

Csound and CsoundVST

Michael Gogins
gogins@pipeline.com

25th July 2004

Abstract

This document explains how to download, build, install, extend, and use Csound and CsoundVST on Windows and Linux.

1 Introduction

Csound is a unit-generator based, user-programmable computer music system. It was originally written by Barry Vercoe at the Massachusetts Institute of Technology in 1984 as the first C language version of this type of software. Since then Csound has received numerous contributions from researchers, programmers, and musicians around the world.

Around 1991, John fitch ported Csound to Microsoft DOS. Csound currently runs on many varieties of UNIX and Linux, Microsoft DOS and Windows, all versions of the Macintosh operating system including Mac OS X, and others.

Csound is maintained by John fitch at <http://www.sourceforge.net/projects/csound>. Documentation for the Csound language is maintained by Kevin Conder at <http://kevindumpscore.com/>. Csound's "home page" is maintained by Richard Boulanger at <http://csounds.com>.

There are newer computer music systems that have graphical patch editors (e.g. Max/MSP, PD, jMax, or Open Sound World), or that use more advanced techniques of software engineering (e.g. Nyquist or SuperCollider). But Csound still has the largest and most varied set of unit generators, is the best documented, runs on the most platforms, and is the easiest to extend. It is possible to compile Csound using double-precision arithmetic throughout for superior sound quality. In short, Csound must be considered one of the most powerful musical instruments ever created.

Csound development is ongoing, and currently stands at version 5 beta. New features in Csound 5 include the GNU Lesser General Public License, plugin unit generators, an application programming interface (API) for embedding Csound in other software, and the use of widely accepted third-party libraries for cross-platform development: libsndfile for reading and writing soundfiles, PortAudio for reading and writing digital audio from sound cards, and the Fast Light Tool Kit (FLTK) for graphics.

To create music with Csound:

1. Write an orchestra (`.orc` file) that creates instruments and signal processors by connecting unit generators (also called opcodes, in Csound-speak) using Csound's simple programming language.
2. Write a score (`.sco` file) that specifies a list of notes and other events to be rendered by the orchestra.
3. Run Csound to compile the orchestra and score, run the sorted and preprocessed score through the orchestra, and write digital audio out to a soundfile or sound card.

CsoundVST is an extended version of Csound that adds a graphical user interface, C++ and Python APIs, Python scripting, a library of Python extension modules for algorithmic composition, a VST plugin interface, and a *Mathematica* interface.

In addition to this “canonical” version of Csound and CsoundVST, there are other versions of Csound and other front ends for Csound, many of which can be found at <http://csounds.com>.

2 Downloading

Csound is hosted at <http://www.sourceforge.net/projects/csound>. Source and binary packages are available from the `files` link off that page.

The latest Csound source code is available through the Concurrent Versions System (CVS) (<http://www.cvshome.org>). To download Csound sources using CVS, run the following commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/csound login
```

```
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/csound co csound5
```

Information about accessing the CVS repository may be found in the SourceForge document “Basic Introduction to CVS and SourceForge.net (SF.net) Project CVS Services”.

If you wish to become a Csound developer, obtain a SourceForge login, and then apply to John fitch at the <http://www.sourceforge.net/projects/-csound> site.

3 Building

Csound and CsoundVST are built using the Python package `scons`, not with makefiles or GNU autotools. Experience shows that `scons` build systems are easier to write, easier to use, and run faster than autotools build systems. The only file used to build the entire Csound system is the `SConstruct` file, which is a Python script run by the `scons` shell script.

To build Csound 5:

1. Obtain the Csound source code from a SourceForge Csound 5 package file, or from SourceForge CVS.
2. Install and configure the following software packages:

- (a) Python (required) for running the build (also used for Csound-VST scripting), from <http://www.python.org>.
- (b) SCons (required) for running the build, from <http://www.scons.org>.
- (c) libsndfile (required) for reading and writing soundfiles, from <http://www.mega-nerd.com/libsndfile/>.
- (d) PortAudio for reading and writing real-time audio, from <http://www.portaudio.com/>.
- (e) FLTK version 1.1.x for displaying graphs of function tables, and for widget opcodes, from <http://www.fltk.org>.

If you also want to build CsoundVST, you must configure the FLTK libraries to enable threads (`./configure --enable-threads`). And you will need to install these additional packages:

1. The Software Interface and Wrapper Generator (SWIG) for generating Python interfaces to CsoundVST (required for CsoundVST), from <http://www.swig.org>.
2. The boost C++ template libraries for random numbers and linear algebra (required for CsoundVST), from <http://www.boost.org>.

3.1 Platforms

Currently, Csound 5 builds and runs on Windows using either the Cygwin environment (<http://www.cygwin.com>), or the MinGW (<http://www.mingw.org>) environment with the MSys shell (<http://www.mingw.org/msys.shtml>). Both of these environments are free, open source, and emulate the standard Unix/Linux environment and tools. On Linux, Csound 5 builds using the standard tools. Unix should work the same way as Linux.

On Windows, the Cygwin build procedure is more like the Linux one. However, the MinGW build is preferred, since the resulting executables do not require the Cygwin DLLs and run faster.

3.1.1 Linux

If you have properly installed all the dependencies mentioned above, you can build Csound 5 and CsoundVST simply by opening a console, changing to the `csound5` directory, and executing the `scons` command. To see the various configuration options, execute `scons -h`.

3.1.2 Windows with Cygwin

The build procedure for Cygwin is identical to Linux. However, Cygwin comes with its own customized version of Python, while CsoundVST uses the regular version of Python from <http://www.python.org>, which is built with Microsoft Visual C++. Make sure to install SCons in the Cygwin version of Python, and use that version for the build, even though CsoundVST will use the Windows version of Python.

3.1.3 Windows with MinGW and MSys

For MinGW, you may need to patch SCons as follows. Change line 51 of `SCons/Tool/mingw.py` from:

```
cmd = SCons.Util.CLVar('$SHLINK', '$SHLINKFLAGS')
```

to:

```
cmd = SCons.Util.CLVar(['$SHLINK', '$SHLINKFLAGS'])
```

It is highly recommend that you update your MinGW installation from the SourceForge site to the “current” level for core gcc, g++, binutils, utils, and the Windows API headers and libraries (w32api).

Rebuild and install a version of libsndfile no earlier than:

<http://www.mega-nerd.com/tmp/libsndfile-1.0.10pre4.tar.gz>

For PortAudio, I have found that `./configure --with-winapi=wmme` works and other `configure` options do not. If you can build PortAudio version 19 with MinGW and DirectX, ASIO, or Windows kernel streaming, please let me know how to do it.

The build procedure for MinGW is similar, but not identical, to the Cygwin procedure. The MSys shell does not allow the user to execute Python commands directly. Therefore, you need to install the *Windows* versions of Python and SCons, make sure that Python is in your Windows executable path, and run the build like this:

```
$ python c:/tools/python23/scripts/scons
```

You will also need to customize the `custom.py` file to declare to `scons` the locations of required header files and libraries, since on Windows there is no standard location for these as there is on Unix and Linux.

4 Installing

Once you have either unpacked a binary distribution, or built Csound from sources, you will need to install and configure Csound so that it will run properly on your system.

4.1 Csound

Consult the Csound language documentation for instructions on how to install and configure Csound.

On Windows, make sure that the `windows_dlls` directory is in your `PATH` environment variable, or else copy the files in it to your Windows `system32` directory.

4.2 CsoundVST

CsoundVST requires some additional configuration:

1. Make sure that the Csound executables are in your `PATH` environment variable.

2. On Windows, make sure the directory containing the `_CsoundVST.dll` file and all the Csound plugin opcodes is also in your `PATH` variable. On Unix and Linux, either install the Csound program in one of the system bin directories, typically `/usr/local/bin`, and the Csound and plugin shared libraries in one of the system lib directories, typically `/usr/local/lib`; or make sure that the directory containing the `_CsoundVST.so` file and all the Csound plugin opcodes is in your `LD_LIBRARY_PATH` environment variable. This variable may have a different name in different operating systems.
3. On all platforms, CsoundVST requires that you have Python installed on your computer. The directory containing the `_CsoundVST` shared library and the `CsoundVST.py` file must be in your `PYTHONPATH` environment variable, so that the Python runtime knows how to load these files.

5 Extending

Csound uses plugin unit generators. These are dynamic link libraries (DLLs) on Windows, and loadable modules (shared libraries that are `dlopened`) on Linux. It is relatively easy to extend Csound by writing new unit generators in C or C++.

The following assumes you already know how to make a regular Csound unit generator. If you don't, consult the Csound language documentation. Supposing that your unit generator is named `xxx`, perform the following steps:

1. Write your `xxx.c` and `xxx.h` file as you would for a regular Csound unit generator. Put these files in the `csound5/Opcodes` directory.
2. `#include "csdl.h"` in your unit generator sources. This causes the plugin development environment to emulate the regular Csound unit generator development environment.
3. Add your `OENTRY` records and unit generator registration functions at the bottom of your C file. Example (but you can have as many unit generators in one plugin as you like):

```
\#define S sizeof

static OENTRY localops[] = {
{ "xxx", S(XXX), 5, "a", "ao", (SUBR)xxxset, NULL, (SUBR)xxx}
};

/*
 * The following macro from csdl.h defines
 * the "opcode_size()" and "opcode_init()"
 * opcode registration functions for the localops table.
 */
LINKAGE
```

4. Add your plugin as a new target in the plugin opcodes section of the `SConstruct` build file:

```

pluginEnvironment.SharedLibrary('xxx',
    Split('','Opcodes/xxx.c
    Opcodes/another_file_used_by_xxx.c
    Opcodes/yet_another_file_used_by_xxx.c'))

```

5. Run the Csound 5 build in the regular way.

5.1 About OENTRY

The OENTRY structure (see `H/csoundCore.h`, `Engine/entry1.c`, and `Engine/rdorch.c`) contains the following fields:

name, dspace, thread, outarg, inargs, isub, ksub, asub, dsub

dspace There are two types of opcodes, polymorphic and non-polymorphic.

For non-polymorphic opcodes, the **dspace** flag specifies the size of the opcode structure in bytes, and arguments are always passed to the opcode at the same rate. Polymorphic opcodes can accept arguments at different rates, and those arguments are actually dispatched to other opcodes as determined by the **dspace** flag and the following naming convention:

0xffff The type of the first argument determines which unit generator function is actually called: **XXX** \Rightarrow **XXX_a**, **XXX_i**, or **XXX_k**.

0xfffe The types of the first two arguments determine which unit generator function is actually called: **XXX** \Rightarrow **XXX_aa**, **XXX_ak**, **XXX_ka**, or **XXX_kk**, as in the **oscil** unit generator.

0xfffd Refers to one argument, but does not allow **i** type, as in the **peak** unit generator.

0xfffc Similar to **0xfffe**, but deals with division by zero — thus, allows **a**, **k** and **i** type arguments.

thread Specifies the rate(s) at which the unit generator's functions are called, as follows:

thread	Description
0	i-rate <i>or</i> k-rate (B out only)
1	i-rate
2	k-rate
3	i-rate <i>and</i> k-rate
4	a-rate
5	i-rate <i>and</i> a-rate
7	i-rate <i>and</i> (k-rate <i>or</i> a-rate)

outargs Lists the return values of the unit generator functions, if any. The types allowed are:

Type	Description
i	i-rate scalar
k	k-rate scalar
a	a-rate vector
x	k-rate scalar or a-rate vector
w	w-rate spectral data type
f	f-rate streaming pvoc fsig type
m	multiple outargs (1 to 4 allowed)

inargs Lists the arguments the unit generator functions take, if any. The types allowed are:

Type	Description
i	i-rate scalar
k	k-rate scalar
a	a-rate scalar
x	k-rate scalar or a-rate vector
w	w-rate spectral data type
f	f-rate streaming pvoc fsig type
S	string
B	
l	
m	begins an indefinite list of iargs (any count)
M	begins an indefinite list of args (any count and rate)
n	begins an indefinite list of iargs (must be an odd count)
o	optional, defaulting to 0
p	optional, defaulting to 1
q	optional, defaulting to 10
v	optional, defaulting to .5
j	optional, defaulting to -1
h	optional, defaulting to 127
y	begins an indefinite list of aargs (any count)
z	begins indefinite list of kargs (any count)
Z	begins alternating kakaka... list (any count)

isub The address of the unit generator function (of type `int (*SUBR)(void *)`) that is called at i-time, or null for no function.

ksub The address of the unit generator function (of type `int (*SUBR)(void *)`) that is called at k-rate, or null for no function.

asub The address of the unit generator function (of type `int (*SUBR)(void *)`) that is called at a-rate, or null for no function.

dsub The address of the unit generator function (of type `int (*SUBR)(void *)`) that is called after performance, or null for no function.

6 Using

Assuming that you have installed and configured the software, Csound and CsoundVST can be operated in a variety of modes and configurations.

The `.csd` and `.py` files in the `examples` demonstrate a few of these modes of operation. Some of these scores are simple, others are moderately complex.

You may need to edit the `--opcode-lib` option in the Csound command in some of the `.csd` and `.py` files to match your environment. Similarly, you may need to edit the SoundFont file paths in instrument definitions that use the `fluid` SoundFont 2 player opcode to match your environment.

For real-time audio output, with or without MIDI control, you will probably want to tune the `kr` and `ksmps` orchestra statements, and the `-b` and `-B` command-line options, to give you the shortest possible latency that does not cause clicks or stutters in Csound's audio output. For example, with the Windows multimedia library build of the PortAudio library, `sr = 44100`, `kr = 441`, `ksmps = 100`, `-b100` and `-B2000` give a CD-equivalent audio sampling rate of 44,100 frames per second, a control sampling rate of 441 control samples per second with 100 audio sample frames per control sample, an audio output software buffer size of 100 sample frames, and an audio output device hardware buffer size of 2,000 sample frames, which yields an audio output latency of about 45 milliseconds — not fast enough for precise and expressive keyboard playing, but fast enough for smooth control of effects or algorithmically generated or triggered scores.

On Linux with ALSA, the audio output device should be selected using a special form of the `-odac` option, for example `-odac:plughw:0` for device 0. The `plughw` option translates Csound's audio output to the format expected by the sound card. With ALSA, latencies of a few milliseconds are possible, and expressive real-time instrumental performance should be quite feasible.

6.1 Csound

6.1.1 The `csound` Command

The original method for running Csound was as a console program. This, of course, still works. Running `csound` without any arguments prints out a list of command-line options, which are more fully explained in the Csound language documentation. Normally, the user executes something like `csound -W -omysoundfile myorchestra.orc myscore.sco` or, to use the single-file Csound structured data (`.csd`) format, `csound myscore.csd`.

Csound can read and write soundfiles (off-line rendering), read and write digital audio using a sound card (real-time rendering), read and write MIDI files, and read and write MIDI using a MIDI interface and controller (real-time control). See the Csound language documentation for more details.

6.1.2 The Csound API

The Csound API consists of the Csound library (`libcsound.a`) and the Csound header file (`csound.h`). You can build Csound into your own

software very easily using this API. For example, the Csound command itself is made this way:

```
#include "csound.h"

int main(int argc, char **argv)
{
    // Create Csound.
    void *csound = csoundCreate(0);
    // One complete performance cycle.
    int result = csoundCompile(csound, argc, argv);
    if(!result)
    {
        while(csoundPerformKsmps(csound) == 0){}
        csoundCleanup(csound);
    }
    // Destroy Csound.
    csoundDestroy(csound);
    return result;
}
```

6.2 CsoundVST

CsoundVST is a multi-function front end for Csound, based on the Csound API. CsoundVST runs as a stand-alone graphical user interface to Csound, or as a VST plugin in hosts such as the Cubase audio sequencer. CsoundVST provides both a C++ and a Python API to Csound, and to a set of classes for algorithmic composition.

CsoundVST contains a built-in Python interpreter. With Python, the user can generate a score, import a MIDI file, process notes, load and run a Csound orchestra, and in general do anything that can be done either with Csound or in Python.

6.2.1 Standalone

To run CsoundVST as a stand-alone front end to Csound, execute **CsoundVST**. When the program has loaded, you will see a graphical user interface with a row of buttons along the top. Click on the *Open...* button to load a *.csd* file. You can also click on the *Open...* button and load a *.orc* file, then click on the *Import...* button to add a *.sco* file. You can edit the Csound command, the orchestra file, or the score file in the respective tabs of the user interface. When all is satisfactory, click on the *Perform* button to run Csound. You can stop a performance at any time by clicking on the *Stop* button.

6.2.2 The CsoundVST API

CsoundVST extends the Csound API with C++. There is a C++ class for the Csound API proper, another C++ class for manipulating Csound files in code, and additional classes for algorithmic composition based on music space. All these C++ classes also have a Python interface in the

CsoundVST Python extension module. For more information, consult the Doxygen-generated files in the `csound5/doc` directory.

You can build CsoundVST into your own software using the `_CsoundVST` shared library and `CsoundVST.hpp` header file. For example, the CsoundVST stand-alone graphical user interface program is made this way:

```
#include <CsoundVST.hpp>
#include <CsoundVstFltk.hpp>

int main(int argc, char **argv)
{
    CsoundVST *csoundVST = CreateCsoundVST();
    AEffEditor *editor = csoundVST->getEditor();
    editor ->open(0);
    return 0;
}
```

There is also a high-level C API for CsoundVST, declared in `frontends/CsoundVST/csoundvst_api.h`. Any program able to interface with C calling convention functions can use this API. For example, a *Mathematica* 5.0 notebook can use the .NET/Link package's `DefineDLLFunction` to access the CsoundVST API to create an instance of CsoundVST, load an orchestra, generate a score using the power of *Mathematica*, and render that score.

6.2.3 Python scripting

You can use CsoundVST as a Python extension module. In fact, you can do this either in a standard Python interpreter, such as Python command line or the Idle Python GUI, or in CsoundVST itself in Python mode.

To use CsoundVST in a standard Python interpreter, import CsoundVST.

```
import CsoundVST
```

The CsoundVST module automatically creates an instance of CppSound named `csound`, which provides an object-oriented interface to the Csound API. In a standard Python interpreter, you can load a Csound .csd file and perform it like this:

```
C:\Documents and Settings\mkg>python
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>> import CsoundVST
>> csound.load("c:/projects/csound5/examples/trapped.csd")
1
>> csound.exportForPerformance()
1
>> csound.perform()
BEGAN CppSound::perform(5, 988ee0)...
BEGAN CppSound::compile(5, 988ee0)...
Using default language
0dBFS level = 32767.0
```

```
Csound version 5.00 beta (float samples) Jun  7 2004
libsndfile-1.0.10pre6
orchname:  temp.orc
scorename: temp.sco
orch compiler:
398 lines read
    instr  1
    instr  2
    instr  3
    instr  4
    instr  5
    instr  6
    instr  7
    instr  8
    instr  9
    instr 10
    instr 11
    instr 12
    instr 13
    instr 98
    instr 99
sorting score ...
... done
Csound version 5.00 beta (float samples) Jun  6 2004
displays suppressed
OdBFS level = 32767.0
orch now loaded
audio buffered in 16384 sample-frame blocks
SFDIR undefined.  using current directory
writing 131072-byte blks of shorts to test.wav
WAV
SECTION 1:
ENDED CppSound::compile.
ftable 1:
ftable 2:
ftable 3:
ftable 4:
ftable 5:
ftable 6:
ftable 7:
ftable 8:
ftable 9:
ftable 10:
ftable 11:
ftable 12:
ftable 13:
ftable 14:
ftable 15:
ftable 16:
ftable 17:
```

```

ftable 18:
ftable 19:
ftable 20:
ftable 21:
ftable 22:
new alloc for instr 1:
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 32.7 0.0
new alloc for instr 1:
B 1.000 .. 3.600 T 3.600 TT 3.600 M: 207.6 0.1

...

B 93.940 .. 94.418 T 98.799 TT281.799 M: 477.6 85.0
B 94.418 ..100.000 T107.172 TT290.172 M: 118.9 11.5
end of section 4          sect peak amps: 25950.8 26877.4
inactive allocs returned to freespace
end of score.             overall amps: 32204.8 31469.6
              overall samples out of range: 0 0
0 errors in performance
782 131072-byte soundblks of shorts written to test.wav WAV
Elapsed time = 13.469000 seconds.
ENDED CppSound::perform.
1
>>

```

To use CsoundVST itself as your Python interpreter, click on the CsoundVST Settings tab, and select the Python check box in the Csound performance mode box. Do not create a new CppSound object; you must use the builtin `csound` object in the CsoundVST module.

The `koch.py` script shows how to use Python to do algorithmic composition for Csound. You can use Python triple-quoted string literals to hold your Csound files right in your script, and assign them to Csound:

```

csound.setOrchestra(''sr = 44100
kr = 441
ksmps = 100
nchnls = 2
Odbfs = .1
instr 1,2,3,4,5 ; FluidSynth General MID
I; INITIALIZATION
; Channel, bank, and program determine the preset, that is, the actual sound.
ichannel = p1
iprogram = p6
ikey = p4
ivelocity = p5 + 12
ijunk6 = p6
ijunk7 = p7
; AUDIO
istatus = 144;
print iprogram, istatus, ichannel, ikey, ivelocityleft, aright
fluid "c:/projects/csound5/samples/VintageDreamsWaves-v2.sf2", \

```

```

        iprogram, istatus, ichannel, ikey, ivelocity, 1
outs_aleft, arightendin'')
csound.setCommand("csound --opcode-lib=c:/projects/csound5/fluid.dll \\
-RWdfo ./koch.wav ./temp.orc ./temp.sco")
csound.exportForPerformance()
csound.perform()

```

To run your script in Csound VST, click on the Perform button.

6.2.4 VST Plugin

The following instructions are for Cubase SX. You would follow roughly similar procedures in other hosts.

Use the *Devices* menu, *Plug-In Information* dialog, *VST Plug-Ins* tab, *Shared VST Plug-ins Folder* text field to add your **csound5** directory to Cubase's plugin path. You can have multiple directories separated by semicolons.

Quit Cubase, and start it again.

Use the *File* menu, *New Project* dialog to create a new song.

Use the *Project* menu, *Add Track* submenu, to add a new MIDI track.

Use the pencil tool to draw a *Part* on the track a few measures long. Write some music in the *Part* using the *Event* editor or the *Score* editor.

Use the *Devices* menu (or the F11 key) to open the *VST Instruments* dialog.

Click on one of the *No VST Instrument* labels, and select *_ Csound-VST* from the list that pops up.

Click on the *e* (for edit) button to open the *_ CsoundVST* dialog.

Click on the *Open* button to bring up the file selector dialog. Navigate to a directory containing a Csound csd file suitable for MIDI performance, such as `csound/CsoundVST/examples/CsoundVST.csd`. Click on the OK button to load the file. You can also open and import a suitable `.orc` and `.sco` file as described above.

Click on the *VST Instruments* dialog's on/off button to turn it on. This should compile the Csound orchestra. *Note: If you don't compile the orchestra, you won't be able to assign the plugin to a track.*

In the *Cubase Track Inspector*, click on the *out: Not Assigned* label and select *_ CsoundVST* from the list that pops up.

On the ruler at the top of the *Arrangement* window, select the loop end point and drag it to the end of your part, then click on the loop button to enable looping.

Click on the *play* button on the *Transport* bar. You should hear your music played by CsoundVST.

Try assigning your track to different channels; a different Csound instrument will perform each channel.

When you save your song, your Csound orchestra will be saved as part of the song and re-loaded when you re-load the song.

You can click on the *Orchestra* tab and edit your Csound instruments while CsoundVST is playing. To hear your changes, just click on the *CsoundVST Perform* button to recompile the orchestra.

You can assign up to 16 channels to a single CsoundVST plugin. However, you can't have more than one CsoundVST plugin in the same song!

7 Licenses

7.1 Csound and CsoundVST

Csound is ©1991-2003 by Barry Vercoe and John ffitich.

CsoundVST is ©2001-2004 by Michael Gogins.

Csound and CsoundVST are free software; you can redistribute them and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Csound and CsoundVST are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Csound and CsoundVST; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

7.2 Virtual Synthesis Technology

Virtual Synthesis Technology (VST) PlugIn interface technology by Steinberg Soft- und Hardware GmbH.

CsoundVST source code contains modified versions of source code files from the VST SDK distributed by Steinberg. *These files are to be used only for building CsoundVST.* You are *not* licensed to use these files for any other purpose. If you make a derived product based on CsoundVST or the modified VST source files herein, you *must* apply to Steinberg for your own license to use the VST SDK.

8 Contributors

Csound contains contributions from musicians, scientists, and programmers from around the world. They include (but are not limited to):

- Allan Lee
- Andres Cabrera
- Barry Vercoe
- Bill Gardner
- Bill Verplank
- Dan Ellis
- David Macintyre
- Eli Breder
- Gabriel Maldonado
- Greg Sullivan
- Hans Mikelson
- Istvan Varga

- Jean Piche
- John ffitch
- John Ramsdell
- Marc Resibois
- Mark Dolson
- Matt Ingalls
- Max Mathews
- Michael Casey
- Michael Clark
- Michael Gogins
- Mike Berry
- Paris Smaragdis
- Perry Cook
- Peter Neubacker
- Peter Nix
- Rasmus Ekman
- Richard Dobson
- Richard Karpen
- Rob Shaw
- Robin Whittle
- Sean Costello
- Steven Yi
- Tom Erbe
- Victor Lazzarini
- Ville Pulkki

9 To Do

This is a “to do” list, not necessarily complete, and in no particular order of priority or time, for Csound and CsoundVST:

1. See also the `To-fix-and-do` file in the `csound5` directory.
2. Get the ASIO version of PortAudio working on Windows, or write a kernel streaming PortAudio driver, for latency adequate to real-time performance.
3. Create a Pure Data (PD) adapter opcode that will run Pure Data (PD) patches in Csound orchestra, for both MIDI and audio input and output.
4. Create a *Mathematica* interface for CsoundVST.
5. Create a COM interface for CsoundVST.

6. Create better examples, especially to demonstrate the use of Python and of VST plugins. One example should be a live performance instrument with a Python GUI that controls instrument parameters or algorithmic composition parameters in real time.
7. Complete the work of making Csound multi-instantiable.
8. All Csound documentation in this one PDF.