linked to a soundfi le. (The # number of the soundin.# fi le is used by the Library *sf* instrument algorithms)

lplink — creates a fi le called *lp.#* in your current working Unix directory, linked to an *lpc* analysis fi le on your *snd* directory. (The "#" number of the lp.# fi le is used by algorithms *resyn, xsyn, gxsyn* or *pt.)*

pvlink : creates a link between a *pv.#* fi le and a *pvanal* fi le used by *Csound* unit generator *pvoc*

adsynlink : creates a link between an existing *hetro* analysis fi le and a new fi le called *adsyn.#,* used by the *Csound adsyn* unit generator

8.9. Some additional information on using the Library score-based algorithms

8.9.1. Library control shape functions

The Library *score11 template* files and *example score* files include all function definitions required by an instrument algorithm. Several of these instruments include subroutines that make use of control functions to vary some parameter (such as vibrato or tremolo rate) between two values within each note. In such cases, the control function determines the shape of the change between value 1 and value 2. (Note: It does not matter if value 1 is higher or lower than value 2)

The most commonly used control functions are:

f50 : linear change between value 1 and value 2

f60 : exponential change between value 1 and value 2

f54 : the change between value 1 and value 2 follows the curve of the first quarter of a sine wave

----

Symmetrical functions:

f52 : linear pyramid change from value 1 to value 2 and back to value 1

f52 : exponential pyramid change from value 1 to value 2 then back to value 1

f56 : change from value 1 to value 2 and then back to value 1 follows the shape of the first half of a sine wave

----

Value 2 to value 1:

f51 : linear change between value 2 and value 1

f61 : exponential change between value 2 and value 1

f54 : change between value 2 and value 1 follows the curve of the first quarter of a cosine wave (or second quarter of a sine wave)

Example:

Library instrument *bsn* has 3 p-fields for vibrato rate. In *p12* you specify an initial value, in *p13* a second value, and in *p14* a function to control the shape of the change between your *p12* and *p13* values. (In this instrument, the function is read exactly once per note.) Several possible control functions are provided in the template (functions *50, 51, 52, 53, 60, 61* and *62*). The following values:

p12 3.; <First vibrato rate
p13 5.; <Second vibrato rate
p14 nu 50/ 60/ 52/ 51; < Function for change in vibrato rate

would produce the following results:

First note: the vibrato rate increases linearly from 3 herz (beginning of note) to 5 herz (end of note)
Second note: the vibrato rate increases exponentially from 3 herz (beginning of note) to 5 herz (end). Most of the increase will come near the end of the note.
Third note: the vibrato rate increases linearly from 3 herz (beginning of note) to 5 herz (middle of note), then back to 3 herz (end of note)
Fourth note: the vibrato rate decreases linearly from 5 herz (beginning of note) to 3 herz (end of note)

Often, we wish to vary some value such as vibrato speed so that it sometimes increases within a note, sometimes decreases, and sometimes becomes faster and then slower, or else slower and then faster. For note-to-note variety, we also may wish to vary the *rate* (or *shape*) at which these changes occur —sometimes linearly (smoothly) and sometimes exponentially (more abruptly). All of these note-to-note variations could be achieved by using *p-field* values such as the following:

p12 1. 3. 5.5; < vary opening vibrato rate between 3 and 5.5 herz
p13 1. 4. 6.5; < vary second vibrato rate within a similar range
p14 sets 5 50 52 60 62; < use various linear and exponential functions

For advanced users: After you learn how to create function definitions, you may wish to substitute some of their own functions for those provided. Functions that will not be used in a score can be commented out, since they take time to compile and also take up memory space.

Advanced users may also wish to include some of the Library functions in their own Csound instruments and scores. All Library functions are available in a separate directory for such purposes. They can be listed by typing

lsfunc

To display one or more of these function definitions, type

getfunc *function(s)*

where *function* is the number or name of one or more Library functions. *getfunc* can also be used to redirect one or more function into one of your score files. The command

getfunc 50 60 >> myscore.s1

would copy definitions for library functions *50* and *60* to the end of a score file named *myscore.s1*

8.9.2. Global instruments

A few Library "effects" instrument algorithms, such as **rev** (a reverberator) and **delays,** (a delay line) do not generate audio signals themselves, but rather process (modify) signals that they receive from one or more other instruments. These **global** post-processing instruments require additional global variable definitions that must be included in an orchestra file before the names of any instruments to be used. These global variables are created by means of the macro **GLOBALS** on a line by itself. Additional macros can be used to route the outputs of other instruments into these global processing instruments, as in the following examples. Full details on these *GLOBAL* macros are provided within the MAN pages to these instruments.

The ECMC script *mko* is not well suited to creating orchestra files that include global isntruments. Rather, you need to type in a macro file, then expand these macros into the usable Csound orchestra file *orch.orc* by using te program *m4orch*:

m4orch   filename

Here are two example macro files to illustrate this process:

Example macro file 1	Example macro file 2
SETUP(4410,4410,2)	*SETUP(44100,4410,1)*
GLOBALS	*GLOBALS*
SFS([REVIN])	*BSN([DELAYSIN])*
REV	*CSBN([DELAYSIN])*
	DELAYS

Results: In *example 1* Library instrument *sfs* reads in existing stereo soundfiles and passes them to instrument *rev* for reverberation. In *example 2* instruments *bsn* and *cbsn* pass their output signals to the delay line *delays*, which adds echos, flanging, comb filtering or some other delay line effect.

Added user code

After all computations have been performed by an instrument for a sample pass, the output is labeled *a1*. If the instrument is stereo, the outputs are labelled *a1* (left channel) and *a2* (right channel). Most of the instruments are *monophonic* by default, but some (such as *rev* and *delays)* can be either mono or stereo, depending upon the value for *NCHNLS* in your header. All instruments send their output to the *standard output*, where it is added to the output of any other instruments currently active, and eventually written to the disk.

Advanced users can substitute any amount of additional Csound code for the standard *out a1* output statement in an instrument. The added code must be enclosed in matched delimiting pairs of parentheses and square brackets. Here is an example:

```
SETUP(44100,2205,1)
COMMENT add 2 delays to marimba output
MARIMBA([aout multitap a1, .1, .5, .25, .2
out aout])
```

This will add two echos, .1 and .25 seconds after the onset of each note, to the *marimba* output.
Note that the left **([** pair must immediately follow the instrument name, with no blank space intervening. The concluding **])** pair, however, can be preceded by blank spaces or a newline. Instrument definitions which do not have matching ([and]) pairs will cause the error message

Parentheses not matched

and the *csound* job will abort. More information on adding code is contained in the *Eastman Csound Tutorial.*

The COMMENT symbol : The ECMC universal comment symbol **COMMENT** can be used anywhere in a score11 input file or (as above) within a macro file used to create a Csound orchestra file in order to document what you are doing. Whenever *m4orch* or *score11* encounter a comment symbol, they ignore the rest of this line. The shorter comment symbol $\boxed{<}$ is more commonly used in *score11* input files.