

Chapter 2

Key System Concepts

Abstract This chapter provides an overview of the key principles that underline the operation of Csound: frontends; the sampling theorem; control and audio rates; processing blocks; function table generation; real-time and offline audio; the API. These ideas will be presented as a foundation for the detailed exploration of Csound programming in the subsequent chapters.

2.1 Introduction

In this chapter, we will start to introduce the reader to a number of important principles in computer music. Many of these have been hinted at in the survey of systems, languages and software packages in Chapter 1. Here, we will try to move one step deeper into these ideas. We will also focus on the issues that are central to the operation of Csound, and begin using code examples of the current language to illustrate these, even if not all of the concepts embodied in such examples were fully explained. The discussion will proceed in a mosaic-like fashion, and hopefully the full comprehension of these ideas will emerge as the following chapters tackle them more completely.

A number of key concepts are behind the workings of Csound. These extend from fields such as digital signal processing (e.g. the sampling theorem), to computation structures (sample blocks, buffers etc), and system design (frontends, API). In our discussion of these, they are not going to be organised by the areas they stem from, but by how they figure in the workings of Csound. We assume that the reader has a reasonable understanding of basic acoustics principles (sound waveforms, propagation, frequency, amplitude etc.), but more complex ideas will be introduced from the ground up.

2.2 General Principles of Operation

Csound can be started in a number of different ways. We can provide the code for its instruments, also known as the orchestra, and a numeric score containing the various sound events that will be performed by them. Alternatively, we can supply the instruments, and make Csound wait for instructions on how to play them. These can come from a variety of sources: score lines typed at the terminal; Musical Instrument Digital Interface (MIDI) commands from another program or an external device; Open Sound Control (OSC) commands from a computer network source; etc. We can also submit an orchestra that includes code to instantiate and perform its instruments. Or we can just start Csound with nothing, and send in code whenever we need to make sound.

The numeric score is the traditional way of running Csound instruments, but it is not always the most appropriate. MIDI commands, which include the means of starting (NOTE ON) and stopping (NOTE OFF) instruments might be more suitable in some performance situations. Or if we are using other computers to issue controls, OSC is probably a better solution, as it allows for a simpler way of connecting via a network. Both MIDI and OSC, as well as the numeric score, will be explored in detail later on in this book. Of course, if the user is keen to use code to control the process interactively, she can send instruments and other orchestra code directly to Csound for compilation and performance.

2.2.1 CSD Text Files

Let's examine a simple session using the basic approach of starting Csound by sending it some code. In that case, we often package these two in a single text file using the CSD format. This is made up of a series of tags (much like an XML or HTML file) that identify sections containing the various textual components. The minimum requirement for a CSD file is that it contains a section with one instrument, which can be empty (see listing 2.1).

Listing 2.1 Minimal legal CSD code

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
endin
</CsInstruments>
</CsoundSynthesizer>
```

The relevant tag for instrument (orchestra) code is `<CsInstruments>`, which is closed by `</CsInstruments>`. Everything is enclosed within the `<CsoundSynthesizer>` section, and anything outside it is ignored. In order to get sound, we need to give some substance to the instrument, and play it (listing 2.2).

Listing 2.2 Minimal sound-producing CSD code

```

<CsoundSynthesizer>
<CsInstruments>
instr 1
  out rand(1000)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>

```

In this case, what is happening is this: Csound reads the CSD file, finds the instruments, compiles it, and starts its operation. In the code, there is an instrument defined (with a noise-producing unit generator or opcode), and outside it, an instruction (`schedule`) to run it for a certain amount of time (1 second). Once that is finished, Csound continues to wait for new code, instructions to play the instrument again, etc. If nothing is provided, no sound will be output, but the system will not close.¹

2.2.2 Using the Numeric Score

Optionally, we could have started Csound with a numeric score in addition to its instruments. This is defined by the `<CsScore>` tag. The example in listing 2.3 is equivalent to the previous one, except for one difference: with the presence of the score, Csound terminates once there are no further events to perform.

Listing 2.3 Minimal sound-producing CSD code with numeric score

```

<CsoundSynthesizer>
<CsInstruments>
instr 1
  out rand(1000)
endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>

```

You can see from this example that the numeric score has a different syntax to the orchestra code. It is very simple, just a list of space-separated *parameter fields* (p-fields). Ways to use a score and keep Csound open for other external event sources will be discussed later in this book. Both score events and new instruments can be

¹ The amplitude 1,000 refers to the default setting of 32,768 for 0 dB full scale. It will produce a white noise of $\frac{1000}{32768} = 0.0305$ or -30 dB. In modern Csound coding practice, the 0 dB value is set to 1 by the statement `0dbfs = 1`, as shown in listing 2.6. More details are given in Section 2.4.2.

supplied to Csound after it has started. How this is done will depend on the way Csound is being used, and how it is being hosted.

2.2.3 Csound Options

Csound's operation is controlled by a series of options. There is a very extensive set of these, and many of them are seldom used. However, it is important to understand how they control the system. There are different ways to set the values for these options, and that also depends on how Csound is hosted. A portable way of making sure the settings are correct for the user's needs is to add them to the CSD file. This can be done under the `CsOptions` tag as shown in listing 2.4.

Listing 2.4 CSD with options controlling

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
instr 1
  out rand(1000)
endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

The example shows the setting of an option to run Csound with its output directed to the Digital-to-Analogue Converter (DAC), a generic name for the computer sound card. This is what we need to do if we want to run it in real-time. With no options, by default Csound works in an offline rendering mode, writing the output to a soundfile (named "test" by default).

Options start with a dash (-), for simple single-letter options, or with two dashes (--) for longer names. A list of all options can be found in the Csound Reference Manual, but in this book we will introduce the most relevant ones, as we come across them.

2.3 Frontends

Users interact with Csound through a program called the *frontend*, which hosts the system. There are many different types of frontends, some designed for different specific tasks, others with a more general-purpose application. The Csound 6 soft-

ware package provides a few different frontends itself, and there are also a number of third-party ones which are well maintained, and kept up to date with the system.

This is the current list of frontends maintained as part of the Csound project:

- **csound**: a general-purpose command-line interface (CLI) frontend.
- **csound6~**: a Pure Data [103] object that allows Csound to be run inside that system.
- **csound~**: an object for the MaxMSP [136] system.
- **CsLadspa**: a plug-in generator for LADSPA hosts.
- **CsoundVST**: a plug-in for VST hosts.
- **winsound**: a legacy GUI frontend originally for Windows.

Some of these are not distributed in binary form by the project, but are available as source code in the Csound repository. In terms of third-party frontends, there are many options, of which four are of note:

- **CsoundQt**: a general-purpose Integrated Development Environment (IDE) for Csound, which allows the building of user interfaces (UIs) and has facilities for scripting using the Python language.
- **Cabbage**: another IDE designed mostly for the development of plug-ins and stand-alone programs. using Csound, with full support for building UIs.
- **Blue**: a composition system for computer music that uses Csound as its sound engine.
- **WinXsound**: a GUI program with text-editing facilities, built on top of the CLI csound frontend.

2.3.1 The `csound` Command

The `csound` command, although the most basic of the frontends listed above, provides access to the full functionality of the system. It allows users to run Csound code, and to interact with it in a number of ways. It does not have a graphical interface of its own, although the Csound system includes GUI-building opcodes that can be used for that purpose. An advantage that this frontend has over all the others is that it is present wherever Csound is installed. It is useful to learn how to use it, even if it is not the typical way we interact with the system.

While it is beyond the scope of this book to discuss the operation of different frontends (for which help can be found elsewhere), we would like to provide some basic instructions on the command-line usage. For this, you will need to open up a terminal (also known as a shell, or a command line) where commands can be typed. The basic form of the `csound` command is:

```
csound [options] [CSD file]
```

The options are the same as discussed before (Sec.2.2). Typically we should supply a CSD file to Csound to get it started, unless we use a specific option to tell Csound to start empty. If that is not the case, a usage message will be printed to the terminal with a list of basic options. We can get a full list of options by using `--help`:

```
csound --help
```

Once started, Csound will behave as discussed in Sec.2.2. It can be stopped at any point by pressing the `ctrl` and the `'c'` keys together (the `ctrl-c` 'kill program' sequence).

2.3.2 Console messages

Csound provides a means of informing the user about its operation through *console* messages. These will be displayed in different ways, depending on the frontend. In the case of graphic frontends, generally there will be a separate window that will hold the message text. In the case of the `csound` command, the console is directed by default to the standard error (`stderr`), which is in most cases the terminal window. Once the system is started, the console will print basic details about the system, such as version, build date, CSD filename, etc., culminating in the start of performance, when `SECTION 1:` is displayed:

```
time resolution is 1000.000 ns
virtual_keyboard real time MIDI plug-in for Csound
0dBFS level = 32768.0
Csound version 6.07 beta (double samples) Dec 12 2015
libsndfile-1.0.25
UnifiedCSD: test.csd
...
SECTION 1:
```

Following this, as instruments start playing, we will get some information about them and their output. It is possible to write code to print custom messages to the console. We can suppress most of the console displays by adjusting the messaging level with the option `-m N`, where `N` controls how much is printed (0 means reducing messages to the minimum).

2.4 Audio Computation, the Sampling Theorem, and Quantisation

Earlier in this chapter, we introduced the idea of *signals*, in particular audio signals, without actually explaining what these are. We expect that the reader would identify this with the sound that the program is generating, but it is important to define it more precisely. Computers, through programs like Csound, process sound as a digital signal. This means that this signal has two important characteristics: it is sampled in time, and quantised in value.

Let's examine each one of these features separately. The sound that we hear can be thought of as a signal, and it has a particular characteristic: it is continuous, as far as time is concerned. This means, for instance, that it can be measured from one infinitesimal instant to another, it 'exists' continuously in time. Digital computers cannot deal with this type of signal, because that would require an infinite amount of memory. Instead, this signal is encoded in a form that can be handled, by taking samples, or measurements, regularly in time, making a discrete (i.e. discontinuous) representation of a sound recording.

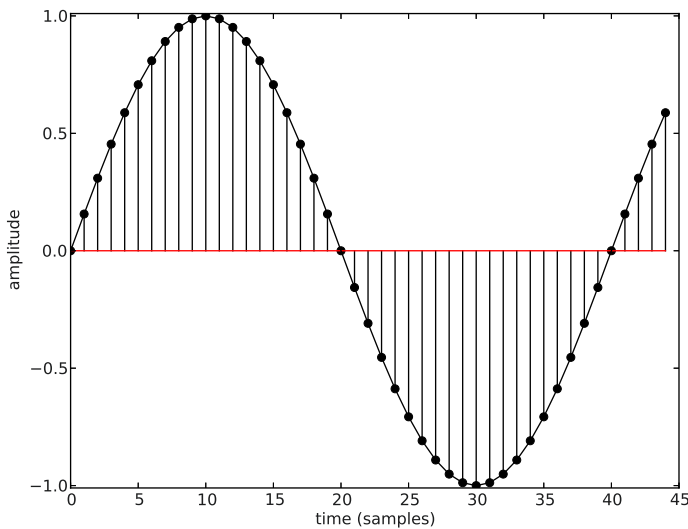


Fig. 2.1 A sampled waveform and its underlying continuous-time form. The vertical lines represent the times at which a measurement is made, and the dots represent the actual samples of the waveform

In Fig. 2.1, a sample sine wave and its underlying continuous-time form is shown. The vertical lines represent the times at which a measurement is made, and the

dots represent the actual samples of the waveform. Note that, as far as the discrete representation is concerned, the signal is not defined in the times between each sample. However, it can be reconstructed perfectly, as denoted by the continuous plot.

Equally, the sound waveform in the air can vary by infinitesimal differences as time progresses, and computers cannot, for the same reasons, deal with that. So when a measurement is made to produce a sampled representation, it needs to place that value in a finite grid of values. We call this *quantisation*. Two sampled numbers that are very close together might be represented by a single quantised output, and that will depend on how fine the grid used is. We can think of sampling and quantisation as slicing the sound waveform in two dimensions, in time, and in amplitude.

2.4.1 Aliasing

There are some key implications that arise as a result of these two characteristics of digital audio. First and foremost, we need to understand that a discrete representation is not the same thing as the continuous signal it encodes. However, we can define the conditions in which the two can be considered equivalent.

With regards to sampling in time, the first thing to recognise is that we are introducing a new quantity (or parameter) into the process, which determines how often we will be taking measurements. This is called the sampling rate (sr) or sampling frequency, and is measured in samples per second, or in Hertz (Hz, which is another way of saying ‘something per second’). Sound itself is also composed of time-varying quantities, which can also be measured in Hz. The sampling process introduces a complex interaction between the frequency of the components of a sound wave and the sr. This relationship is captured by the sampling theorem, also known as the Nyquist(-Shannon) theorem, which tells us that

In order to encode a signal containing a component with (absolute) frequency X , it is required to use a sampling rate that is at least equivalent to $2X$. [95, 116]

This places some limits in terms of the sampling rate used and the types of signals we want to use. The main implication of this is that if any component in a digital audio signal exceeds the sampling rate, it will be folded over, *aliased*, into the range of possible frequencies, which extends from 0 Hz to $\pm \frac{sr}{2}$. These aliased components can appear as a form of noise in the digital signal (if they are numerous), or unwanted/unrelated inharmonic components of an audio waveform. Within the stated limits, we can, for all practical purposes, accept that the digitally encoded signals are the same as their original form (in terms of their frequency content). In Csound, the default sampling rate is set at 44,100 Hz, but this can be modified by setting the `sr` constant at the top of the orchestra code (listing 2.5), or by using the relevant option.

Listing 2.5 Setting the sr to 48,000 Hz

```
<CsoundSynthesizer>
```



```
<CsInstruments>
sr = 48000
instr 1
  out oscili(1000,440)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>
```

How do we avoid aliasing? There are two main cases where this can be present: when we convert an original (‘analogue’) signal into its digital form (through an Analogue-to-Digital Converter, the ADC, a general name for the computer input sound card), and when we generate the sound directly in digital form. In the first case, we assume that ADC hardware will deal with the unwanted high-frequency components by eliminating them through a low-pass filter (something that can cut signals above a certain frequency), and so there is no need for any further action. In the second case, we need to make sure that the process we use to generate sounds does not create components with frequencies beyond $\frac{sr}{2}$ (which is also known as the Nyquist frequency). For instance, the code in listing 2.5 observes this principle: the `sr` is 48,000 Hz, and the instrument generates a sine wave at 440 Hz, well below the limit (we will study the details of instruments such as this one later in the book). If we observe this, our digital signal will be converted correctly by a Digital-to-Analogue Converter (DAC, represented by the computer sound card) (within its operation limits) to an analogue form. It is important to pay attention to this issue, especially in more complex synthesis algorithms, as aliasing can cause significant signal degradation.

2.4.2 Quantisation Precision

From the perspective of quantisation, we will also be modifying the original signal in the encoding process. Here, what is at stake is how precisely we will reproduce the waveform shape. If we have a coarse quantisation grid, with very few steps, the waveform will be badly represented, and we will introduce a good amount of modification into the signal. These are called quantisation errors, and they are responsible for adding noise to the signal. The finer the grid, the less noise we will introduce. However, this is also dependent on the amount of memory we have available, as finer grids will require more space per sample.

The size of each measurement in bytes determines the level of quantisation. If we have more bits available, we will have better precision, and less noise. The current standard for audio quantisation varies between 16 and 64 bits, with 24-bit encoding being very common. Internally, most Csound implementations use 64-bit floating-point (i.e. decimal-point) numbers to represent each sample (double precision), although in some platforms, 32 bits are used (single precision). Externally,

the encoding will depend on the sound card (in case the of real-time audio) or the soundfile format used.

Generally speaking quantisation size is linked to the maximum allowed absolute amplitude in a signal, but that is only relevant if the generated audio is using an integer number format. As Csound uses floating-point numbers, that is not very significant. When outputting the sound, to the sound card or to a soundfile, the correct conversions will be applied, resolving the issue.

However, for historical reasons, Csound has set its default maximum amplitude (also known as ‘0dB full scale’) to the 16-bit limit, 32768. This can be redefined by setting the `0dbfs` constant in the Csound code (or the relevant option). In listing 2.6, we see the maximum amplitude set to 1, and the instrument generating a signal whose amplitude is half scale.

Listing 2.6 Setting the `0dbfs` to 1

```
<CsoundSynthesizer>
<CsInstruments>
0dbfs=1
instr 1
  out rand(0.5)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>
```

In summary, when performing audio computation, we will be dealing with a stream of numbers that encodes an audio waveform. Each one of these numbers is also called a *sample*, and it represents one discrete measurement of an continuous (analogue) signal. Within certain well-defined limits, we can assume safely that the encoded signal is equal to its intended (‘real-world’) form, so that when it is played back, it is indistinguishable from it.

2.4.3 Audio Channels

A digital audio stream can accommodate any number of channels. In the case of multiple channels, the samples for these are generally arranged in *interleaved* form. At each sample (measurement) point, the signal will contain one value for each channel, making up a *frame*. For example, a two-channel stream will have twice the number of samples as a mono signal, but the same number of sample frames. Channels in a frame are organised in ascending order. Csound has no upper limit on the number of channels it can use, but this will be limited by the hardware in case of real-time audio. By default, Csound works in mono, but this can be changed by setting a system parameter, `nchnls`, which sets both the input and output number of channels. If a different setting is needed for input, `nchnls_i` can be used. Listing 2.7 shows how to use the `nchnls` parameter for stereo output.

Listing 2.7 Setting the nchnls to 2

```

<CsoundSynthesizer>
<CsInstruments>
nchnls=2
instr 1
  out rand(1000), oscili(1000,440)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>

```

2.5 Control Rate, ksmpls and Vectors

As discussed before in Sec. 1.2, the idea of having distinct audio and control signals is an enduring one. It is fundamental to the operation of Csound, and it has some important implications that can inform our decisions as we design our instruments. The first one of these is very straightforward: control rate signals are digital signals just like the ones carrying audio. They have the same quantisation level, but a lower sampling rate. This means that we should not use them for signals whose frequencies are bound to exceed the stated limit. In other words, they are suited to slow-varying quantities.

In Fig. 2.2, we demonstrate this by showing a 100 Hz envelope-shaped sine wave. The one-second envelope (top panel) has three stages, the shortest of which lasts for 100 ms. The waveform, on the other hand has cycles that last 10 ms each. The output signal is shown in the bottom panel. This illustrates the fact that the rates of change of audio and control signals have different timescales, and it is possible to compute the latter at a lower frequency

Internally, the control rate (kr) is also what drives the computation performed by all unit generators. It defines a fundamental cycle in Csound, called the k-cycle, whose duration is equivalent to one control period. The audio cycle determined by the sampling rate becomes a subdivision of this. In other words, at every control period, a control signal will contain one sample, and an audio signal will contain one or more samples, the number of which will be equivalent to the ratio $\frac{kr}{sr}$. This ratio has to be integral (because we cannot have half a sample), and is called ksmpls, the number of audio samples in a control period.

Another way of looking at control and audio signals is this: in the course of a computation cycle, the former is made up of a single sample, while the latter contains a block of samples. A common name given to this block is a *vector*, while the single value is often called a *scalar*. It is very important to bear this fact in mind when we start looking at how Csound is programmed. The default control rate in Csound is 4,410 Hz, but this can also be determined by the code (listing 2.8), or by a command-line option.

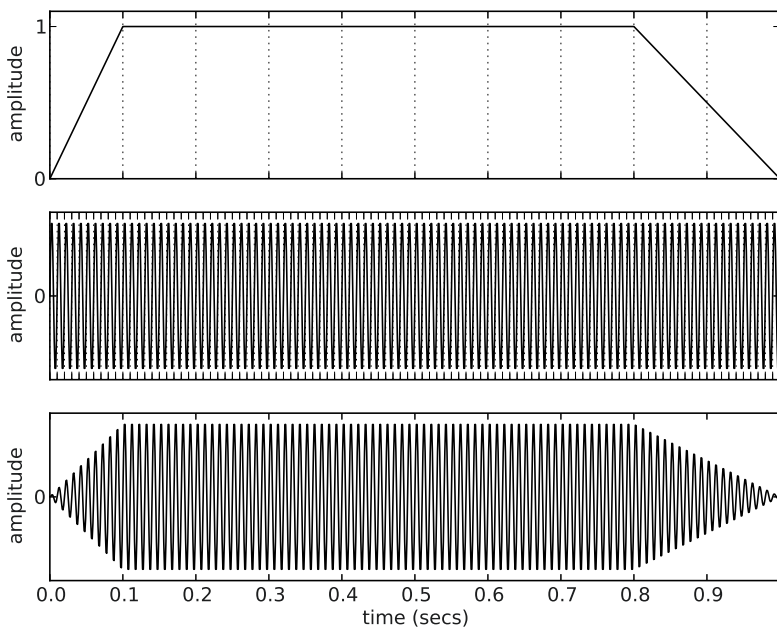


Fig. 2.2 A comparison of control and audio signal timescales. On the top plot, we have a control signal, the envelope; in the middle, an audio waveform; and at the bottom, the envelope-shaped signal. The gridlines show the difference in timescales between an audio waveform cycle and an envelope stage

Listing 2.8 Setting the *kr* to 441Hz

```
<CsoundSynthesizer>
<CsInstruments>
kr = 441
instr 1
  out oscili(1000,440)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>
```

The *kr* cannot be arbitrary: only values that ensure an integral number of *ksmps* are allowed. So often we want to set the *ksmps* directly instead, which is shown in listing 2.9, where *ksmps* = 64 makes *kr* = 689.0625 (at *sr* = 44,100). The control rate can have a fractional part, as implied by this example.

Listing 2.9 Setting ksmps to 64

```

<CsoundSynthesizer>
<CsInstruments>
ksmps = 64
instr 1
  out oscili(1000,440)
endin
schedule(1,0,1)
</CsInstruments>
</CsoundSynthesizer>

```

One important aspect is that when two control and audio signals are mixed in some operation, the former will be constant for a whole computation (ksmps) block, while the latter varies sample by sample. Depending on the $\frac{sr}{kr}$ ratio, this can lead to artefacts known as zipper noise. This is a type of aliasing in the audio signal caused by the stepping of the control signals, which are staircase-like. Zipper noise will occur, for instance, in control-rate envelopes, when ksmps is large.

In Fig. 2.3, we see an illustration of this. Two envelopes are shown with two different control rates, applied to a waveform sampled at 44,100 Hz. The topmost plot shows an envelope whose control rate is 441 Hz (ksmps=100), above its resulting waveform. Below these, we see a control signal at 44.1 Hz (ksmps=1000) and its application in the lower panel. This demonstrates the result of using a control rate that is not high enough, which is shown to introduce a visible stepping of the amplitude. This will cause an audible zipper noise in the output signal. It is important to make sure the control rate is high enough to deal with the envelope transitions properly, i.e. a short attack might require some careful consideration. Of course, envelope generators can also be run at the sampling rate if necessary.

The final implication is that, as the fundamental computation cycle is determined by the *kr*, event starting times and durations will be rounded up to an even number of these *k*-periods. If the control rate is too slow, the timing accuracy of these events can be affected. In the examples shown in listings 2.8 and 2.9, events will be rounded up to a 2.27 and 1.45 ms time grid.

2.6 Instruments, Instances, and Events

We have already outlined that one of the main structuring pieces in Csound is the *instrument*. This is a model, or recipe, for how a sound is to be processed. It contains a graph of interconnected opcodes (unit generators), which themselves embody their own model of sound generation. In order for anything to happen, we need an *instance* of an instrument. This is when the computation structures and operations defined in an instrument get loaded into the audio engine so that they can produce something. Instances are created in response to *events*, which can originate from

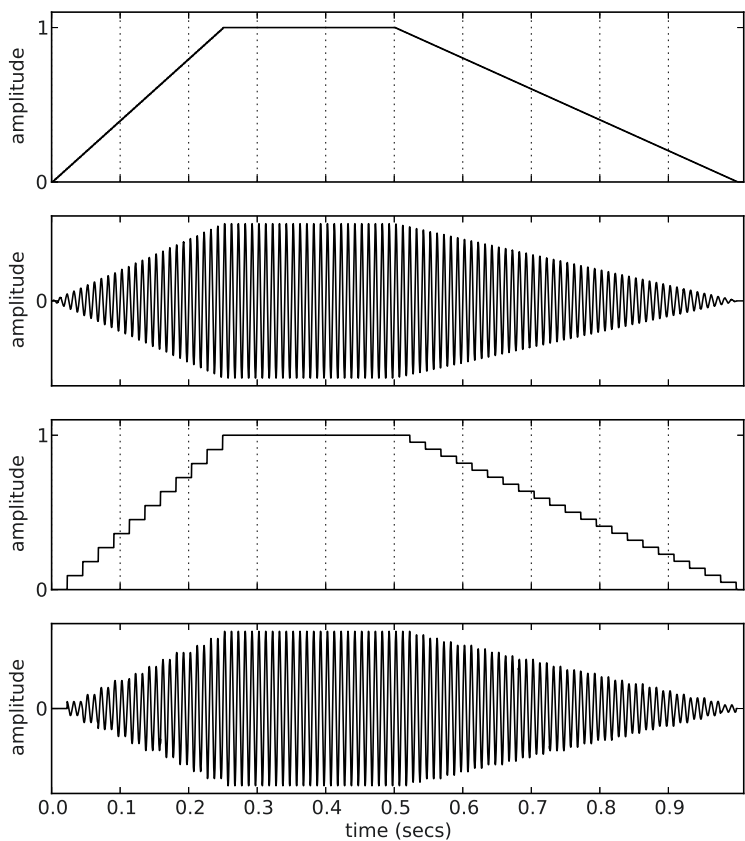


Fig. 2.3 A comparison of two different $\frac{sr}{kr}$ ratios and the resulting zipper artefacts. The illustration shows two envelopes with the resulting application of these to an audio waveform. The top example uses a $\frac{sr}{kr}$ ratio (ksmps) of 100, whereas the other control signal has its ksmps set to 1000. The zipping effect is clearly seen on the lower plot, which results into an audible broadband noise

various sources: the numeric score, MIDI commands, real-time inputs and orchestra opcodes.

2.6.1 The Life Cycle of an Instrument

An instrument passes through a number of stages in its complete life cycle: it is compiled from a text representation into a binary form, and then loaded by the engine to perform audio processing. Finally, it is de-instantiated when it stops making sound.

Compilation

Instruments start life as plain text, that contains Csound orchestra code. The next stage is the compilation. Any number of compilations can be made as Csound runs, although the first one is slightly special in that it allows for system constants to be set (e.g. `sr`, `kr` etc.), if necessary. Compilation is broken down into two steps: parsing and compilation proper. The first takes the code text, identifies every single element of it, and builds a tree containing the instrument graph and its components. Next, the compilation translates this tree into a series of data structures that represent the instrument in a binary form that can be instantiated by the engine. All compiled instruments are placed in an ordered list, sorted by instrument number.

Performance

When an event for a given instrument number is received, the audio engine searches for it in the list of compiled instruments, and if it finds the requested one, instantiates it. This stage allocates memory if no free space for the instance exists, issuing the console message

```
new alloc for instr ...,
```

with the name of the instrument allocated. This only happens if no free instrument ‘slots’ exist. If a previously run instrument has finished, it leaves its allocated space for future instances. The engine then runs the initialisation routines for all opcodes in the instrument. This is called the *init-pass*. Once this is done, the instance is added to a list of running instruments, making it perform audio computation.

Csound performance is bound to the fundamental *k-cycle* described in Section 2.5. The engine runs an internal loop (the *performance loop*) that will go into each instrument instance and call the *perform* routine of each opcode contained in it. The sequence in which this occurs is: ascending instrument number; for each instrument, instantiation time order (oldest first); and opcode order inside an instrument (line order). For this reason, opcode position inside an instrument and instrument number can both be significant when designing Csound code. The audio engine loop repeats every *k-cycle*, but instruments can also subdivide this by setting a local *ksmps*, which will trigger an inner loop operating over a shorter period for a given

instance. The life cycle of an instrument, from text to performance, is depicted in Fig. 2.4.

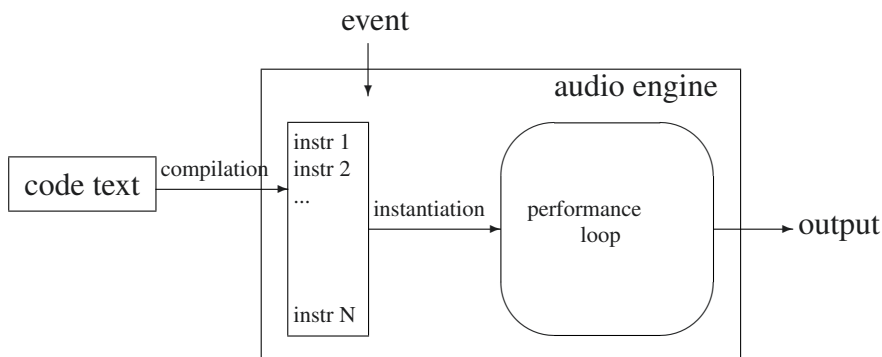


Fig. 2.4 The life cycle of an instrument: on the left, it begins as code in text form; then it is compiled and added to the list of instruments in the audio engine; once an event is received for that instrument, it gets instantiated and produces an output, running in the performance loop

De-instantiation

An instrument instance will either run until the event duration elapses, or if this duration is undefined, until a turnoff command is sent to it. In any case, the instrument can also contain code that determines a *release* or extra time period, in which case it will carry on for a little longer. Undefined duration instances can originate from MIDI NOTEON commands, in which case a corresponding NOTEOFF will trigger its turnoff, or via an event duration set to -1. In this case the instance can be turned off from an event whose instrument number is negative and corresponds to a currently running instance. On turnoff, an instance is deallocated, and for any opcodes that have them, deallocation routines are run. The memory for a given instrument is not recovered immediately, so it can be used for a future instance. Instruments exist in the engine until they are replaced by a new compilation of code using the same number or explicitly removed.²

² See opcode `remove`.

2.6.2 *Global Code*

Code that exists outside instruments is global. Only init-time operations are allowed here, and the code is executed only once, immediately after compilation. It can be used very effectively for one-off computation needs that are relevant to all instrument instances, and to schedule events. Data computed here can be accessed by instrument instances via global variables. In addition, system constants (sr, kr, ksmps, nchnls, 0dbfs) can be set here, but are only used in the first compilation (and ignored thereafter). This is because such attributes cannot be changed during performance.

2.7 Function Tables

Another key concept in Csound is the principle of *function tables*. These have existed in one form or another since the early days of computer music. They were introduced to provide support for a fundamental unit generator, the table-lookup oscillator (which will be discussed in detail in a subsequent chapter), but their use became more general as other applications were found. In a nutshell, a function table is a block of computer memory that will contain the results of a pre-calculated mathematical expression. There are no impositions on what this should be: it may be as simple as a straight-line function, or as involved as a polynomial. Tables generally hold the results of a one-dimensional operation, although in some cases multiple dimensions can also be stored.

The actual contents of a table are very much like a data array: a series of contiguous values stored in memory (arrays as data structures will be discussed in the next chapter). Each one of these values is a floating-point number, usually using double precision (64-bit), but that is dependent on the platform and version of Csound. Tables can be accessed via a variety of means, of which the simplest is direct lookup: an index indicating the position to be read is used to read a given value from the table. Fig. 2.5 illustrates this: a function table with 18 points, whose position 9 is being looked up, yielding 0.55 as a result. Note that indexing is zero-based, i.e. the first position is index 0, and the last is the table size - 1. Csound provides unit generators for direct lookup, which can be used for a variety of applications. Similarly, there are several other opcodes that use function tables as an efficient way of handling pre-defined calculations, and will employ a lookup operation internally.

2.7.1 *GEN Routines*

The data stored in tables is computed at the time they are created. This is done by invoking a GEN routine, which implements the mathematical operations needed for that. The contents of a table will depend on the routine used. For instance, the GEN function might be asked to construct one cycle of a waveform and place it in

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	GP
0.00	0.05	0.10	0.15	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	0.00

↑
 index = 9
 value = 0.55

Fig. 2.5 An example of a function table with 18 points, whose position 9 is being looked up, yielding 0.55 as a result

the table. Or we might want to define an envelope shape with a certain number of segments. We could store a sequence of pitch values to use in a loop, or to be read in random order.

2.7.2 Normalisation

Once the data is computed, we can choose to store it in raw form, i.e. the exact results, or we can normalise these prior to keeping them. Normalisation scales the function to a given range (e.g. 0 to 1, -1 to 1), which is useful for certain applications. In the Csound Reference Manual, this is called re-scaling. For instance, when storing an audio signal, such as a waveform, it is often useful to keep it normalised so that when we read it, we can apply any amplitude to it. If, however, we want to store some frequency values in Hz to use in an instrument, then it will be best to turn off re-scaling. The default for Csound, as we will see, is to normalise.

2.7.3 Precision

A table is defined by one very important parameter: its size. This can be set to anything, but as we will see later, some opcodes are designed to work with tables of specific sizes, and thus we need to be careful about this. For instance, some unit generators require tables to be set with a power-of-two size. From the perspective of signal quality, longer tables are more desirable. They will hold better precisely calculated results. On the other hand, more memory is consumed, which in most modern systems is not a significant issue.

2.7.4 *Guard Point*

Regardless of its size, every table created in Csound will include an extra point (or position) at the end, called a *guard point*, which is also illustrated in Fig. 2.5. This is to facilitate the implementation of an operation called interpolation, which provides the means for finding an in-between value when the position sought is not a whole number. This extra point can be a copy of the first position on the table (as in Fig. 2.5), or it can extend the contour calculated for the table (called an *extended guard point*). If the table is to be read iteratively (wrapping around at the ends), then we should create it with an ordinary guard point (the default). In applications using a one-shot lookup, extended guard points are needed.

2.7.5 *Table types*

There are various uses for function tables, and several types of data can be held in them. A few examples of these are:

- **Wavetables:** tables holding one or more cycles of a waveform. Typical applications are as sources for oscillators. For instance, a table might contain one cycle of a wave constructed using a Fourier Series, with a certain number of harmonics. Another example is to load into a table the samples of a soundfile, which are then available to instruments for playback.
- **Envelopes:** these are segments of curves, which can be of any shape (e.g. linear, exponential), and which can be looked up to provide control signals.
- **Polynomials:** polynomial functions are evaluated over a given interval (e.g. -1 to 1), and can be used in non-linear mapping operations.
- **Sequences:** tables can be used to store a sequence of discrete values for parameter control. For instance you might store a pattern of pitches, durations, intensities, etc, to be used in a composition.
- **Audio storage:** we can also write to tables directly from instruments, and use them to store portions of audio that we want to play back later, or to modify. This can be used, as an example, for audio ‘scratching’ instruments, or to create bespoke audio delays.

Beyond these examples, there are many other applications. Csound has over 40 different GEN routines, each one with many different uses. These will be discussed in more detail in the relevant sections of this book.

2.8 Audio Input and Output

Audio input and output (IO) is the key element in a music programming system. In Csound, it is responsible for delivering audio to instruments and collecting their

output, connecting to whichever sources and destinations are being used. These can be, for instance, a computer sound card or an audio file, depending on the options used when Csound is started. These will control whether the software is run in a real-time mode, or will work offline with soundfiles. It is also possible to combine real-time output with soundfile input, and vice versa.

In addition to the main IO, discussed in this section, there are other forms of IO that are implemented by specific opcodes. These can be used, for example, to write to a given network address or to open soundfiles for reading or writing.

2.8.1 Audio Buffers

Audio IO employs a software device called a *buffer*. This is a block of memory that is used to store computation results before they can be sent to their destination. The reason these are used is that it is more efficient to read and write chunks of data rather than single numbers. The norm is to iterate over a block of samples to produce an output, and place that result in a buffer, then write data (also in blocks) from that buffer to its destination. Similarly, for input, we accumulate a certain number of samples in a buffer, then read blocks out of it when needed. This mechanism is used regardless of the actual source or destination, but it has slightly different details depending on these.

2.8.2 The Audio IO Layers

Csound's main IO consists of a couple of layers that work together to provide access to the inner sound computation components (the audio engine). The outer level is the one used to access the actual IO device, whatever form it might take. It is composed of buffers that are used to accumulate the audio data produced by the software before reading or writing can be performed. The size of these buffers can be defined by options that are passed to Csound. Depending on the type of IO device, one or two buffers are used, a *software* buffer and a *hardware* buffer. We will discuss the particular details of these two buffers when exploring real-time and offline audio IO operations below. The relevant options to set the size of these buffers are

```
-b N
-B N
```

or in long option form

```
--iobufsamps=N
--hardwarebufsamps=N
```

In the case of the software buffer (`-b` or `--iobufsamps`), `N` refers to the number of sample *frames* (see Section 2.4.3), whereas for the hardware buffer (`-B` or `--hardwarebufsamps`), the size `N` is defined in samples.

The innermost layer is where Csound's fundamental k-cycle loop (see Sec. 2.5) operates, consuming and producing its audio samples. Here we have one buffer for each direction, input or output, called *spin* and *spout*, respectively. These buffers are ksmpls sample frames long. For input, they hold the most current audio input block, and can be accessed by the Csound input unit generators. The output buffer adds the output blocks from all active instruments. The input buffer is updated each k-cycle period, by reading ksmpls sample frames from the input software buffer. Once all instruments are processed, the spout buffer is written to the output software buffer. If used, a hardware buffer provides the final connection to the IO device used. Figure 2.6 shows the inner and outer layers of the Csound main IO system.

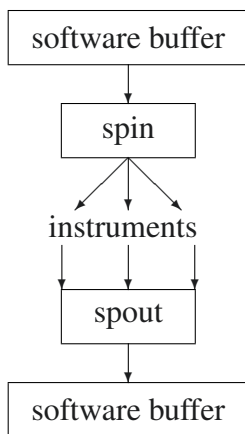


Fig. 2.6 Csound software and inner engine buffers: the audio samples are stored in the input buffer; passed on to the spin buffer at every k-cycle and accessed by instruments, which write to spout; this is then copied into the output buffer

The outer buffers can read/write to soundfiles, or to the audio device (ADC/-DAC). As already explained in Sect. 2.2, the relevant option for output is `-o`. More precisely

```
-i fnam sound input filename
-o fnam sound output filename
```

or in long format

```
--input=FNAME          Sound input filename
--output=FNAME         Sound output filename
```

Here, *filename* is taken in the UNIX sense of a logical device, which can be either a proper disk file, or in the case of real-time audio, `adc` for input or `dac` for output. Depending on the platform, it should be possible to pipe the output to

a different program as well. For file input, the data will be streamed into the input buffer until the end of file (EOF) is reached. Beyond this, no more data will be read, as there is no mechanism to loop back to the beginning of the file (to do this, there are specialised opcodes, which we will study later in this book). The `-i` option can be useful for offline batch processing of soundfiles.

2.8.3 Real-Time Audio

Real-time audio in Csound is implemented externally to the main system via a series of backend plug-ins (Fig. 2.7). These are interfaces to the various systems provided by the platforms that support real-time IO. Across the major desktop platforms, Csound provides a standard plug-in based on the cross-platform third-party library *portaudio*. In addition to this, there is also a plug-in that interfaces with the Jack IO kit, which exists in the Linux and OSX platforms. There are also OS-specific plug-ins: AuHAL and ALSA, under OSX and Linux respectively. Finally, some platforms implement exclusive audio IO solutions, which are the only option provided by the system. This the case for iOS, Android and Web-based implementations.

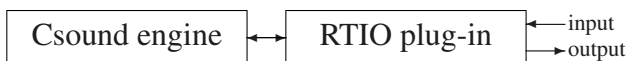


Fig. 2.7 The Csound real-time audio system based on plug-ins that provide backends across various platforms

The main consideration for real-time audio is latency. This is the time it takes for a sound to appear at the physical audio output. It can refer to the full round-trip (bidirectional) period from the sound going into the software and coming out of the speakers, or it can relate to the time from a control input (say a key pressed) until the corresponding effect happens at the output (single direction). Musicians are very sensitive to latencies, and to improve performance, we attempt to minimise it. The time lag will depend primarily on the buffer size, but it will also depend on the hardware and software platform. Smaller buffers will imply shorter latencies.

We can estimate the latency a buffer adds to the signal by dividing the number of frames in it by the sr. For instance, a buffer of 128 sample frames will have an inherent latency of 2.9 milliseconds. Only the outer buffers add latency, the spin/spout do not contribute to it (but their size will limit how small the IO buffers can be, as these cannot be less than 1 ksmps). Very low-latency buffers might cause samples to be dropped out, causing clicks in the output audio (known as ‘drop-outs’). This is because the system cannot cope with producing the audio in time to fill the output buffer. In general, this is what limits how smaller a buffer can be, and it will depend

on the platform used. Experimentation is often required to achieve low-latency audio.

In addition, drop-outs will also occur when the processing requested exceeds the computing capacity of the platform. In that case, increasing the amount of buffering will not eliminate the issues. Measures will need to be taken to reducing the computation load. These might include reducing the sr, increasing ksmps, simplifying the instrument code, reducing the number of allowed parallel instrument instances (limiting polyphony in MIDI-based performances) or a combination of these.

If present, the portaudio backend is used by default. On Linux, the `alsa` plug-in is loaded if `portaudio` is not found, otherwise a dummy IO module is employed. This does not output any audio, but runs the synthesis system under a timer, mimicking the behaviour of a sound card. For all of Csound's IO modules, the `-odac` and `-iadc` options will open the default playback and record devices. In the following sections, we will discuss the details of each one of the main backends provided by Csound.

Portaudio

The portaudio plug-in comes in two different forms, blocking and non-blocking (callback-based). The former employs a simple method to place the audio out of Csound: a subroutine is called, which writes the audio to the soundcard, blocking execution until the operation is finished. It works in a similar fashion with input audio. Both operations operate on the software buffer, which is read/written as a block to/from the system. A performance price is paid for all this simplicity: in general it is not possible to employ small buffers for low latency (drop outs resulting) with blocking operation.

The non-blocking mode of the portaudio plug-in performs much better with regards to latency. It works in asynchronous mode: the audio is written to the soundcard inside a callback routine. This is invoked by the system whenever new data needs to be sent to the output or copied from the input. In parallel to this, Csound does its processing, consuming/filling a buffer that will be filled/consumed by the callback. This buffer has the same size as Csound's software buffer, from which it gets its data, and to which it sends its samples.

Latency in both modes is dependent on the size of the software buffer (`-b` option), which is set in sample frames. The hardware buffer size (`-B`) is only used to suggest a latency (in samples) to the portaudio library. The relevant options for loading this module in Csound are

```
-+rtaudio=pa_cb  
-+rtaudio=pa_bl
```

for callback and blocking modes, respectively. Specific audio devices in this backend can be accessed as `dac0`, `dac1` etc. (for output), and `adc0`, `adc1` etc. (for input).

Jack

The Jack IO kit is an audio system that allows program inputs and outputs to be interconnected. It can also provide a low-latency route to the soundcard. Jack works as a server with any number of clients. Through the use of a dedicated plug-in, Csound can be used as a client to the system. This uses a callback system with circular buffering, whose size is set to `-B` samples. The software buffer writes `-b` samples to this. The circular buffer needs to be at least twice the size of this buffer, e.g. `-b 128 -B 256`. The software buffer also cannot be smaller than the Jack server buffer.

This module can be loaded with

```
-+rtaudio=jack
```

Different audio destinations and sources can be selected with `dac:<destination>` or `adc:<source>`, which will depend on the names of the desired Jack system devices.

ALSA

ALSA is the low-level audio system on the Linux platform. It can provide good performance with regards to latency, but it is not as flexible as Jack. It is generally single-client, which means that only one application can access a given device at a time. Csound will read/write `-b` samples from/to the soundcard at a time, and it will set the ALSA hardware buffer size to `-B` samples. The rule of thumb is to set the latter to twice the value of the former (as above, with Jack).

The alsa module is selected with the option:

```
-+rtaudio=alsa
```

The destinations and sources are selected by name with the same convention as in the jack module: `dac:<destination>` or `adc:<source>`, which will refer to the specific ALSA device names.

AuHAL

The AuHAL module provides a direct connection to OSX's CoreAudio system. It is also based on a callback system, using a circular buffer containing `-B` sample frames, with a software buffer size set to `-b` sample frames. As with Jack, the optimal configuration is to set the former to twice the value of the latter. The AuHAL backend allows very small buffer sizes, which can be used for very low latencies.

The module is loaded with

```
-+rtaudio=auhal
```

As with portaudio, specific audio devices can be accessed as `dac0`, `dac1`, etc (for output), and `adc0`, `adc1` etc. (for input).

2.8.4 Offline Audio

Audio can be computed offline by Csound. In this case, we do not have the constraints imposed by real-time operation, such as latency and computing capacity. In addition, the software will use all the processing available to generate its output as quickly as possible. This can be very useful for post-processing of recorded audio, or for composing fixed-media pieces. As noted in Chapter 1, originally all music programming systems operated purely offline, and Csound was originally designed for this type of use. As it evolved into the modern system, it kept this working model intact as its default option.

While buffering is still employed in this mode, it is less critical. The `-b` and `-B` options still refer to software and hardware (disk) buffer sizes, but its default values (1,024 and 4,096, respectively) are valid across all platforms and rarely need to be modified. The soundfile interface used by Csound is based on the third-party *libsndfile* library, which is the standard across open-source systems. Thus, Csound will be able to deal with all file formats supported by that library, which extend from the major uncompressed types (RIFF-Wave, AIFF etc.) to the ones employing compressed data (ogg, FLAC etc.).

As mentioned above, in offline mode, Csound's main software buffers will read or write to disk files. The input will draw audio from a file until it reaches the end-of-file, and will be silent thereafter. This reading begins at the same time as Csound starts processing audio and is independent from any instrument instance, because it feeds the main buffer, not a particular instance. So if an instrument uses the main inputs, it will start processing the input file at the time it is instantiated, and not necessarily at the beginning of the file (unless the instance is running from time zero). Similarly, the output is directed to a file and this is only closed when Csound finishes performance (or we run out of disk space). The relevant options for offline audio are listed in Table 2.1. These include the various possibilities for output soundfile format, and their different encoding precision settings.

2.9 Csound Utilities

The Csound software distribution also includes various utility routines that implement spectral analysis, noise reduction, soundfile amplitude scaling and mixing, and other operations. These can be accessed via the `-U` option:

```
-U <utility name> <arguments>
```

where the arguments will depend on the particular utility being used. Frontend hosts have full access to these via the API, with some of them providing graphical interfaces to these routines.

Option	Description
-i FILE, --input=FILE	input soundfile name
-o FILE, --output=FILE	output soundfile name
-8, --format=uchar	precision is set to 8-bit (unsigned)
-c, --format=schar	precision is set to 8-bit (signed)
-a, --format=alaw	a-law compressed audio format
-u, --format=ulaw	u-law compressed audio format
-s, --format=short	precision is set to 16-bit (integer)
-3, --format=24bit	precision is set to 24-bit, with formats that support it
-l, --format=long	precision is set to 32-bit (integer), with formats that support it
-f, --format=float	precision is set to single-precision (32-bit) floating point, with formats that support it
--format=double	precision is set to double-precision (64-bit) floating point, with formats that support it
-n, --nosound	no sound, bypasses writing of sound to disk
-R, --rewrite	continually rewrite the header while writing the soundfile (WAV/AIFF formats)
-K, --nopeaks	do not generate any PEAK chunks, in formats that support this
-Z, --dither--triangular, --dither--uniform	switch on dithering of audio conversion UDO from internal floating point to 32-, 16- and 8-bit formats. In the case of -Z the next digit should be a 1 (for triangular) or a 2 (for uniform)
-h, --noheader	no header in soundfile, just audio samples
-W, --wave, --format=wave	use a RIFF-WAV format soundfile
-A, --aiff, --format=aiff	use an AIFF format soundfile
-J, --ircam, --format=ircam	use an IRCAM format soundfile
--ogg	use the ogg compressed file format
--vbr-quality=X	set variable bit-rate quality for ogg files
--format=type	use one of the libsndfile-supported formats. Possibilities are: aiff, au, avr, caf, flac, htk, ircam, mat4, mat5, nis, paf, pvf, raw, sd2, sds, svx, voc, w64, wav, wavex, xi

Table 2.1 Offline audio options. These include the various possibilities for output soundfile format, and their different sample precision settings

2.10 Environment Variables

Csound takes notice of a number of environment variables. These are used to configure a system, and can be set by the user to define certain default file locations, etc. Environment variables are system-dependent, and the methods to set them will depend on the operating system. In UNIX-like systems, these can be set in shell configuration files such as `.profile`, `.bashrc` (on the bash shell) or `.cshrc` (C shell) using commands such as `export` or `setenv`. Generally these will only affect any programs run from the shell. Some OSs allow the user to set these environment vars for the whole system. GUI frontends, such as CsoundQt, will also allow these to be set inside the program as part of their configuration options.

The environment variables used by Csound are:

- `OPCODE6DIR64` and `OPCODE6DIR`: these indicate the place where Csound will look for plug-ins, in double-precision or single-precision versions of Csound. In fully installed systems, they do not need to be set, as Csound will look for plug-ins in their default installed places (system-dependent).
- `SFDIR`: soundfile directory, where Csound will look for soundfiles by default. If not set, Csound will look for soundfiles in the current directory (which can be the directory where the CSD file is located). Note that soundfiles (and other files) can also be passed to Csound with their full path, in which case `SFDIR` is not used.
- `SSDIR`: sound sample directory, where Csound will look for sound samples by default. As above, if not set, Csound will look for sound samples in the current directory (which can be the directory where the CSD file is located).
- `SADIR`: sound analysis directory, similar to the above, but for sound analysis files.
- `INCDIR`: include directory, where Csound will look for text files included in the code (with the `#include` preprocessor directive).

2.10.1 Configuration File

A file named `.csoundrc` can be used by Csound to hold default options for your system. It should reside in the user home directory (the topmost user directory), and contain any options that the user wants to keep as default, in a single line, such as:

```
-o dac -i adc -b 128 -B 512
```

These options will be used whenever Csound is run without them. If the same options are already passed to a frontend, either in the CSD or as command-line parameters, the configuration file will be ignored (the order of precedence is: parameters to program; CSD options; configuration file options).

2.11 The Csound API

The API underpins the operation of all Csound-based applications. While the main details of its operation are beyond the scope of this book, we will introduce it here in order to give the reader an insight into the lower levels of operation inside the software. An API is the public face of a programming library, which is a component of an application that provides support for specific tasks. Developers would like to use software libraries for almost all of their work, as they save the need to reinvent every step of each fundamental operation they want to use. So these components are pre-packaged bundles of functionality that can be reused whenever they are required.

Csound was originally a monolithic program, whose only interfaces were those provided by its language, and a few extra controls (MIDI and events typed at the terminal or piped from another program). It evolved into a software library with the exposure of some of its internal operations in an early form of the API. From this stage, it was then more formally developed as a library, and the functionality exposed by the API was enhanced. One key aspect of these changes is that the internals were modified to make the library *reentrant*. This allowed the Csound engine to be treated like an object, which could be instantiated multiple times. So a program using the library, such as MaxMSP or PD, or a DAW, can load up several copies of a Csound-based plug-in without them interfering with each other.

The Csound API is written in the C language, but is also available in C++ through a very thin layer. From these two basic forms, various language wrappers have been created for Java (via its Java Native Interface, JNI), Python, Lua, Closure (via JNI), and others. So programmers can access the functionality via their preferred language. In particular, scripting languages interface very well with Csound through the API, allowing composers to make use of extended possibilities for algorithmic approaches. This will be explored in later sections of this book.

2.11.1 A Simple Example

The API can be used to demonstrate the operation stages of Csound, exposing some of the internals to the reader. The presentation should be clear enough even for non-programmers. We show here a simple example in C++ that will guide us through the steps from start to completion. This code implements a command-line program that is very similar to the `csound` frontend discussed in Section 2.3. It takes input from the terminal, runs the audio engine, and closes when the performance is finished. The program code is shown in listing 2.10.

Listing 2.10 Simple Csound API example in C++

```
1 #include <csound.hpp>
2
3 int main(int argc, char** argv){
```

```

4   Csound  csound; // csound object
5   int    error;    // error code
6
7   // compile CSD and start the engine
8   error = csound.Compile(argc, argv);
9
10  // performance loop
11  while(!error)
12      error = csound.PerformKsmps();
13
14  return 0;
15  }

```

This example uses the bare minimum functionality, but demonstrates some important points. First, in a C++ program, Csound is a class, which can be instantiated many times. In line 4, we see one such object, `csound`, which represents the audio engine. This can be customised through different options etc, but in this simple program, we move straight to the next step (line 8). This takes in the command-line arguments and passes them to the engine. The parameters `argv` and `argc` contain these arguments, and how many of them there are, respectively. This allows Csound to compile a CSD code (or not, depending on the options), and once this is done, start. Note that these are other forms of the `Compile()` method, as well as other methods to send orchestras for compilation, and other ways of starting the engine. This is the most straightforward of them.

If there were no errors, the program enters the *performance loop*, which has been discussed in some detail in Section 2.6. This does all the necessary processing to produce one `ksmps` of audio, and places that into the output buffer. The method returns an error code, which is used to check whether the loop needs to continue to the next iteration. The end of performance can be triggered in a number of ways, for instance, if the end of the numeric score is reached; via a ‘close csound’ event; or with a keyboard interrupt signalling KILL (`ctrl-c`). In this case, `PerformKsmps()` returns a non-zero code, and the program closes. The C++ language takes care of all the tidying up that is required at the end, when the `csound` object is destroyed.

To complete this overview, we show how the C++ code can be translated into a scripting language, in this case, Python (listing 2.11). We can very easily recognise that it is the same program, but with a few small changes. These are mainly there to accommodate the fact that C/C++ pointers generally do not exist outside these languages. So we need an auxiliary object to hold the command-line argument list and pass it to `Compile()` in a form that can be understood. These small variations are inevitable when a C/C++ API is wrapped for other languages. Otherwise, the code corresponds almost on a line-by-line basis.

Listing 2.11 Simple Csound API example in Python

```

1  import csnd6
2  import sys
3

```

```
4 csound = csnd6.Csound()  
5 args = csnd6.CsoundArgVList()  
6 for arg in sys.argv: args.Append(arg)  
7  
8 error = csound.Compile(args.argc(), args.argv())  
9  
10 while (not error):  
11     error = csound.PerformKsmmps()
```

2.11.2 Levels of Functionality

The example above shows a very high level use of the API. Depending on the application, much lower-level access can be used. The whole system is fully configurable. The API allows new opcodes and GEN routines to be added, new backend plug-ins to be provided for audio, MIDI and utilities. We can access each sample that is computed in the spout and output buffers. It is also possible to fill the input and spin buffers with data via the API. Each aspect of Csound's functionality is exposed. The following sections provide an introduction to each one of the areas covered by the interface.

Instantiation

The fundamental operations of the API are to do with setting up and creating instances of Csound that can be used by the host. It includes functions for the initialisation of the library, creation and destruction of objects, and ancillary operations to retrieve the system version.

Attributes

It is possible to obtain information on all of the system attributes, such as sr, kr, ksmmps, nchnls and Odbfs. There are functions to set options programmatically, so the system can be fully configured. In addition, it is possible to get the current time during performance (so that, for instance, a progress bar can be updated).

Compilation and performance

The API has several options for compiling code. The simplest takes in command-line parameters and compiles a CSD ready for performance as seen in listings 2.10 and 2.11. Other functions will take in orchestra code directly, as well as a full CSD

file. At lower levels it is possible to pass to Csound a parsed tree structure, to parse code into a tree, and to evaluate a text.

Performance can be controlled at the k-cycle level (as in the examples in listings 2.10 and 2.11), at the buffer level or at the performance loop level. This means that we can process as little as one ksmps of audio, a whole buffer, or the whole performance from beginning to end, in one function call. There are also functions to start and stop the engine, to reset it and to do a clean up.

General IO

There are specific functions to set the main Csound inputs and outputs (file or device names), file format, MIDI IO devices and filenames.

Real-time audio and MIDI

The API provides support for setting up a real-time audio IO backend, for hosts that need an alternative to the plug-ins provided by the system. It is also possible to access directly the spin, spout, input and output audio buffers. Similarly, there is support for interfacing with Csound's MIDI handling system, so applications can provide their own alternatives to the MIDI device plug-ins.

Score handling

Functions for reading, sorting, and extracting numeric scores are provided. There are transport controls for offsetting and rewinding playback, as well as for checking current position.

Messages and text

It is possible to redirect any messages (warnings, performance information etc.) to other destinations. By default these go to the terminal, but the host might want to print them to a window, or to suppress them. The API also allows users to send their own text into Csound's messaging system.

Control and events

Csound has a complete software bus system that allows hosts to interface with engine objects. It is possible to send and receive control and audio data to/from specifically named channels. This is the main means of interaction between Csound and applications that embed it. In addition, it is possible to send events directly to instan-

tiate instruments in the engine. The API allows programs to kill specific instances of instruments. It can register function callbacks to be invoked on specific key presses, as well as a callback to listen for and dispatch events to Csound.

Tables

Full control of tables is provided. It is possible to set and get table data, and to copy in/out the full contents from/to arrays. The size of a function table is also accessible.

Opcodes

The API allows hosts to access a list of existing opcodes, so they can, for instance, check for the existence of a given unit generator, or printout a list of these. New opcodes can also be registered with the system, allowing it to be easily extended.

Threading and concurrency

Csound provides auxiliary functionality to support concurrent processing: thread creation, spinlocks, mutexes, barriers, circular buffers, etc. These can be used by hosts to run processing and control in parallel.

Debugger

An experimental debugger mode has been developed as part of the Csound library. The API gives access to its operation, so that breakpoints can be set and performance can be stepped and paused for variable inspection.

Miscellaneous

A number of miscellaneous functions exist in the API, mostly to provide auxiliary operations in a platform-independent way. Examples of these are library loading, timers, command running and environmental variable access. Csound also allows global variables to be created in an engine object, so that data can be passed to/from functions that have access to it. Finally, a suite of utilities that come with the system can also be manipulated via the API.

2.12 Conclusions

In this chapter, we introduced a number of key concepts that are relevant to Csound. They were organised in a mosaic form, and presented at a high level, in order to give a general overview of the system. We began by looking at how the system operates, and discussed CSD files, numeric scores and options. The principle of Csound as a library, hosted by a variety of frontends, was detailed. The `csound` command was shown to be a basic but fully functional frontend.

Following this system-wide introduction, we looked at fundamental aspects of digital audio, and how they figured in Csound. Sampling rate and quantisation were discussed, and the conditions for high-quality sound were laid out. We also discussed the concept of control rate, k-cycles and ksmips blocks, which are essential for the operation of the software.

The text also described in some detail concepts relating to instruments, instances and events. We outlined the life-cycle of an instrument from its text code form to compilation, instantiation, performance and destruction. The basic principle of function tables was explored, with a description of the typical GEN routines that Csound uses to create these.

Completing the overview of fundamental system operation, we looked at audio input and output. The two main modes of operation, real-time and offline, were introduced. The principle of audio IO plug-ins that work as the backend of the system was outlined. This was followed by the description of the common audio modules included in Csound. Offline audio and soundfile options were detailed.

The chapter closed with an overview of the Csound API. Although the details of its operation are beyond the scope of this book, we have explored the key aspects that affect the use of the system. A programming example in C++ and Python was presented to show how the high-level operation of Csound can be controlled. To conclude, the different levels of functionality of the API were discussed briefly. This chapter concludes the introductory part of this book. In the next sections, we will explore Csound more deeply, beginning with a study of its programming language.

Csound

A Sound and Music Computing System

Lazarini, V.; Yi, S.; Ffitch, J.; Heintz, J.; Brandtsegg, Ø.;

McCurdy, I.

2016, XXX, 516 p. 125 illus., 12 illus. in color.,

Hardcover

ISBN: 978-3-319-45368-2