From interface design to the software instrument - Mapping as an approach to FX-instrument building

Alex Hofmann
Department of Music Acoustics
University of Music and Performing Arts Vienna
Vienna, Austria
hofmann-alex@mdw.ac.at

Bernt Isak Wærstad
Department of Music, Music Technology Group
Norwegian University of Science and Technology
Trondheim, Norway
bernt.warstad@ntnu.no

Saranya Balasubramanian
Department of Music Acoustics
University of Music and Performing Arts Vienna
Vienna, Austria
balasubramanian@mdw.ac.at

Kristoffer E. Koch
Independent Electrical Engineer
Oslo, Norway
koch@kristofferkoch.com

ABSTRACT

To build electronic musical instruments, a mapping between the real-time audio processing software and the physical controllers is required. Different strategies of mapping were developed and discussed within the NIME community to improve musical expression in live performances. This paper discusses an interface focussed instrument design approach, which starts from the physical controller and its functionality. From this definition, the required, underlying software instrument is derived. A proof of concept is implemented as a framework for effect instruments. This framework comprises a library of real-time effects for Csound, a proposition for a JSON-based mapping format, and a mapping-to-instrument converter that outputs Csound instrument files. Advantages, limitations and possible future extensions are discussed.

Author Keywords

mapping, instrument design, Csound, real-time audio effects

ACM Classification

H.5.5 [Information Interfaces and Presentation] Sound and Music Computing, H.5.2 [Information Interfaces and Presentation] User Interfaces—Haptic I/O

1. INTRODUCTION

In contrast to sound production on acoustic instruments where musicians have to use finger actions e.g. on the piano [7] or apply blowing pressure like on the clarinet [5] to produce a sound, electronic instruments do not require direct player actions for sound production. Nevertheless, electronic instruments, based on analogue circuits mostly provide knobs and switches on a front panel to let the performer adjust parameters in the circuit to modify the sound. For example, this is similar to the function of clarinet keys that can be used to modify the pitch of the instrument [9].

When building electronic instruments with software, a common approach is using a *Music Programming System* (MPS) (e.g. Csound, PureData, Chuck, SuperCollider) [14]. Most MPSs make use of the unit generator principle. Unit generators are modules that contain digital signal processing (DSP) functions and the MPS allows the user to quickly combine these by either using a graphical or a text-based user interface.

For live electronics, hardware controllers are required to let performers physically interact with a software instrument on stage. A variety of innovative physical controllers can be found in the NIME Proceedings [12] e.g. full body tracking suits [8] or reductionist *one knob* interfaces [4].

Different methods of routing the sensed controller values to the arguments of a software instrument are discussed intensively in the NIME community and are referred to as the *mapping problem* [11]. During the years, different approaches were presented ranging from complex mapping libraries (e.g. the MnM toolbox [2] for Max/Msp, the Modality toolkit [1] for SuperCollider) to self learning tools based on different methods of machine learning [6, 13].

For the COSMO project, we designed a framework around the *Raspberry Pi* (RPi) to build Csound based instruments as standalone hardware devices [10][1]. The hardware framework comprises a custom designed shield for the RPi to connect up to 8 analogue controller inputs, 8 on/off switches and 8 LEDs, as well as a stereo analogue dry/wet circuit with true bypass if used as an effect processor. More than 20 COSMO hardware instruments were built in three workshop sessions held during the last year, with participants from different backgrounds (e.g. musicians, composers, artists, programmers, engineers). On the software side, we provide a pre-configured operating system for the RPi with example Csound effect instruments (details in Section 3.1, Table 1).

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

[1]http://cosmoproject.github.io

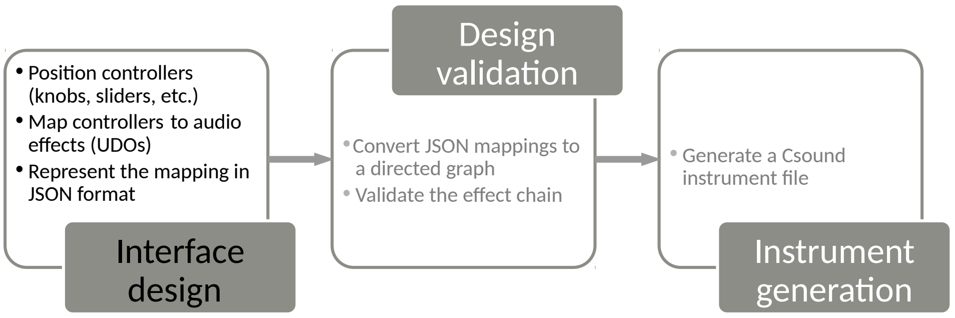
Position controllers (knobs, sliders, etc.)
Map controllers to audio effects (UDOs)
Represent the mapping in JSON format
Interface design
Design validation
Convert JSON mappings to a directed graph
Validate the effect chain
Generate a Csound instrument file
Instrument generation

Figure 1: Schematic of an *interface focussed instrument design* procedure. The left box (black) shows the required user actions for designing the instrument, the middle and right box (grey) show the underlying software implementation.

Figure 2: Photos taken during the workshops, while participants design their layout for the front panel, either on paper (top left) or on the enclosure, by arranging the caps of knobs.

2. MOTIVATION AND CONCEPT

During the workshops, all participants were encouraged to design the front panel of their COSMO instrument themselves (see Figure 2 and 3). We observed that they placed each controller with a specific intent for what it should control in the sound, even before they thought about any underlying Csound software instrument. So the design and layout of their controllers, to some extent, determined what the Csound instrument should do.

Based on this observation, we aim to design a software framework which would allow users to create a software instrument simply by mapping controllers to high level DSP building blocks. This integrates the interface design, sound generator design and mapping into one fluid process and thereby liberating the users from the task of DSP programming in Csound. In that way performers have a more holistic musical perspective throughout the entire instrument design process.

Figure 1 gives an overview of the three step procedure starting with the interface design by the user, followed by an automated validation process and an automated generation of the underlying software instrument. An approach we describe as *interface focussed instrument design*.

Figure 3: Example of finished designs for COSMO boxes which originated from the project.

3. IMPLEMENTATION

Based on the idea that the user will specify the controls required to shape the sound, we setup a software environment that a) provides an open and expandable library of Csound effects (Section 3.1), b) uses a common file format (JSON) to store the controller mappings (Section 3.2), and c) provides a translator tool from the mapping file to a Csound instrument (Section 3.3).

3.1 Effects Library

Csound contains hundreds of unit generators called 'opcodes', which generate or modify sound (e.g. oscillators, filters, envelopes, sample players, and more). Users can combine opcodes for more advanced signal processing by writing code in Csound language. Blocks of Csound code can be stored as 'user-defined opcodes' (UDOs [15]) and reused.

For the COSMO workshops, the participants were not required to have experience in Csound programming, so a simplified, yet flexible system for the instrument design was needed. Following a modular approach, we created a library of ready-made effects and instruments in Csound. Each effect is provided as an UDO, specifically designed for this project[2] and stored in a separate file (see Table 1). Each

[2]https://github.com/cosmoproject/cosmo-dsp

UDO file contains a header with a clear description of the *effect parameters* controllable through the *input arguments* and provides *default values* if arguments are not specified by the user.

All input values to the UDOs are expect to be linear and normalized between 0 and 1. Inside each UDO the control values are scaled to the parameter requirements of the different Csound opcodes used, as is common for control voltage in analogue modular synthesizers. This includes a conversion of the linear input values from a controller to exponential curves, if useful for the opcode parameters (e.g. filter cutoff-frequency). All scaled parameters are printed to the console for visual feedback during performance (see UDO example code listing in Appendix A).

3.2 Mapping Format

The controller inputs and their functionality are stored in a JSON file format[3]. We chose this format because it is a) human readable, b) supported by many programming languages, and c) easy to extend in the future for more complex mapping functionality.

```
{"MIDI-Patch": {
    "CC0_CH1":
        {
            "Lowpass": "Cutoff",
            "RandDelay": "Feedback"
        },
    "CC1_CH1":
        {
            "RandDelay": "Dry/wet mix"
        },
    "CC2_CH1":
        {
            "Reverb": "Dry/wet mix",
            "RandDelay": "Range"
        }
    }
}
```

Figure 4: MIDI controller mappings in JSON format for *interface focussed instrument design* using the COSMO software framework.

The example in Figure 4 shows the use case with standard MIDI-controller numbers (0-127) and MIDI-channels (1-16) abbreviated as `CCx_CHx`[4]. The patch shown in Figure 4 maps three continuous controllers to a chain of effect processing modules (UDOS from Table 1). The first controller (`CC0_CH1`) is assigned to the 'cutoff frequency' of a 'lowpass filter' (Lowpass.csd) and at the same time to the amount of 'feedback' of a 'delay effect with randomized delay times' (RandDelay.csd), a so called *one-to-many* mapping [11]. The second knob (`CC1_CH1`) controls the mix of the delay signal with the input signal, a *one-to-one* mapping. Finally, the third controller (`CC2_CH1`) adds 'reverb' (Reverb.csd) to the signal but also modifies the range of frequency changes of the random delay.

The order of appearance of the UDOs in the JSON file defines the order of the effects in the audio signal path. Figure 5 shows the resulting effects patch, defined by the mappings in Figure 4. The red arrows lay out the audio signal path, where the blue connections show mapped controllers.

[3]`http://www.json.org/`

[4]For the COSMO-Boxes different variable names must be used to read-in the control values via the Raspberry PI GPIO header.

Table 1: User-Defined Opcodes for Csound in the COSMO Effects Library.

UDOs[2]	Arguments
AnalogDelay.csd	Delay time Feedback Dry/wet mix
Blur.csd	Blur time Gain StereoMode Dry/wet mix
Chorus.csd	Feedback Dry/wet mix
Distortion.csd	Level Drive Tone Dry/wet mix
FakeGrainer.csd	Dry/wet mix
Hack.csd	Frequency Dry/wet mix
Lowpass.csd	Cutoff frequency Resonance Distortion
MultiDelay.csd	Multi tap on/off Delay time Feedback Cutoff Dry/wet mix
PitchShifter.csd	Semitones (-/+ 1 octave) Stereo mode Dry/wet mix
RandDelay.csd	Range Feedback Dry/wet mix
Repeater.csd	Range Repeat time On/off
Reverb.csd	Decay time Cutoff frequency Dry/wet mix
Reverse.csd	Reverse time Dry/wet mix
SimpleLooper.csd	Record/Play Stop/start Speed Reverse Audio Through
SineDelay.csd	Range Frequency Feedback Dry/wet mix
SolinaChorus.csd	LFO1 Frequency LFO1 Amp LFO2 Frequency LFO2 Amp Stereo mode on/off Dry/wet mix
Tremolo.csd	Frequency Depth
TriggerDelay.csd	Threshold DelayTime Min DelayTime Max Feedback Min Feedback Max Width Level Portamento time Cutoff frequency Bandwidth Dry/wet mix
Volume.csd	Level
Wobble.csd	Frequency Dry/wet mix

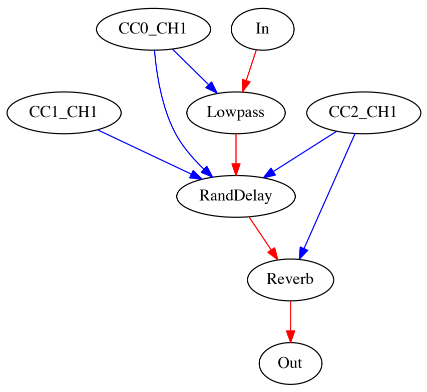

Figure 5: Graph structure generated from the mappings given in Figure 4. The graph shows the instrument's effect chain (red) and the controller mappings (blue).

3.3 Instrument design based on mappings

A mapping (.json) to Csound instrument (.csd) converter is written in the Python programming language (Version 2.7). The underlying procedure works in two main steps.

First, from the user mappings input file (see Section 3.2), a directed graph is build using the Python NetworkX package [16]. The default graph contains two basic nodes: *In* and *Out*, which are not connected in the beginning. From the JSON mapping file, the controllers (`CC0_CH1`, `CC1_CH1`, ...) and the UDOs are added as nodes to this graph, labelled as either ctrl-nodes or udo-nodes. Edges in the graph, can be audio routings between udo-nodes ('a') or control routings between ctrl-nodes and udo-nodes ('k', following the Csound variable scheme [3, p. 22]). All assignments between the nodes are taken from the JSON representation and added as edges to the graph. Edges from one udo-node to the next udo-node are made according to the order of appearance in the mapping file. The *In*-node is connected to the first udo-node, and the last udo-node is finally connected to the *Out*-node (see Figure 5). Before a Csound Instrument file (.csd) is written, the graph is validated i.e. if there is an audio signal path from *In* to *Out*.

Second, based on the graph structure, a Csound instrument file (.csd) is compiled which calls the UDOs from the Effects Library (Section 3.1) and assigns hardware device data streams to the UDO input parameters. Depending on whether it is a MIDI-Patch or a COSMO-Patch, the corresponding lines of Csound code, to read-in the hardware (ctrl7 for MIDI, chnget for COSMO) and store the values in control variables, are generated. In the case of a MIDI-controller all input values are normalized to be between 0–1 to be compatible with the UDO Library.

For each udo-node in the audio signal path, a line of Csound code is generated using the information given in the UDO file header. Earlier generated control variables or default values are assigned to the UDO input parameters corresponding to the structure of the graph. Finally, a Csound instrument definition is written into a .csd file, containing a header with Csound settings and the generated lines of Csound code (see Appendix B).

4. DISCUSSION AND FUTURE WORK

In this paper we propose an approach to instrument design from a performer's perspective and provide a proof-of-concept software framework that uses this principle to build a musical effect instrument. In contrast to traditional instrument-to-controller mapping, this approach starts at the functionality of the interface and defines the required underlying instrument. This shifts the focus from 'building a software instrument and afterwards mapping controllers to its parameters' to 'designing an interface which defines and creates the required software instrument'. *Interface focussed instrument design* therefore primarily focusses on the human-computer interaction of a software instrument. The strong link between the interface and the underlying software may also result in an interface which reflects the signal path. Although here, an example implementation is provided for the COSMO-Project [10] in combination with a library of Csound effects, this approach is not restricted to any specific hardware or software.

In the mapping literature [11] three basic types of instrument-controller mappings are discussed, respectively *one-to-one, one-to-many* and *many-to-one*. The current implementation only supports *one-to-one* and *one-to-many* mappings. Mappings of multiple controllers to the same UDO argument *many-to-one* would require adding controller-merge nodes to the graph. In the graph-to-csound code compilation, these nodes would have to result in extra lines of Csound code in the instrument definition, a possibility to explore further.

In the current implementation the main assignments are in the JSON file but some mapping details (parameter ranges, mapping curves) are handled inside the UDOs written in Csound language. Providing the UDOs this way, on one hand, it is supposed to make it easier for novice users to quickly setup their instrument. On the other hand this has limitations concerning the ability to fine-tune the instrument for optimal expressive control without knowing Csound programming. However, all code is open-source and more advanced users are encouraged to modify the existing UDOs or to design their own UDOs using customized mapping curves and parameter ranges.

However, especially in the case of *one-to-many* mappings it can be useful to quickly define control ranges already in the JSON-mapping format, a feature foreseen to be added in the future. The possible gain of flexibility can be crucial especially in situations where a performer wants to fine tweak the instrument in a rehearsal or in a soundcheck right before a concert.

A great effort towards such flexibility is provided by the Modality-toolkit [1] for Supercollider (SC). Their main motivation was an easier mapping of control data streams to SC-instrument inputs (SynthDef arguments). Modality supports many commercial controllers with different data formats (MIDI, HID, OSC), and even on-the-fly mapping and re-mapping of controllers to the SynthDefs is possible during performance. However, the Modality-toolkit is fundamentally different from our approach in terms of that a Modality-mapping does not define the instrument logic inside the SynthDef. In our case a re-mapping of the physical controls would result in a different instrument or at least in a different signal path inside the instrument, whereas in Modality only the position of the physical control is changed.

Training of movement patterns is essential for performing musicians to archive a virtuosic playing level [17]. Changing a mapping on-the-fly means that the same instrument needs to be played with different body movements from now on. Imagine a concert pianist having to remember that the pedals changed their function or more drastically that the order of tones produced by the keys has changed during the piece.

The strong binding between the interface and the software instrument, as we propose in this paper, might on one hand have advantages like an emphasis on the interface in the design process, and a link between the interface and the instrument signal path. On the other hand, this might also bring limitations in terms of less flexibility to create complex signal paths or complex mappings, different aspects that need to be explored further. As this concept was developed after the experience from the COSMO workshops mentioned in the introduction, a detailed study with users coming from different backgrounds (e.g electronic musicians, traditional musicians, programmers, non-programmers) is foreseen to better understand the applicability of this approach. Therefor a graphical-user-interface, with editor functionality for the JSON mapping files is going to be implemented.

With the current proof-of-concept implementation we aim to simplify the process of building a COSMO effect instrument in Csound by focusing only on defining controller mappings to input parameters of given UDOs. Designing instruments from the perspective of the performer's interface may open up new ways of thinking about software instrument design.

5. ACKNOWLEDGEMENTS

We are thankful to the anonymous reviewers for their valuable comments on the manuscript. This research was supported by the Arts Council Norway. The authors made extensive use of free software under GNU/ Linux.

6. REFERENCES

[1] M. Baalman, T. Bovermann, A. de Campo, and M. Negrão. Modality. In *Proceedings of the ICMC/SMC*, pages 1069–1076, Athens, Greece, 2014.

[2] F. Bevilacqua, R. Müller, and N. Schnell. Mnm: a max/msp mapping toolbox. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 85–88, Vancouver, BC, Canada, 2005.

[3] R. C. Boulanger. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. MIT press, 2000.

[4] J. Bowers, J. Richards, T. Shaw, J. Frieze, B. Freeth, S. Topley, N. Spowage, S. Jones, A. Patel, and L. Rui. One knob to rule them all: Reductionist interfaces for expansionist research. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, volume 16, pages 433–438, Brisbane, Australia, 2016. Queensland Conservatorium Griffith University.

[5] V. Chatziioannou and A. Hofmann. Physics-based analysis of articulatory player actions in single-reed woodwind instruments. *Acta Acustica united with Acustica*, 101(2):292–299, 2015.

[6] A. Cont, T. Coduys, and C. Henry. Real-time gesture mapping in pd environment using neural networks. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 39–42, Hamamatsu, Japan, 2004.

[7] W. Goebl, R. Bresin, and I. Fujinaga. Perception of touch quality in piano tones. *The Journal of the Acoustical Society of America*, 136(5):2839–2850, 2014.

[8] S. Goto and R. Powell. Netbody - "augmented body and virtual body ii" with the system, bodysuit, powered suit and second life - its introduction of an application of the system. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 48–49, Pittsburgh, PA, United States, 2009.

[9] A. Hofmann and W. Goebl. Finger forces in clarinet playing. *Frontiers in Psychology*, 7:1140, 2016.

[10] A. Hofmann, B. Waerstad, and K. Koch. Csound instruments on stage. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, volume 16, pages 291–294, Brisbane, Australia, 2016. Queensland Conservatorium Griffith University.

[11] A. Hunt and M. M. Wanderley. Mapping performer parameters to synthesis engines. *Organised sound*, 7(02):97–108, 2002.

[12] A. R. Jensenius and M. J. Lyons. Trends at nime—reflections on editing a nime reader. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, volume 16 of *2220-4806*, pages 439–443, Brisbane, Australia, 2016. Queensland Conservatorium Griffith University.

[13] C. Kiefer. Musical instrument mapping design with echo state networks. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 293–298, London, United Kingdom, 2014. Goldsmiths, University of London.

[14] V. Lazzarini. The development of computer music programming systems. *Journal of New Music Research*, 42(1):97–110, 2013.

[15] V. Lazzarini, S. Yi, J. ffitch, J. Heintz, Ø. Brandtsegg, and I. McCurdy. *User-Defined Opcodes*, pages 139–151. Springer International Publishing, Cham, 2016.

[16] D. A. Schult and P. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, volume 2008, pages 11–16, 2008.

[17] R. J. Zatorre, J. L. Chen, and V. B. Penhune. When the brain plays music: auditory–motor interactions in music perception and production. *Nature Reviews Neuroscience*, 8(7):547–558, 2007.

APPENDIX

A. EFFECTS LIBRARY EXAMPLE

To provide details of the underlying Effects library, an excerpt of Csound Code from the (Lowpass.csd) UDO containing a lowpass filter with an additional distortion effect is shown in this Section. Lines 1–15 of the Csound code below give a description of the UDO in a user friendly style. The "Arguments" and "Defaults" definitions are also relevant for the JSON-to-Csound converter, as they provide the order of input arguments and default values required when generating the Csound instrument definition (see Section 3.3).

Lines 18–41 in the code listing contain the definition of the user defined opcode for the lowpass filter. The linear input controller values are converted to an exponential curve (L. 21) for the cutoff frequency and scaled to a meaningful parameter range (L. 22, from 30 Hz to 12 kHz), before printed to the console (L. 23–24). The controller values are smoothed (L. 25) to avoid parameter jumps, that might be caused by the resolution of the controller (e.g. 128 steps for MIDI controllers). Similar processing steps done for all input parameters.

Finally, the Csound opcode for a resonant low pass filter (lpf18) is called for of the two each stereo channels (L. 37–

38) and the processed input parameters are assigned.

```
1  /********************************************
2
3          Lowpass.csd
4
5          Arguments: Cutoff frequency, Resonance,
6                     Distortion
7          Defaults:  0.8, 0.3, 0
8
9          Cutoff frequency: 30Hz - 12000Hz
10         Resonance: 0 - 0.9
11         Distortion: 0 - 0.9
12
13         Description:
14         A resonant lowpass filter with distortion
15
16 ;*********************************************
17
18 opcode Lowpass, aa, aakkk
19         ainL, ainR, kfco, kres, kdist xin
20
21         kfco expcurve kfco, 30
22         kfco scale kfco, 12000, 30
23         Srev sprintfk "LPF Cutoff: %f", kfco
24                 puts Srev, kfco
25         kfco port kfco, 0.1
26
27         kres scale kres, 0.9, 0
28         Srev sprintfk "LPF Reso: %f", kres
29                 puts Srev, kres
30         kres port kres, 0.01
31
32         kdist scale kdist, 0.9, 0
33         Srev sprintfk "LPF Dist: %f", kdist
34                 puts Srev, kdist
35         kdist port kdist, 0.01
36
37         aoutL lpf18 ainL, kfco, kres, kdist
38         aoutR lpf18 ainR, kfco, kres, kdist
39
40         xout aoutL, aoutR
41 endop
```

B. GENERATED CSOUND INSTRUMENT

In this Appendix Section the Csound instrument definition generated from the graph structure shown in Figure 5 is presented. In the code listing below, lines 8–10 import the required UDO files from the Effects Library. Then, an instrument definition is written (L. 12–27). First, two channels of audio data input stream are stored in the audio variables named aL and aR (L 13). Then, the 7-bit MIDI controller values are read-in and assigned to control variables (L. 16–18). In the lines 20–23 the earlier imported UDOs are called in the order of their appearance in the audio signal graph. Audio variables, control variables or default values are assigned to the UDO's input parameters, based on the descriptions given in the UDO file. Finally the processed audio signals are routed to the sound device output (L. 25).

```
1  <CsInstruments>
2
3  sr     = 44100
4  ksmps  = 64
5  0dbfs  = 1
6  nchnls = 2
7
8  #include "../DSP-Library/Effects/Lowpass.csd"
9  #include "../DSP-Library/Effects/RandDelay.csd"
10 #include "../DSP-Library/Effects/Reverb.csd"
11
12 instr 1
13
14  aL, aR ins
15
16  gkCC2_CH1 ctrl7 1, 2, 0, 1
17  gkCC0_CH1 ctrl7 1, 0, 0, 1
18  gkCC1_CH1 ctrl7 1, 1, 0, 1
19
20  aL, aR Lowpass aL, aR, gkCC0_CH1, 0.3, 0.0
21  aL, aR RandDelay aL, aR, gkCC2_CH1, gkCC0_CH1,
22                                      gkCC1_CH1
23  aL, aR Reverb aL, aR, 0.85, 0.5, gkCC2_CH1
24
25  outs aL, aR
26
27 endin
28
29 </CsInstruments>
```