The Cagire Book

The Cagire Book

A Forth Sequencer

Raphaël Forment

2026

The Cagire Book
First edition, 2026.
Copyright © 2026 Raphaël Forment. All rights reserved.

This book and Cagire's in-app help are built from the same Markdown sources. Edit the docs, rebuild, and the book follows. This copy was built on September 12, 2026. The text is set in New Computer Modern. Code is set in PT Mono. Typeset with Typst (typst.app). The cover shows the Cagire moutain located in Occitania.

Computers are bad.
Go touch some grass.

Contents

I Introduction 1
1 Welcome 2
1.1 About this book 3
1.2 What is live coding? 4
2 What is Cagire? 6
2.1 Two languages 7
2.2 Self-contained 7
3 Who is it for? 9
4 Where to start? 11
4.1 Reading this manual 12
5 Feedback 14
II The interface 15
1 Basics 16
1.1 First run 16
1.2 Keybindings 18
1.3 Navigation 21
1.4 Command Palette 23
1.5 Desktop & Terminal 24
1.6 Launching Cagire 25
2 Views 27
2.1 Dictionary 27
2.2 Patterns 30
2.3 Options 33
2.4 Help 38
2.5 Sequencer 40
2.6 Engine 45
2.7 Samples 47
2.8 Sampler 50
2.9 Visualizers 52
3 Using the sequencer 54
3.1 What is a step 54
3.2 The step editor 55
3.3 The sample finder 60
3.4 Snippets 62

3.5 Filling steps 63
3.6 Mirrored steps 65
3.7 Splitting and merging 66
4 Project & session 71
4.1 Session organization 71
4.2 Playing with patterns 72
4.3 Pattern properties 76
4.4 Saving and loading 77
4.5 Watching the file 78
4.6 Exporting and sharing 80
4.7 Taking notes 82
5 Sound resources 83
5.1 Loading samples 83
5.2 Recording samples 84
5.3 Editing samples 87
5.4 Loading soundfonts 90
6 Configuration 92
6.1 Audio input 92
6.2 Audio output 93
6.3 Synchronization 96
6.4 Sample folders 97
6.5 MIDI (I/O) 98
6.6 OSC (I/O) 100
III The language 103
1 Introduction 104
2 Basic concepts 106
2.1 First sounds 106
2.2 The stack 108
2.3 Words & comments 111
2.4 Values & types 113
2.5 The sound register 115
2.6 The dictionary 119
2.7 Quotations & combinators 122
2.8 The live model 124
2.9 Scope and variables 127

2.10 Error handling 129
3 Language reference 132
3.1 Overview 132
3.2 Lexical structure 133
3.3 Stack & math words 136
3.4 The sound command 144
3.5 Polyphony 146
3.6 Lists 149
3.7 Variables 152
3.8 Context 154
3.9 Control flow 158
3.10 Combinators 161
3.11 Definitions 166
3.12 Selection & cycling 170
3.13 Randomness 177
3.14 Periodicity & timing 180
3.15 Generators 185
3.16 Patterns 193
3.17 Pitch & harmony 196
3.18 Motion 207
3.19 MIDI, OSC & recording 231
4 Your own language 238
4.1 Global prelude 238
4.2 Bank prelude 239
4.3 User dictionary 240
IV The engine 243
1 Introduction 244
1.1 Inside a voice 245
1.2 Voices and orbits 245
1.3 What this part covers 246
2 Sources 247
2.1 Periodic oscillators 247
2.2 Wavetable oscillators 251
2.3 Complex oscillators 255
2.4 Vocal synthesis 257

2.5 Drum synthesis 258
2.6 Noise sources 271
2.7 Audio samples 272
2.8 Live audio 273
2.9 Soundfonts 274
3 Managing voices 277
3.1 What is a doux voice? 277
3.2 Cut groups 280
3.3 Monophonic & held voices 282
3.4 Gain and amplitude 284
3.5 Panning 285
4 Source modulation 287
4.1 Spread, detune & glide 287
4.2 Pitch modulation 289
4.3 Amplitude & Ring Modulation 291
4.4 Sub-oscillators 293
4.5 Hard-sync and soft-sync 294
4.6 Frequency Modulation (FM) 296
4.7 Phase distortion 298
5 Sample mangling 301
5.1 Selecting samples 301
5.2 Playback speed 302
5.3 Looping samples 304
5.4 Granulation 307
5.5 Scanning 310
5.6 Live recording 313
6 Envelopes 315
6.1 Voice envelope 315
7 Insert effects 319
7.1 What are insert effects? 319
7.2 Filters 320
7.3 Distortion & Degradation 331
7.4 Space & time effects 339
7.5 Frequency & pitch shifting 348
8 Orbit effects 351

8.1 What are orbits? . . . 351
8.2 Reverbs . . . 353
8.3 Delays . . . 356
8.4 Comb . . . 359
8.5 Feedback . . . 361
8.6 Compression & sidechain . . . 362
9 Output . . . 366
9.1 Multichannel output . . . 366
9.2 Master output . . . 369
9.3 Master dynamics . . . 372
9.4 Monitoring orbits . . . 376
V Audio patching . . . 381
1 A second language . . . 382
1.1 What this part covers . . . 383
2 Basics . . . 385
2.1 Inline synths . . . 385
2.2 Named patches . . . 387
2.3 The voice contract . . . 390
2.4 Inside and outside the braces . . . 394
2.5 Patches as effects . . . 398
3 Language tour . . . 404
3.1 Oscillators & noise . . . 404
3.2 Subtractive shaping . . . 407
3.3 Saturation & dynamics . . . 411
3.4 Modulation & envelopes . . . 415
3.5 Names, locals & reuse . . . 420
3.6 Channels & feedback . . . 424
3.7 Placing sound in stereo . . . 430
3.8 Effects in practice . . . 433
3.9 Putting it together . . . 437
4 Reference . . . 442
4.1 Core words . . . 442
4.2 Oscillators . . . 447
4.3 Noise & sources . . . 450
4.4 Patch filters . . . 453

4.5 Shaping, dynamics & pan 456
4.6 Delays & time 458
4.7 Envelopes & triggers 460
4.8 Buffers & sampling 463
4.9 Patch math 464
4.10 Limits 468
Appendices 473
A Is Cagire a Forth? 474
A.1 Comments 474
A.2 Unknown words 475
A.3 Programs for sound generation 475
A.4 Familiar names do different jobs 475
A.5 Conditionals come in two forms 476
A.6 There is no memory 476
A.7 There is no return stack 476
A.8 Built-ins cannot be redefined 477
A.9 Named arguments use a head block 477
A.10 So, is it a Forth? 477
B General MIDI presets 478
B.1 Preset reference 478
C Troubleshooting 487
C.1 macOS says Cagire is damaged and cannot be opened 487
C.2 Windows says it protected your PC 487
C.3 Cagire does not play any sound 488
C.4 The Engine view shows an error 488
C.5 Crackles and dropouts 488
C.6 The Load meter runs red 489
C.7 The audio device disappeared 489
C.8 My step is playing an old version 489
C.9 A sample name plays nothing 489
C.10 A patch plays nothing or will not change 490
C.11 Link synchronization errors 490
C.12 External gear receives nothing 490
C.13 A note keeps ringing on external gear 490

C.14 Live recording sounds wrong 491
C.15 My interface has more inputs than Cagire opened . 491
C.16 Nothing arrives over OSC 491
C.17 The screen looks broken 491
D About the author 493
D.1 Other projects 494
E Contributing 495
E.1 Working on the code 495
E.2 Conduct and license 496
F Acknowledgments 497
Glossary 499
Bibliography 503
Index of Concepts 505
Index of Words 507

List of Figures

Figure 1 Cagire interface: playing a pattern. 2
Figure 2 Cagire splashscreen 16
Figure 3 Cagire lays out its views in three columns around the central Sequencer, each reachable by a function key. . 21
Figure 4 The Dictionary view: the Cagire and Arf tabs on top, the category tree on the left, the chosen section's words on the right. 27
Figure 5 The Patterns view: banks in the left gutter, each bank's patterns beside it, the playing pattern marked, with the Steps strip and Properties box below. 30
Figure 6 The Options view: settings grouped under section headings, the focused row highlighted, with a box below describing it. 33
Figure 7 The Help view: chapter tabs along the top, the current chapter's sections in the sidebar, and the page beside them with a runnable example in focus. 38
Figure 8 The Sequencer view: the step grid with the cursor, the playhead trailing its fading comet, and the focused step open on its script. 40
Figure 9 The Engine view: one tab per configuration area on the left, the monitor column tracking output levels and audio status on the right. 45
Figure 10 The Samples view: folders and their files side by side, with a preview showing the selected sample's waveform, its facts, and the words that play it. 47
Figure 11 The Sampler with its Record and Edit modes; the stereo waveform fills the view and the start and end markers trim the region to keep. 50
Figure 12 The five visualizers at a glance: stereo dB meters, a triggered scope, a frequency spectrum, a mid/side Lissajous, and a sample's filled waveform. 52
Figure 13 The step editor beside the board, with the focused step's script under the text cursor and the pattern laid out as squares on the right. 55

Figure 14 The sample finder: a fuzzy search line above folder/number result rows, with the highlighted sound's waveform and facts previewed below. 60
Figure 15 Three of the fill tools over the same steps, all seeded from the cursor: e spreads evenly, r repeats, z scatters. 63
Figure 16 Step 04 is the bright source; 02 and 07 mirror it as dimmed steps marked →04, all sharing one script. . 65
Figure 17 A project nests in four layers: banks hold patterns, patterns hold steps, and a step holds the short script. 71
Figure 18 Both preludes run ahead of the steps, the project prelude first, then the bank's, so shared setup is ready before the pattern plays. 72
Figure 19 The layout Cagire expects: each sub-folder of a registered folder is a bank, its files sorted and numbered from zero. 83
Figure 20 An .sf2 dropped into a registered sample folder replaces the bank Cagire ships with. The first one found wins, and only one plays at a time. 90
Figure 21 The MIDI tab: the Nudge row over the four Output slots and the four Input rows, with a connected slot's timing and clock controls open. 98
Figure 22 The OSC tab: the Nudge row, the Mirror slot, the four endpoint slots, and the Listen row with its Last line. 100
Figure 23 Values land on the top and are taken from the top, so the last one down is the first one back. 108
Figure 24 Each column is the stack after that word runs in 5 dup + print; print shows 10 and leaves the stack unchanged. 109
Figure 25 In 5 3 -, the deeper value 5 is the left operand and 3 the right, so the line means 5 - 3 = 2. 111

Figure 26 A parameter word like gain takes a value off the stack into the register under its own name; . reads the whole register out to the engine. 116
Figure 27 Reading (2 *) holds the code as one value on the stack above the 5; apply runs it on the 5, leaving 10. 122
Figure 28 One unchanging script fired on three passes; iter advances 0, 1, 2, so index picks c4, then e4, then g4. 125
Figure 29 A four-step pattern looping three times: iter ticks once per lap and is shared, while runs counts one step's own plays. 126
Figure 30 Four reordering words as before and after, top on the right; the brick tile marks what dup, swap, over, and rot each left on top. 137
Figure 31 depth before and after, top on the right: on 1 2 3 it pushes the count 3 (the brick) and leaves the items untouched below. 138
Figure 32 sum before and after, top on the right: the list [1 2 3] collapses into its total, the brick 6. 139
Figure 33 Truth table for the six binary logic words: a and b down the left, one operator per column, filled cells for 1. 143
Figure 34 One sound command fans out into three voices; note and pan are lists (one value each), while the lone gain is shared by all three. 147
Figure 35 The three variable words compared by what they leave: !name empties the stack, @name keeps the box freq, ,name keeps a copy. 153
Figure 36 mx runs left to right and my top to bottom over the window, matching the screen; here the cursor sits at mx 0.66, my 0.36. 157
Figure 37 cycle, bounce, and walk stepping through a four-slot group as one step repeats; the filled slot on each row is the one selected. 172

Figure 38 index picks a slot of [c4 e4 g4] by number; the index wraps by modulo and a negative index counts from the end. 173
Figure 39 Which passes each gate opens across twelve loops read against iter; a filled cell is a pass where the quotation runs. 181
Figure 40 The four fixed range builders as number lines; the even ladders of .., ., and lin.. against the widening gaps of geom.. 186
Figure 41 Three hits spread as evenly as the maths allows over eight steps land on 0, 3, 6; euclidrot shifts them to 1, 4, 6. 188
Figure 42 div slices the step into evenly spaced offsets; swing keeps the on-beats and pushes every second offset late, ghost ticks marking the straight positions. 192
Figure 43 Three pattern strings laid onto the same four steps, showing how hits, rests, held notes, and bracketed groups map to slots. 194
Figure 44 Intervals as semitone distances from a root over one octave of keys, from $C_4 = 60$ to $C_5 = 72$. 197
Figure 45 C major laid out as degrees across the octave, where deg picks a degree by index and mode rotates which degree begins the scale. 201
Figure 46 A chord quality stacks its intervals on the root: maj7 (0 4 7 11) on c4 fans into keys 60 64 67 71. 203
Figure 47 edo cuts the 1200-cent octave into equal steps; tuning instead places steps at cents you choose. 206
Figure 48 The six curves a travel can take from start to end, from linear to the eight-step stair; slews use the same shapes. 208
Figure 49 The three random motions: jit jumps to a fresh value each period, drunk wanders to a neighbour, a drift oscillation smooths the jumps into a glide. 210

Figure 50 The ADSR shape the ead family sweeps a parameter through: rise over attack, fall to sustain over decay, hold, then return over release. 211
Figure 51 The four osc shapes over two cycles: sine, triangle, saw, and square. 222
Figure 52 One rising ramp under three curve amounts: slow start above 0, straight at 0, fast start below 0. . . . 223
Figure 53 m. tests the flags in a fixed order and sends the first that matches, from control change down to a plain note. 232
Figure 54 o. turns the register into one OSC message: the address becomes the path, and every other parameter becomes a named argument carrying its typed value. 235
Figure 55 One voice's fixed processing order, from source through filters and effects out to its orbit; gain drives the chain, postgain trims the result. 245
Figure 56 The four basic waveforms: sine, triangle, sawtooth, and square. 247
Figure 57 At 0.25 the wave is high a quarter of the cycle, a square at 0.5, high three-quarters at 0.75. 249
Figure 58 `vosc` lays the four shapes along the `wave` axis: sine at 0, triangle at 0.33, sawtooth at 0.67, square at 1. . 250
Figure 59 A wavetable is a row of single-cycle waveforms; scan parks between two adjacent cycles and crossfades between them. 251
Figure 60 Every frame of a wavetable is a single cycle, and a cycle can take almost any shape. 252
Figure 61 The three macros and a slot name go in, drumkit picks which of the two kits renders that slot, and both kits fill all nine slots at matched levels. 261
Figure 62 Spectral slope sets the three colours apart: white is flat, pink rolls off at about −3 dB per octave, brown at about −6 dB. 271

Figure 63 Audio input enters through live (also mic) and runs the full voice chain to the speakers; in the same room it loops back into the mic and howls. 273
Figure 64 One .sf2 file opens into a whole bank of sources, one gm word per instrument, with the drum kit on its own bank. 274
Figure 65 A voice runs its source through its own processing and effects, then feeds an orbit. 277
Figure 66 Above, one sound command fans out into one voice per value: the longest list sets the count and shorter lists wrap to fill it. Below, two hats share a cut group and the second one chokes the first. 281
Figure 67 The three placement controls: pan sets left-right position, width scales the stereo image, haas widens with a short inter-channel delay. 285
Figure 68 spread fans one voice into seven copies: the original at centre, six detuned in symmetric pairs that pan wider as they detune further. 287
Figure 69 Vibrato oscillates the pitch around the note; vib sets the rate, vibmod sets the depth. 289
Figure 70 Driving pitch from a generator: a slide glides between two pitches, while an lfo on note sweeps it up and down. 290
Figure 71 The same modulator against zero: amplitude modulation stays between zero and one, ring modulation crosses into negative and flips the voice's polarity. 291
Figure 72 The sub-oscillator runs an octave below the main tone, one sub cycle spanning two of the main, mixed in underneath to add weight. 293
Figure 73 A saw synced to a slower master: hard sync snaps the wave back at each master cycle, soft sync reverses its direction instead. 295

Figure 74 The three routing controls: fmpivot blends between cascade and parallel, fmfb feeds the top operator back into itself, fmloop rings op 1 into op 2. 297
Figure 75 The phase ramp bent three ways by warp, mirror, and size, each bent ramp (left) beside the waveform it produces (right). 299
Figure 76 Cagire numbers the hats bank from 0; 1 n selects open.wav, and n wraps an index past the end back to the start. 301
Figure 77 Two ways to address a sample: begin and end window an arbitrary fraction, while slice cuts it into equal numbered parts and pick selects which one plays. . 306
Figure 78 The six stages of the DAHDSR envelope: delay, attack, hold, decay, sustain, and release. Only attack, decay, and release are curved. 315
Figure 79 Frequency response of the three filter shapes, with the resonance peak sitting right at the cutoff. 321
Figure 80 The three-band EQ lifts or dips each region in decibels; the inset shows tilt pivoting the whole spectrum from dark (-1) to bright (+1). 326
Figure 81 The same sine through each lo-fi effect: crush stair-steps the wave vertically in amplitude, coarse stair-steps it in time. 331
Figure 82 The six saturator curves dmode selects, each an input-to-output transfer against the no-distortion diagonal, with a dasym bias clipping one waveform unevenly. 334
Figure 83 Once the driven signal crosses the threshold it folds back on itself, and each reflection adds bright partials. 337
Figure 84 The three curves the signal reflects against: triangle keeps hard corners, sine rounds them off, wrap snaps back like a saw. 338
Figure 85 An oscillator sweeps the short delay time against the dry voice; chorusdepth sets how far it swings. . . . 340

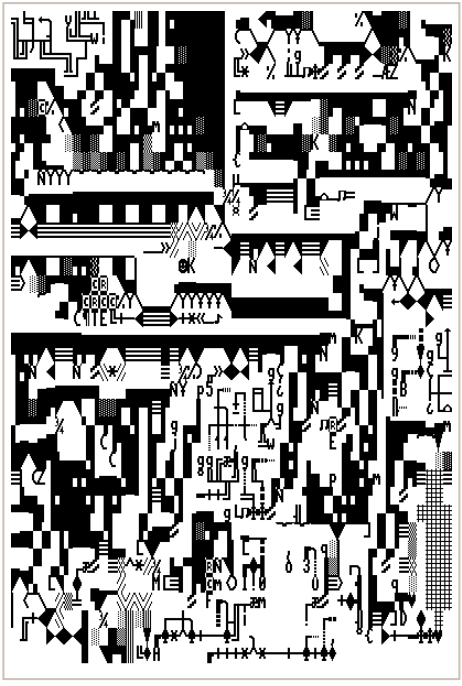
Figure 86 A short delay feeds back on itself and sweeps against the dry voice, carving an evenly spaced comb of notches that slide across the spectrum. 342
Figure 87 A phaser mixes the dry voice against a chain of allpass stages, sweeping a few unevenly spaced notches across the spectrum. 344
Figure 88 The flanger's comb carves many evenly spaced notches that slide together; the phaser's allpass stages carve four uneven ones that widen toward the top of the spectrum. 344
Figure 89 A transient enters a cascade of twelve allpass filters and spreads into a soft wash; smearfb feeds the chain back for resonance. 347
Figure 90 Frequency shifting adds a fixed number of Hz to every partial; pitch shifting multiplies each by a common ratio. 349
Figure 91 Many voices feed one orbit, which runs a fixed effect chain and passes the shared result on. 351
Figure 92 A reverb impulse over time: dry input, a predelay gap, early reflections, then a diffuse tail whose length verbdecay sets. 354
Figure 93 The four delaytype flavours share one feedback line and differ only in routing: standard, pingpong, tape, and multitap. 358
Figure 94 A noise burst enters a short delay tuned by combfreq, then feeds back through the damping one-pole and feedback multiplier before summing with the input. 359
Figure 95 A top-down ring of output pairs; superpan sets the voice's azimuth and superwidth lights the neighbouring pairs with a cosine taper. 367
Figure 96 Every output channel ends in the master bus, a three-stage chain: a DC blocker, a linked peak limiter, then a tanh soft clip. 368

Figure 97 Cagire runs the script as sequencing language, but captures everything between the braces as one synth definition that becomes the voice's source. 382
Figure 98 Cagire derives a patch's identity from its text. Unchanged text reuses the same instrument, any real edit compiles a fresh one. 385
Figure 99 `synth` and `defsynth` install into one patch registry; `synth` names the patch from its text, `defsynth` from a name you choose. 388
Figure 100 The engine feeds `notefreq`, `gate`, and `vel` in. The pattern drives declared `params`, and the voice wraps out with its envelope, gain, and orbit effects. . . . 391
Figure 101 The braces choose the dictionary: outside them lpf sets a filter param on the voice, inside it is a filter ugen in the graph. 394
Figure 102 Where an effect patch lives: `dsp` closes the orbit chain and works on the whole bus, `fx` sits inside one voice before its envelope. Both replace what they read. . 399
Figure 103 The three source groups: periodic sources carry pitch, noises carry none, and a few make control signals rather than tone. 404
Figure 104 A saw enters with a full stack of harmonics; the low-pass, high-pass, and band-pass each carve a different slice out of it. 408
Figure 105 One modulator, two regimes: heard as motion below roughly twenty cycles a second, fused into sidebands and timbre above it. 415
Figure 106 The gate signal becomes a shape: adsr traces its held attack, decay, sustain, and release; ar fires once on the rising edge. 418
Figure 107 Three ways to reuse a signal: a colon recipe pastes a fresh copy, a tilde value runs once as one shared node, and as name binds a live node mid-chain. 420

Figure 108 A bracket list builds parallel channels for a two-channel out; 2 ! replicates a mono node into a shared pair, and nth keeps one channel. 424
Figure 109 as e binds the live signal and e prime reads it one sample late, so the loop can feed itself without a cycle. 427
Figure 110 Raphaël Forment 493

Part I

Introduction



Chapter 1

Welcome

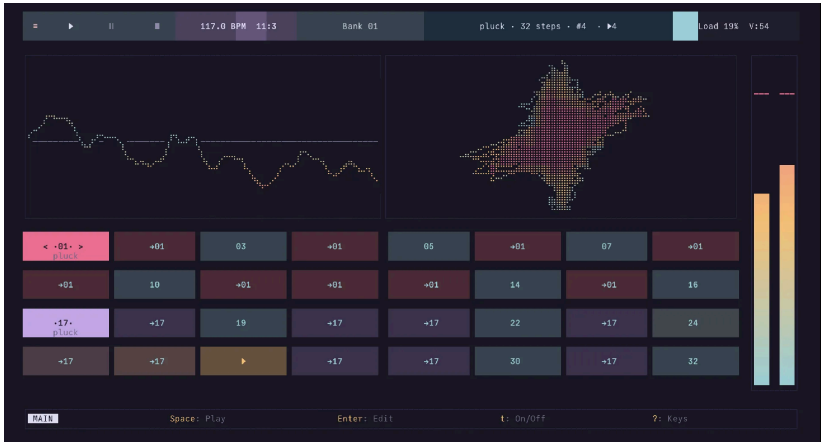


Figure 1: Cagire interface: playing a pattern.

Cagire is a music creation software that can be described as an electronic music instrument built for improvisation and live experimentation. It sits in a weird position in the realm of music software by being both a live coding environment, a step sequencer and an algorithmic composition toolbox. It has far more in common with a strange little instrument than with a Digital Audio Workstation. Cagire is the instrument that I always wanted to play with during Algoraves (*algorithmic rave parties*). Algoraves are events where you can experience a non-stop stream of niche experimental, generative and algorithmic music. Live coders often play techno and pulse-driven music during these events. Actually, every type of music can be heard as well during Algoraves as long as making it involves the use of computers, code and people to have fun. Cagire is meant to be small, fun and seriously powerful.

The software name come from the name of a mountain. The *Pic de Cagire* is a summit of the Pyrenees. I live under it. It is far from

being the highest peak of the chain but it stands ahead of it, alone, and everyone around here knows its silhouette. It felt natural to give its name to this instrument.

Cagire is built to stand on its own. It can do everything you need to improvise and perform live electronic music. It can synthesize sounds on-the-fly, play, record and mangle audio samples, send and receive MIDI and OSC to drive software or hardware instruments. You can use it as a straightforward sequencer or as an experimental music studio. Cagire can be used as the brain of a live setup. It can also hide in a corner and become a simple instrument used in a larger music ensemble. While designing it, I've put a strong emphasis on making everything as efficient and direct as possible. Once you start playing, it is designed to let you stay in the flow. Cagire is centered around writing domain-specific code to get the musical result you want. There are two small programming languages to do so: one made for event-based music, the other to describe signals and audio processing. These languages are made so that you can improvise music on-the-fly. You can create complex musical events, structures and textures with just a few keystrokes. As you go down the rabbit-hole, you will learn to master both languages.

1.1 About this book

This book is both a tutorial and a reference. It is available in two different formats. You can read it from the application itself, in the `Help` view or as a `.pdf` file that you can also print to get a nice book! If you are reading this text from inside the application: every code example is interactive. You can run and edit any example to explore them live. Try to run this example by pressing `n` followed by `Enter`. If you want to edit it, press `e` to toggle the editor. Change `c4` to `c5` and run it again to hear the same note one octave higher:

```
sine snd c4 note .
```

This line is a good introduction to Cagire as a language. It names a `sine` source, plays it, and tunes it to middle C. Change `c4` to `e4`, or `sine` to `saw`, and run it again to hear the difference. Cagire is

all about writing, listening, and repeating that loop over and over again until you are satisfied, tired or bored. This book is going to get as deep as possible in order to teach you everything there is to know about the system. There are keys to make navigation in the in-app documentation faster:

Key	What it does
`n/p`	Move to the next or previous example
`Enter`	Evaluate the focused example
`e`	Edit the focused example
`Ctrl+R`	Run your edited version
`/`	Search the whole manual

The edits you can make to an example stay local. They will reset when you leave the page. Be careful about that!

1.2 What is live coding?

Live coding is the practice of writing and changing code in real time, often in front of an audience, to shape music as it unfolds. It is improvisation and thinking through code. It can be technical, poetic, weird, preferably all at once. It is an autotelic activity: the act of doing it is its own reward. You can read more about it on websites such as toplap.org and livecoding.fr. People started writing books about it more than one decade ago. Live coding is playing the computer as an instrument with code as a conversational partner. Moreover, live coding is not something that you tend to define too much but rather something that you experience and share. Nothing about live coding is new. People were improvising with code back in the 1980s: The Hub passed music between networked computers, and artists at STEIM in Amsterdam rewired their software while it kept running. The practice got its name in a Hamburg bar in 2004, where a handful of musicians founded TOPLAP. The dancing came later when Alex McLean and Nick Collins coined the term *Algorave*. One central idea runs through all of it as a motto: *show us your screens*. The TOPLAP manifesto calls *obscurantism dangerous*.

Live coders project their screens and let the room watch the music get built, edited, and broken in real time. Mistakes, glitches and errors are also an essential part of the show. The distance between an idea and the sound it makes shrinks to a few keystrokes. Live coding is also all about sharing code in open source and helping each other. Nowadays, live coding is also a political and ethical statement. It is about being human beings making do-it-yourself music with computers and friends. In a sense, live coding is not that much about music. It is often a way to rethink electronic music and the musical / technological / political world we grew up with.

Chapter 2

What is Cagire?

Cagire is a step sequencer. What sets it apart is what a step holds: a script. When the sequencer reaches a step in a pattern, it runs that script. The script can do whatever it is written to do: define a synthesizer, play a note, trigger a sample, add effects, draw on randomness, or several of these at once. One single pattern of 32 steps can hold an entire universe of music. Because anything can be programmed, one step can contain a lot of surprises! You can keep playing one pattern forever or layer multiple ones. It is up to you to decide how to use the instrument and how to leverage what it offers you.

From this step sequencer template, Cagire is designed to let you explore new directions and workflows. You can use the sequencer to play beat-driven music or bypass it completely to make generative and algorithmic music. You can also try to test the software limits, going on the experimental side of things. Cagire's languages are designed so that each step can become a rabbit hole with no end in sight. You can be effective with a few words at first, and spend weeks discovering all the other words and capabilities offered by the sound engine. Here is a short script to get your feet wet:

```
saw snd
< c4 eb3 g3 > note
[ .5 .8 ] choose gain
( 8 crush ) sometimes
0.5 decay
.
```

Read each line right to left, one word at a time. `saw snd` starts a sawtooth source. `< c4 eb3 g3 >` cycles between these notes to make an arpeggio. `[ .5 .8 ] choose gain` is choosing two different gain values at random. `( 8 crush ) sometimes` will crush the sound

50% of the time. `0.5 decay` shapes how quickly each note will fade to silence. The final `.` is a word whose role is to output the result of each computation. This is what Cagire is offering in miniature. You define sounds, set their parameters, then play them. It only gets more creative or complex from this point on. Everything else in Cagire, apart from the programming language, works like an ordinary sequencer. You toggle steps on and off, copy and paste them, and rearrange them freely. Steps are grouped into patterns, patterns into banks, and more than one pattern can play at the same time. None of that departs from what a drum machine already does.

2.1 Two languages

The language you just saw is the sequencing language. It decides when events fire and with which settings, using sounds the engine already knows. There is a second language hiding inside it: Arf. Arf is a tiny synthesis language that builds the instrument itself, sample by sample. It lives between curly braces, and the braces can appear anywhere in a script:

```
{
  110 saw
  800 lpf
  out
} synth .
```

You can play for months without ever opening a brace. The built-in sources and effects go a long way. When you want to design your own oscillators, filters and effects, the *Audio patching* (V) part of this manual teaches Arf from scratch. Think of it as your reward to explore after you have mastered the rest of the ecosystem.

2.2 Self-contained

Cagire contains everything you need to make music. It has been conceived to be usable even if you don't have access to other software or resources. It carries a robust and versatile audio engine, Doux,

built into the program. You can perform with nothing but Cagire and your favorite audio samples. It is possible to reach outward when you want it. Cagire speaks MIDI and OSC in both directions, driving your synthesizers and other software while listening back over the same ports. You can also output your sound over multiple channels if you would like to export stems or post-process your sound. It slots into a larger rig as easily as it stands alone.

The software is small and requires no installation other than dropping the binary where your OS expects it to be. You can really think of it as a blob; a free standing computer program that you can download and run easily on Windows / Linux and MacOS altogether. Cagire also tries to be runnable on computers with relatively limited specs. You should be able to run it on a Raspberry Pi or on your dusty Linux home computer. No internet connection will be required to do anything and all projects are sharable through text format, which is the most reliable format there is on this planet.

Chapter 3

Who is it for?

I built Cagire to be an instrument that feels personal while being useful to others. There is no such thing as a neutral tool or software. In Cagire, you are likely to find out how I think about music improvisation and electronic music. Software is a great way to convey ideas, thoughts, theories and world views. Every tool leans somewhere, and the one you reach for every day quietly shapes the music you end up making. Cagire leans hard: spend time with it and you will end up improvising and playing with samples and synthesis a lot. Here are some ideas infused into Cagire that you will have a hard time escaping from:

Cagire is an instrument, not a DAW: There is no timeline to fill, no mixdown to render, no finished file waiting at the end. You are not assembling a track to play back later. You are playing, now, and the performance is the music. If you came looking for a place to record, arrange, and export, Cagire is the wrong tool. If you want something to pick up and play, it is the right one.

Playing means being hands on: You shape a phrase by typing code. You push it somewhere new by modifying that code. You fly on the keyboard. Nothing is fixed until you stop moving. Like any instrument, it gives back what you put in. The more you play, the more it gives. Projects are a convenience if you do not want to start from scratch or if you want to share and store your ideas.

Cagire is free and open source: It will stay that way. You are free to edit Cagire. The code and software belongs to you. I will not gate features behind a paywall or refuse that you have a voice of your own. You can become part of the development team if you want. Cagire is built for people who need it and should be accessible to whoever needs it. If you like Cagire, try sharing with

others too. Send your projects, samples, code snippets, share the music and the software with other folks!

Chapter 4

Where to start?

The quickest way to begin with Cagire is to make your first sound. Press `F5` or use `Ctrl+arrows` to go back to the Sequencer, Cagire's main view. Move the cursor onto the step you'd like to edit and press `Enter` to open its editor. Make sure that the transport is playing beforehand. The two buttons at the top left of the screen tell you which it is: the one that is lit is the state you are in, and the keys that drive them, `F9` and `F10`, are printed underneath. `F9` starts and stops from any view. You can also tell by eye, because a playing pattern has animated steps and things generally seem to be in motion. You are ready to start playing and you can copy this example program:

```
saw snd
5000 500 2 slide lpf
c4 eb4 bb4 note
2 decay
1 verb
.
```

Let's translate this program. `saw snd`: starts a sound source, a sawtooth oscillator running at the default frequency. `5000 500 2 slide lpf` adds a lowpass filter whose cutoff slides from 5000 Hertz down to 500 Hz. `c4 eb4 bb4 note`: plays these three pitches together as a chord. `2 decay`: gives each note a fade after 2 steps. `1 verb` is turning the reverb on for that orbit (audio bus). `.` is the word used to order the sound engine to play the result. You do not need to understand every word you are using just yet. Cagire blends many concerns in one language: music theory, sound synthesis, acoustics, engine parameters, etc. The keybinding described below can be used inside the editor. Make yourself at ease:

Key	What it does
`Enter`	Opens the script editor on the selected step
`Ctrl+R`	Runs the current script once (preview)
`Ctrl+E`	Evaluates the script without closing the editor
`Esc`	Evaluates the script and closes the editor
`Space`	Starts or stops the pattern, and the sequencer with it if it is halted
`F9`	Starts or stops the sequencer, from any view

Cagire can feel a bit weird at first because scripts are mixing a lot of different concerns in one place. When you describe a sound, you have to think a bit about audio synthesis, a bit about notes and theory, a bit about managing the audio engine and session, etc. You can also include notes and comments. Once you get more comfortable with writing Cagire code and hear your first results, the shape of the language starts to explain itself. It allows you to think and describe various aspects of sound and music using a simple and terse language.

4.1 Reading this manual

You do not need to read this manual in order. This is not a page-turner and it has not been written to be read in a linear fashion. The manual is arranged in six parts. Each part will focus on an important aspect of Cagire:

- **Introduction**: What Cagire is and how to get going.
- **Interface**: views, sequencer grid, projects, and settings.
- **Language**: stack, words, logic, sequencing and building scripts.
- **Engine**: sources, effect chains, modulations and audio output.
- **Audio patching**: building your own synthesizers and effects.
- **Appendices**: Reference tables and background.

If none of the interface makes sense yet, start with *The Interface*. If you have never written in a stack language, start with *The language* (III). If it already makes sense to you, open the Dict view with `F1` to browse every word, and press `?` on any view to see its keys. Everything is laid out to be learned once and then played

without thinking about it: muscle memory, interface layout, quick navigation, simple language. Have fun!

Chapter 5

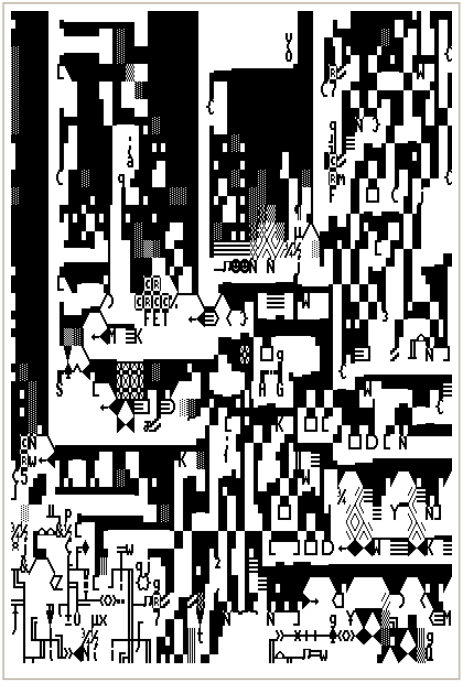
Feedback

I enjoy receiving feedback about the software and the documentation and I read every message that I receive. Cagire is a one-person project, written and maintained in the gaps between coding, playing and the hardships of life. The things I overlook tend to be exactly the things a fresh pair of eyes will catch in the first five minutes. If you notice that something is missing, incorrect or downright wrong while reading, I want to hear about it. Bug reports about the audio engine are welcome too, especially when you can tell me what you did, what you expected, and what happened instead.

The easiest way to reach me is by email at raphael.forment@gmail.com. You can also report things on the project repository at https://git.raphaelforment.fr/BuboBubo/Cagire. This documentation is written as a plain set of markdown files that live alongside the source code. It means you do not have to wait for me to fix the things you find. If you are comfortable with it, you can edit the relevant file directly and send a pull request, however small, whether it is a typo, a clumsy sentence, a missing step or a whole section that you think deserves to exist. I will happily take corrections to my English, which is not my first language, as well as suggestions about ordering, tone and the examples themselves. Maintaining all of this and keeping it current as the software keeps moving is genuinely difficult, so every correction, every question and every pull request makes the next reader's experience a little better, and that help is always very welcome.

Part II

The interface



Chapter 1

Basics

1.1 First run



Figure 2: Cagire splashscreen

Every time you start Cagire, you are welcomed by the splash screen. Any key at all dismisses it. The first time you use the software, you will also see small guides popping up on each view. These popups will teach you what you are looking at and can be easily dismissed. There is no setup wizard and no tour to sit through. You only get the splash screen, the popups and a demo song ready to play. If you would rather be walked through it, this page is that walkthrough: how to take your first steps, and how to bring the help back or turn it off once you know your way around.

Cagire opens on an animated splash. You will see an animated mountain landscape featuring the name and the version number. Pay attention to the version number, it can be quite useful to know what version of Cagire you are running. Press any key or click anywhere and the splash screen goes away. You will always end up

in the Sequencer after that. The splashscreen hits once per launch, and that is the last you see of it until the next one.

1.1.1 A demo to land in

Rather than open onto an empty grid, Cagire loads one of its built-in demo songs. These songs all have something didactic about them. There is sound the moment you press `Space`. Cagire remembers which demo it opened last time and is cycling through them, a different one each launch. Poke at a working project, read its steps, and hear how it is put together before building your own. The demo is an ordinary song like any other; nothing you do to it is precious, and nothing is kept unless you save it yourself. When you would rather land on a blank grid, open the Options view and turn off *Demo on startup*. Cagire opens empty from then on. If you build something you are proud of, you are welcome to contribute a demo of your own. Just send it to me!

1.1.2 Guides as you go

Most views greet you with a short guide the first time you enter them. These guides are just a few sentences on what the view is for and the handful of keys that matter most. A couple of views have two pages of it. The guides are per view, so you meet each one as you arrive rather than all at once. Dismiss a guide with `Enter` and it is gone for good. Close it with any other key and it will greet you again the next time you come back to that view.

Key	What it does
←/→	Move between the guide's pages
`?` / `F1`	Open the full Help on this view
`Enter`	Dismiss this view's guide for good
any other key	Close the guide for now; it returns next time you enter the view

1.1.3 Bringing the guides back

Cagire remembers which guides you have dismissed from one session to the next, in the same settings file as everything else. See *Launching Cagire* (1.6). To see them all again, open the Options

view and choose *Reset guides*: every guide returns as if it were your first run. It is the quickest way to retrace the tour, or to hand a friend a fresh start.

1.2 Keybindings

Cagire has many keybindings, and they are organized in different layers. A small set of core keybindings work everywhere, no matter where you are in the application. They allow you to move between views, to start and stop the playback and to open the command palette. Inside each view, the rest of the keyboard takes on that view's own meaning. When a dialog is open, it also takes over the keyboard until you close it. You never have to hold all of this knowledge at once. Press `?` in any view and Cagire lays out the full key list for wherever you are right now. It is normal to forget some keybindings and to discover some new ones after a while. What follows is the keys that hold across the whole program. The most important keybindings are the one you use to move between views:

Key	What it does
`F1-F8`	Jump straight to a view
`Ctrl+`←/→/↑/↓	Step to the neighbouring view on the grid

Following closely are the keybindings used to start and stop the transport:

Key	What it does
`F9`	Start or stop playback
`F10`	Stop and rewind
`Space` (in the Sequencer and Patterns)	Start or stop the pattern you are pointing at
`Space` (in Dict, Options, Help, and Engine)	Start or stop playback
`p` (in Patterns)	Start or stop playback
`P` (in Patterns)	Stop playback and rewind

`F9` and `F10` drive the sequencer itself, and they reach it from every view without exception. Everything else in that table is a convenience that happens to be free where it sits.

`Space` is the one to think about, because it does not mean the same thing everywhere. In the Sequencer and in Patterns it launches the clip you are pointing at, which is usually what you want from those two views. If the sequencer is halted when you press it, Cagire starts the sequencer along with the clip, so the press is never silent. Elsewhere `Space` has nothing better to do and it starts and stops playback, except in the Sampler, where it auditions a take, and in Samples, where it is left alone.

Starting and stopping are the whole of it. There is no held state to resume from. A stop always rewinds: patterns stay armed but return to their first step, the loop counters go back to zero, and the variables are cleared, so the next start is always a fresh run from the top of the bar.

There are some other keybindings that are essential to feel at ease:

Key	What it does
`:`	Open the command palette
`?`	Show this view's full key list
`s`	Save the song to a `.cagire` file
`l`	Open the Load hub of demos, tutorials, and your own songs
`q`	Quit; Cagire asks before it goes

These four hold everywhere the keyboard is not being typed into. Inside the step editor, the notes, a search box, or a dialog, the letters go to the text and you leave first with `Esc`. Saving and loading are covered in *Saving and loading* (4.4).

The step editor is the one place where the letters are all spoken for, since they belong to the text. Rather than hunt for chords, Cagire hands the keys back to the sequencer with `Esc` and leaves the editor on screen while it does. Nothing on the page moves, and every sequencer key works again at once. `Enter` returns them to the text, and a second `Esc` closes the editor.

Key	Where it takes you
`Enter`	Into the text
`Esc`	Out to the board, editor still on screen
`Esc` again	Editor closed, sequencer full page

Because that trip costs one key and repaints nothing, the editor borrows none of yours. `Tab` indents and takes suggestions, `?` types a question mark, the digits type digits.

`Alt+Shift` and the arrows mean the same thing here as on the sequencer: walk the bank and pattern grid.

Key	What it does
`Alt+Shift+`←/→	Previous / next pattern
`Alt+Shift+`↑/↓	Previous / next bank, on the pattern you were already on

Patterns lie across the row and banks stack down the column, the way the Patterns view arranges them, and a bank move keeps its pattern. Steps are plain arrows on the board, one `Esc` away. `F9` and `F10` reach the transport from inside the editor as they do everywhere else. *The step editor* (3.2) covers the rest.

Some keybindings will work only in certain contexts like dialogs but also in the command palette:

Key	What it does
`Esc`	Close or cancel the dialog in front of you
`Enter`	Confirm, or run the highlighted entry
↑/↓	Move the selection
`y` / `n`	Answer a yes/no prompt; ←/→ switch between them

The mouse can also be used for some but not all operations. You can generally click tiles and various areas of the interface. Most often, it will perform an action that you could have triggered using the keyboard or the options. However, don't try to use the mouse for everything. Cagire works in a terminal emulator after all, and mouse support is quite limited!

1.3 Navigation

Cagire's interface is laid out as a set of views or panels. Each one of them is a full screen dedicated to a single kind of work: building a pattern, launching clips, browsing words, recording a sample, reading this manual. Cagire gives each task its own space and lets you cross between them quickly. You are always in exactly one view. Learning your way around Cagire is mostly learning how to move between these views and not to be lost. Take your time to learn how to move efficiently.

The views or panels sit on a small grid which is three columns wide. The Sequencer is in the middle position. It is the central view and everything is organized around it. The top row holds the Dict, Patterns, and Options views. The middle row holds Help, the Sequencer, and the Engine. A wider bottom row holds the Samples and Sampler views. A faint minimap grid flashes up whenever you switch between views. This minimap is showing where you are and where the next step will land you. It fades away after a while so it never sits in your way. Moving left and right on the minimap will wrap around each row. One more step past the edge brings you back to the other side; up and down stop at the top and bottom.

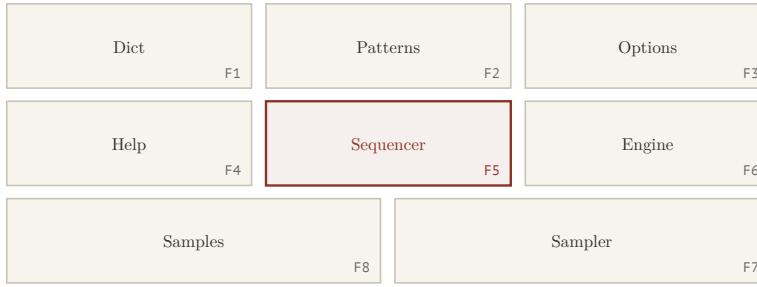


Figure 3: Cagire lays out its views in three columns around the central Sequencer, each reachable by a function key.

Every view/panel can also be accessed through a function key of its own. It is the faster way to reach that view but also the most

cumbersome thing to remember if you really want to be fast. Here are the two ways to move between views in Cagire:

- `Function` (`F1..F12`) keys: jump straight to a view.
- `Ctrl+←/→/↑/↓`: move across the minimap, one view at a time.

The mouse can also help you to navigate between views. If you click on the bottom left corner of the screen, where the name of the view/panel sits, you can also open the minimap and click your way around to reach your destination. The same thing goes for the upper-left corner of the screen where a suspicious hamburger menu icon sits. Click it to open the minimap.

Key	What it does
`F1–F8`	Jump straight to a view: Dict, Patterns, Options, Help, Sequencer, Engine, Sampler, Samples
`Ctrl+←/→/↑/↓`	Step to the neighbouring view on the grid
`F9`	Start or stop playback, from any view
`F10`	Stop and rewind, from any view
`Space`	Start or stop playback; in the Sequencer and Patterns views, the pattern you are pointing at. See *Keybindings* (1.2) for where it does something else
`:`	Open the command palette

With the mouse:

Mouse	What it does
Click a minimap tile	Jump straight to that view
Click the ≡ button, or the footer badge	Pin the minimap so it stays up
Click the tempo in the header	Set the tempo
Click the pattern in the header	Jump to the Patterns view
Click the stats in the header	Jump to the Engine view
Click a transport icon	Play or stop, the same as `F9` and `F10` printed under them

Press `?` in any view for its full key list.

1.4 Command Palette

Most of what Cagire can do has a key of its own. You do not have to remember all of them. Press `:` from any view and the command palette opens. It is one place where every action lives, reachable by name. It is the fastest way to run something you use rarely, and the surest way to find something you have forgotten. The palette is a small box with a `>` prompt waiting for you to type. Start typing a few letters of what you want and Cagire matches them against each command's name, what it does, and its shortcut. You are likely to find at least a rough guess. Open the command palette cold and the commands sit under headings by category. Once you type, they collapse into a single list with the closest match first. Each row carries the command's name on the left and its own keybinding. The palette doubles as a way to learn the shortcuts. Move through the results and run what you land on:

Key	What it does
`↑`/`↓`	Move the selection
`Ctrl+↑`/`↓`	Move the selection by five
`PageUp` / `PageDown`	Move the selection by ten
`Enter`	Run the highlighted command, then close the palette
`Esc`	Close without running anything

What is inside is grouped the way you work, under the same headings you see when you open the palette cold:

Group	What's inside
Project	Starting fresh, saving, loading, and notes
Playback	Starting sound, nudging the tempo, stopping and rewinding
Navigation	Jumping to any view
Editing	Toggling, filling, and reshaping steps

Group	What's inside
Pattern	A pattern's length, speed, copying, and sharing
Bank	Copying, resetting, and sharing whole banks
Clipboard	Copying, pasting, and linking steps
Mute/Solo	The mix
Audio	Restarting the engine, hushing, panicking, a test tone
Display	The Options view's many toggles
Prelude	Your startup code
Help	Searching this manual or the dictionary

There are around eighty commands in all and the number of commands is likely to grow over time. Some commands have no shortcut of their own. The palette is sometimes the only way to reach them. The palette has one trick beyond names. On the Sequencer and the Patterns view, type a number instead of a word and `Enter` jumps straight to that step or pattern.

1.5 Desktop & Terminal

Cagire comes in two versions. These versions share everything, they are essentially the same software. The main difference is how you run them and where they run. One runs inside a terminal (`cagire`), the other in its own dedicated window (`cagire-desktop`). This manual applies to either one without change. Choosing between them is a matter of how you like to launch and look at the program:

- `cagire`: terminal version. It runs inside a terminal emulator. It uses the terminal's own font and size. This version fits naturally if you already work in a terminal. One advantage of it is that it uses whatever theme and font configuration you have already setup.
- `cagire-desktop` : The desktop version. It opens a standalone window and draws the same interface itself. You start it the way you start any other application, by opening the file or clicking

its icon. Right-click anywhere in the window to open a graphical configuration menu.

Beyond the window itself, *the two versions are interchangeable*. The keybindings are identical. If you live in a terminal already, the terminal version slots into what you have. If you would rather not think about terminal setup, the desktop version is the one to reach for.

1.6 Launching Cagire

Most days you start Cagire and nothing more. You type `cagire` in a terminal or open `cagire-desktop` like any other application. Cagire loads your settings, greets you with the splash screen and drops you into a demo. The command line can do a little more, though, and both versions take the same flags. Flags are there when you know exactly how you want Cagire to be configured before opening it. You can make an alias that opens a particular set before even seeing the interface:

```
cagire --project ~/sets/tonight.cagire
```

Started this way, Cagire skips the demo and opens the song directly. The patterns that were playing when the song was saved are queued again, ready to sound on the first bar. If the file cannot be opened, Cagire prints why and stops rather than dropping you into an empty session. Since a song remembers the sample folders it was built from, the one flag usually brings its sounds along with it.

1.6.1 The flags

Flag	Short	What it does
`--project <file>`	`-p`	Open this `.cagire` song at launch, instead of the demo
`--samples <folder>`	`-s`	Use this sample folder for the session; repeat it for several
`--output <device>`	`-o`	The output audio device, by name or by its number in the device list

Flag	Short	What it does
`--input <device>`	`-i`	The input audio device, in the same forms
`--channels <n>`	`-c`	How many output channels to open
`--buffer <n>`	`-b`	The audio buffer size, in samples
`--help`	`-h`	Print the flags and exit
`--version`	`-V`	Print the version and exit

Each flag is an override for that one launch. Cagire remembers your device, buffer size, and sample folders between sessions, and a flag steps in front of the remembered value without you having to change anything in the Options or the Engine view. `--samples` replaces the registered folders for the session rather than adding to them, which is exactly what you want when a set travels with its own sounds. Device names with spaces go in quotes: `cagire -o "Scarlett 2i2 USB"`.

1.6.2 Where Cagire keeps things

Your settings, everything you change in the Options view and the Engine tabs, live in a single file: `~/.config/cagire/default-config.toml` on macOS and Linux, in the `cagire` folder under `%APPDATA%` on Windows. You never have to edit it, but knowing where it is has its uses: copy it to carry your setup to another machine, or delete it to give Cagire a factory reset. Songs are not kept there. A song is a plain `.cagire` file that you save wherever you like and pass around freely; see *Saving and loading* (4.4).

Chapter 2

Views

2.1 Dictionary

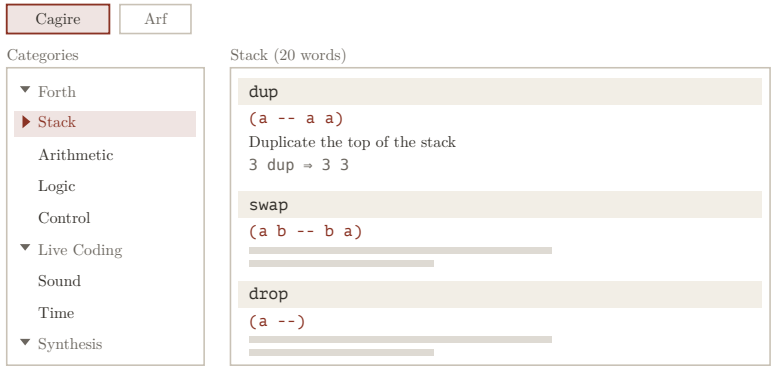


Figure 4: The Dictionary view: the Cagire and Arf tabs on top, the category tree on the left, the chosen section's words on the right.

The Dictionary is Cagire's glossary of words. It is the fastest way to learn what the language can do. It lists every built-in word Cagire knows, grouped into sections. You keep this view a key away and look things up as you play. Press `F1` to open it. Each word is shown with its category, a short example and a stack effect written as `( before -- after )`. This signature names what the word takes off the stack and what it leaves. `+`, for instance, is `( a b -- a+b )`: it takes two numbers and leaves their sum. Reading the stack effect tells you how a word fits into a line before you ever run it. If this information feels very cryptic, I encourage you to familiarize yourself with The Language and to come back after that to read this section.

The bar at the top holds two dictionaries, one per tab. Cagire is the language you sequence with. It is the main language used by

Cagire. Arf is the audio patching language that lives inside `{ ... }` blocks. It is used when you want to go deeper into sound synthesis and digital signal processing. Cagire generates that tab from the engine's own tables, so it always matches what the patch language actually accepts. A spelling like `lpf` can exist in both vocabularies with different meanings, and *Inside and outside the braces* (2.4) sorts them out.

Below the tabs the view has two panes. The first is a tree of categories on the left. Words of the chosen category appear on the right. `Tab` moves focus between them. Here are the keys to move around:

Key	What it does
←/→	Switch dictionary (Cagire / Arf)
`Tab`	Switch between the categories and the words
↑/↓ (or `j`/`k`)	Move through the focused pane
`PageUp` / `PageDown`	Move through it ten at a time
`Enter`	Fold or unfold a section
`/` (or `Ctrl+F`)	Search the dictionary you are on, by name
`Space` (or `F9`)	Start or stop playback while you read
`F10`	Stop and rewind

There is also limited support for mouse movements:

Mouse	What it does
Click a tab	Switch dictionary
Click a category	Select it, or fold it
Click a word	Focus the word list
Scroll	Change category, or scroll the words

2.1.1 User dictionary

You can define words to extend the language with your own vocabulary. These words you add live just next door in the **User Dictionary**. Press `u` to open it. This view becomes increasingly useful as you gain confidence and expertise. It lists your custom

words on the left and lets you edit them in the right pane. This view has a set of keybindings of its own:

Key	What it does
↑/↓ (or j/k)	Move through your names
PageUp / PageDown	Move through them ten at a time
Enter / →	Edit the name in focus
a	Add a word
r	Rename the name in focus
d	Delete it
Ctrl+E	Rebuild and evaluate your definitions
Tab / Esc	Back to the dictionary

Once more, there is limited support for the mouse in this view:

Mouse	What it does
Click a name	Select it
Double-click a name	Focus the body editor
Click the body	Focus it

Cagire reads the built-ins before your own words. A built-in can never be shadowed. The *The dictionary* (2.6) topic explains how the two layers fit together. Press ? for the full key list, and see *The language* (III) to learn how words are read and combined.

2.2 Patterns

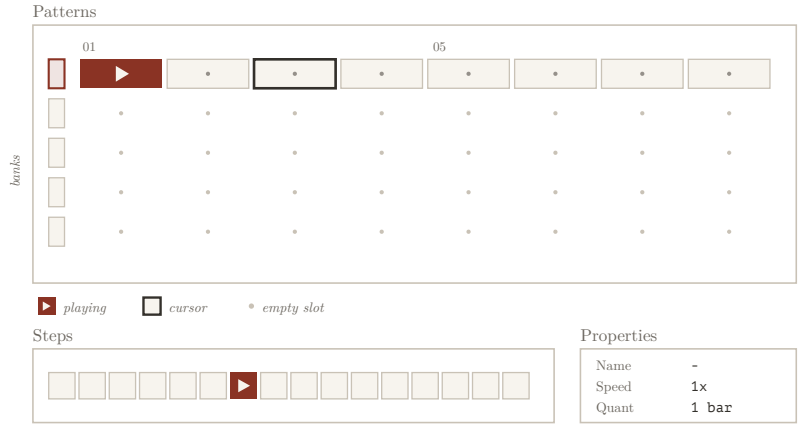


Figure 5: The Patterns view: banks in the left gutter, each bank's patterns beside it, the playing pattern marked, with the Steps strip and Properties box below.

The Patterns view is Cagire's clip launcher, similar in nature to what you can find in other traditional music software. Where the **Sequencer** is a view that makes you focus on one pattern up close, this view is the whole session at a glance. It shows every pattern you have made, laid out so you can start, stop, mute, and rearrange them while the music runs. A project holds up to 32 banks, each bank up to 32 patterns. More than one pattern can play at the same time. This is where you assemble and perform with patterns. Press `F2` to open it.

Banks run down the left edge as a gutter, and each bank's patterns fill the row beside it. A cursor moves freely across the two. Land it on a single pattern to act on that pattern, on a bank in the gutter to act on the whole bank, or hold `Shift` with the arrows to draw a selection and act on a block of either at once. Here are the main keybindings used to move around:

Key	What it does
←/→ ↑/↓	Move the cursor across banks and patterns

Key	What it does
`Enter`	Open the pattern under the cursor for editing
`Esc`	Clear a standing search, then drop a selection, then cancel an armed scene, then step back out
`Space`	Launch or stop the pattern under the cursor; starts the sequencer too if it is halted
`p`	Start or stop the sequencer
`P`	Stop the sequencer and rewind to the top
`F9`	Start or stop the sequencer, the same as `p`
`F10`	Stop the sequencer and rewind, the same as `P`
`+` / `-`	Nudge the tempo up or down, in beats per minute
`T`	Set the tempo exactly; tap `Space` in the dialog to find it by ear

And there are some actions with mouse support:

Mouse	What it does
Click a clip	Select it
Double-click a clip	Open it for editing
Click a bank in the gutter	Focus the whole bank
Scroll	Move the cursor

2.2.1 Finding a pattern

A full session gets crowded, so `/` opens a search bar over the clip names and Cagire moves the cursor to the first match as you type, matching any part of a name whatever its case. Press `Enter` and the bar closes but the query stands: every match keeps its highlight, and `Ctrl+n` and `Ctrl+p` go on walking the hits until you clear it.

Key	What it does
`/`	Search the grid by pattern name
`Ctrl+n` / `Ctrl+p`	Jump to the next or previous match, wrapping at the ends
`Enter`	Keep the query and close the bar; the matches stay highlighted

Key	What it does
`Esc`	While the bar is open, cancel and put the cursor back where it started; afterwards, clear the lingering highlights

2.2.2 How keys fire

By default the action keys take effect at once: `Space` plays or stops the pattern under the cursor, `m` mutes it, and `x` solos it. Press `h` to switch to hold mode, where those same three keys arm a change and `t` launches the whole scene together on your cue. See *Playing with patterns* (4.2) for how the immediate and hold modes work.

Key	What it does
`m` / `x`	Mute / solo the pattern under the cursor
`h`	Turn HOLD on or off
`t`	Launch the armed scene

2.2.3 Handing off to another pattern

A pattern can pass the baton on its own when it finishes a loop. Press `f` on a clip and the follow editor unfolds beside the grid, where single letters pick what plays next and a number sets how long the clip waits. An armed clip wears a small glyph in its cell. *Playing with patterns* (4.2) covers the follow editor in full.

Key	What it does
`f`	Open the follow editor on the pattern under the cursor

2.2.4 Keeping the project in order

The view is also where you keep the project tidy. The properties box that `e` opens carries a Color field as well, the same label color `C` cycles on the grid:

Key	What it does
`e`	Open the pattern's properties
`r`	Rename the pattern or bank under the cursor
`d`	Describe the pattern

Key	What it does
`C`	Cycle the pattern's label color; a selection takes the same color
`c` / `v`	Copy and paste
`[` / `]`	Reorder the pattern within its bank
`Delete`	Reset whatever is under the cursor, after you confirm
`g` / `G`	Share a pattern out, or bring a shared one in
`M` / `X`	Clear all mutes, or all solos, across the grid
`n`	Open the project notes
`Ctrl+Z` / `Ctrl+Shift+Z`	Undo, redo

2.3 Options

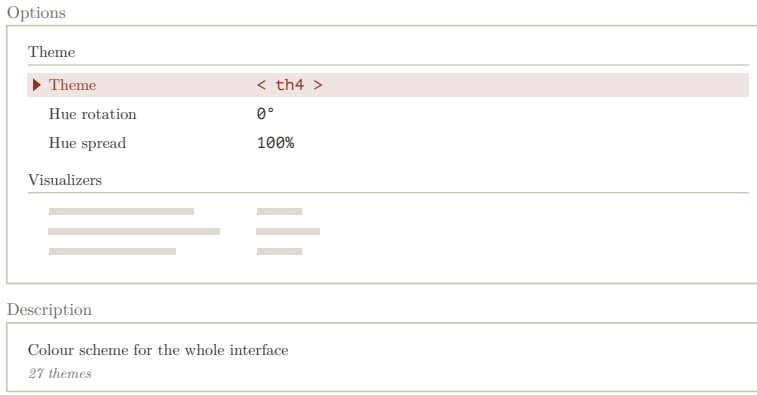


Figure 6: The Options view: settings grouped under section headings, the focused row highlighted, with a box below describing it.

The Options view holds Cagire's preferences: how the interface looks and how it behaves. All your choices are remembered from one session to the next. Press `F3` to open it. The settings are grouped into sections you read from top to bottom, each row showing its current value. Some rows are simple switches, others cycle through a few choices. The tables below list every setting, what it does, and the values it offers. To move around, use the following keys:

Key	What it does
↑/↓ (or `Tab` / `Shift+Tab`)	Move between settings
←/→	Change the focused setting
`Space` (or `F9`)	Start or stop playback, so you can judge a setting by ear
`F10`	Stop and rewind

With the mouse:

Mouse	What it does
Click a row	Focus it
Click the left or right half of a value	Step it down or up
Scroll	Move between settings

2.3.1 Theme

Cagire offers many themes. Some are based on live coders' preferences and visual identity, some are private jokes, and some come from popular code editors. Every bundled theme and every hue setting keeps text, selections, and borders at readable contrast. The hue controls reshape expressive accents while leaving core text and surfaces, plus universal red, green, yellow, and orange status meanings, stable. The opening title illustration is branded artwork and does not recolour.

Setting	What it does	Values
Theme	Colour scheme for the working interface	Bundled themes
Hue rotation	Rotates expressive accent hues while keeping text, surfaces, and status meanings stable	0–360°, step 5°
Hue spread	Compresses or stretches expressive accents around their mean hue	0–300%, step 5%

2.3.2 Visualizers

Cagire uses many audio visualizers throughout the interface. They are really helpful to get immediate visual feedback. They can

also be particularly helpful to understand if your changes are taking effect or even if sound is flowing through the system.

The rows are grouped by the page that draws them, so the heading above a setting tells you where the change will show up. Turning a visualizer off gives its space back to the page rather than leaving a gap.

Visualizers · Sequencer

Setting	What it does	Values
Master meter	Stereo master level in the column beside the sequencer grid	On / Off
Show scope	Draws the oscilloscope of the master output	On / Off
Scope style	Line trace or filled oscilloscope	Line / Filled
Scope orientation	Forces vertical, or follows the panel layout	Auto / Vertical
Show spectrum	Draws the spectrum analyser of the master output	On / Off
Spectrum style	Bar graph or filled spectrum	Bars / Filled
Spectrum orientation	Lays the spectrum horizontally or vertically	Horizontal / Vertical
Spectrum dB	Logarithmic dB scale with a −60..0 dB graticule	On / Off
Show lissajous	XY stereo phase scope, left against right	On / Off
Lissajous trails	Fading motion trail on the phase scope	On / Off
Show orbit levels	Level on each of the eight buses	On / Off
Orbit orientation	Forces vertical, or follows the panel layout	Auto / Vertical

Visualizers · Patterns

Setting	What it does	Values
Master meter	Stereo master level in the column beside the clip grid	On / Off

The Patterns view is otherwise all grid, so this meter is the one thing on the page that answers whether sound is coming out at all. It is the same meter the Sequencer view carries, with its own switch, because the two pages want their columns separately.

Visualizers · Shared

Setting	What it does	Values
Gain boost	Amplifies the scope and Lissajous waveforms	1x / 2x / 4x / 8x / 16x
Normalize	Auto-scales the visualisers to fill the display	On / Off

See *Visualizers* (2.9) for what each one shows and how to read it.

2.3.3 Editor

Cagire ships its own small code editor. This editor is fine tuned for efficiency while editing Cagire and Arf code. Some features are optional. In certain cases, they can even impair readability and turn you crazy. Try to toggle these on and off and see what you prefer.

Setting	What it does	Values
Runtime highlight	Highlights executed code spans during playback	On / Off
Completion	Word-completion popup in the editor	On / Off
Show preview	Step-script preview on the sequencer grid	On / Off

2.3.4 Interface

Setting	What it does	Values
Refresh rate	How often the screen redraws; lower spends less CPU	60 / 30 / 15 fps

Setting	What it does	Values
Performance mode	Hides the header and footer bars for the stage	On / Off
Check for updates	Asks at startup whether a newer Cagire was released, and says so once	On / Off

2.3.5 Project

Setting	What it does	Values
Pattern length	Length for untouched, new, and reset patterns	4 / 8 / 16 / 32 / 64 / 128 steps
Watch project file	Picks up edits made to the current `.cagire` file outside Cagire	Yes / No

Changing the default resizes untouched patterns immediately. Patterns you have edited or explicitly resized keep their length; a reset pattern takes the current default.

When this preference is **Yes**, the watch follows whichever file is open. With a demo, tutorial, or unsaved project there is nothing to watch, but the preference remains **Yes** and watching starts when a file is opened or saved. See *Watching the file* (4.5).

2.3.6 Onboarding

First-run help popups and toggling on/off the startup demos.

Setting	What it does	Values
Reset guides	Re-enables every dismissed guide popup	Action
Demo on startup	Loads a rotating demo song on a fresh start	On / Off

A box at the bottom of the view explains the focused setting and the range of values it takes. Every change takes effect at once and is saved as you make it, so the way you set Cagire up is the way you will find it the next time you open it.

Press `?` for the full key list.

2.4 Help

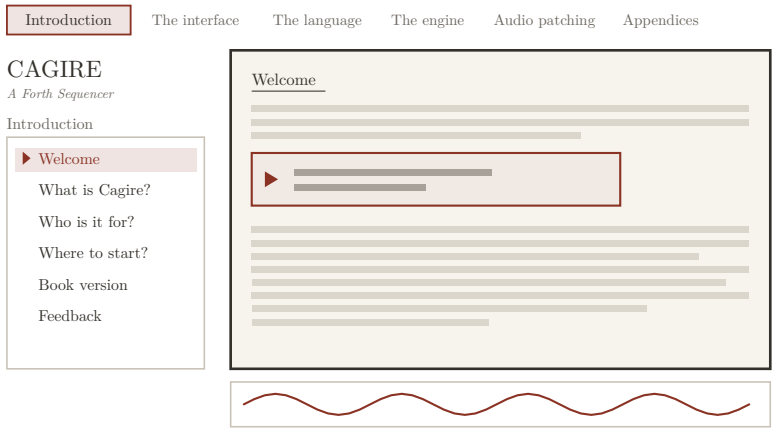


Figure 7: The Help view: chapter tabs along the top, the current chapter's sections in the sidebar, and the page beside them with a runnable example in focus.

The Help view is this manual built into Cagire. The documentation is always within reach when you play. There is no external resource to reach for. Press `F4` to open it, and `?` at any time to see the keys for the view you are in. The manual is laid out as a row of tabs along the top (*chapters*). The sidebar on the left is showing you *sections* side by side beneath it. The sidebar shows you the part you are currently reading, with headings you can fold away. The page itself fills the rest. Focus rests either in the sidebar or in the page and `Tab` moves between them. Here are the keybindings used to move around:

Key	What it does
←/→	Switch between the main parts
↑/↓ (or `j`/`k`)	Move through the sidebar, or scroll the page
`Ctrl`+↑/↓	Move through the sidebar five at a time
`PageUp` / `PageDown`	Scroll the page ten lines
`Tab`	Move focus between the sidebar and the page
`Enter`	Move focus into the page, or fold a heading away

Key	What it does
`Space` (or `F9`)	Start or stop playback while you read
`F10`	Stop and rewind

With the mouse:

Mouse	What it does
Click a part along the top	Switch to it
Click a topic in the sidebar	Open it
Click the page	Move focus into it
Scroll the sidebar, or the page	Move through topics, or scroll the page

2.4.1 Live examples

What sets this manual apart is that every code example in it is live. With focus in the page you can step from one example to the next and run the one in focus, so you read about a word and hear it in the same breath. You can go further and make an example your own: press `e` and it becomes a small editor, so you can change a value and run your version with `Ctrl+R`. Your edits last only while you are on the page and reset the moment you move on, so you can experiment without leaving a mark. Cagire clears any lingering effects before each run, so every example sounds the way it is meant to. You can also search the whole manual at once.

Key	What it does
`n` / `p`	Step to the next or previous example
`Enter`	Run the example in focus
`e`	Edit the example in focus
`Ctrl+R`	Run the example you are editing
`o`	Show or hide the signal scope below the page
`/` (or `Ctrl+F`)	Search the whole manual
`Esc`	Leave an example you are editing, clear the search, or step out of a focused example

2.5 Sequencer

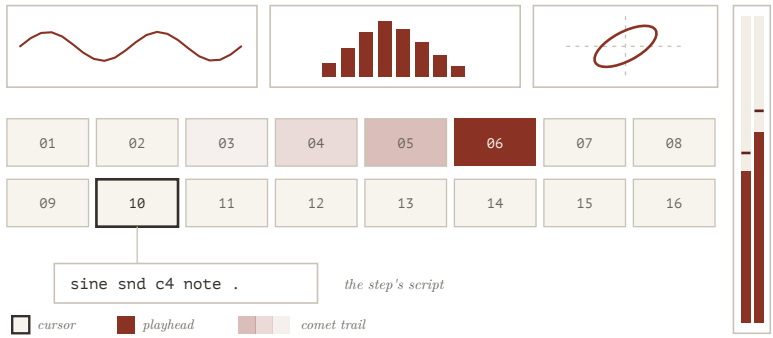


Figure 8: The Sequencer view: the step grid with the cursor, the playhead trailing its fading comet, and the focused step open on its script.

The Sequencer is where you build a pattern, one step at a time. It is the view you will spend the most time in. It is Cagire's home: press `F5` to return to it from anywhere, or find it at the centre of the navigation grid. Every other view is one key away from it! What makes it more than a grid of triggers is what a step holds. Each step holds a short script written in Cagire's language. When the playhead reaches a step it runs that script, and the script can do whatever it is written to do. A pattern runs anywhere from a single step to 1024, and you fill as much or as little of it as you like.

The grid fills the screen. A cursor marks the step in focus, and a fading comet trail follows the playhead as the pattern runs, so you can always see where the music is. Pressing `Enter` opens the editor on the focused step, taking half the page and leaving the pattern on the other half as a board of squares. You type a line, and it takes effect the next time that step comes round. There is no run button and no need to stop. You edit while the sound keeps playing. Try it. Open a step by pressing `Enter` and type:

```
sine snd c4 note .
```

Press `Ctrl+R` to preview the step on the spot, then `Esc` to save and hand the keys back to the pattern, which stays where it is on screen. Every key on this page works again at that point, so

you can copy the step, mirror it or divide it without the editor going anywhere; `Esc` once more closes it. You can also use `Ctrl+E` to evaluate without leaving the text. See *The step editor* (3.2) for everything else you can do once a step is open. There are keys that you should be aware of to move around efficiently:

Key	What it does
←/→	Previous / next step
↑/↓	Move the cursor up or down a row
`Enter`	Open the editor on the focused step
`Ctrl+R`	Run the focused step once, without opening it
`Space`	Start or stop the pattern you are editing, on the next boundary; starts the sequencer too if it is halted
`f`	Toggle Fill; steps read it with the `fill` word
`n`	Open the project notes
`o`	Move the visualizer strip to the top, bottom, left, or right edge
`F9`	Start or stop the sequencer
`F10`	Stop everything and rewind to the top

With the mouse:

Mouse	What it does
Click a tile	Move the cursor there
Double-click a tile	Open the editor on that step
Click a visualizer	Cycle its style
Double-click a visualizer	Flip its orientation
Click the lissajous	Toggle its trails
Scroll over the grid	Move the cursor up or down

2.5.1 Editing steps

These keys act on the steps you have selected, or on the single step under the cursor when nothing is selected:

Key	What it does
`t`	Switch the step on or off
`c` / `v`	Copy steps, paste them at the cursor

Key	What it does
`b`	Paste linked, as a mirror of the source
`h`	Harden mirrors back into independent steps
`i`	Save or insert a snippet over the step or selection
`Ctrl+D`	Duplicate the step or selection
`Delete`	Clear the step or selection, after you confirm
`Shift+`←/→ ↑/↓	Select a run of steps
`a`	Select every step of the pattern
`Esc`	Drop the selection
`Ctrl+Z` / `Ctrl+Shift+Z`	Undo, redo

2.5.2 Splitting and merging

A step can be cut into several cells, each with its own script, and a run of steps can be merged into one group cut the same way. The digit means one thing everywhere: cut the time you have chosen into that many equal cells.

Key	What it does
`2`-`9`	Cut the chosen time into that many equal cells
`1`	Over a selection, weld it into one step; over a group, give its slots back
`0`	Give the time back to the grid: one slot, one cell
← →	Walk the cells of a group, then on to the next step

Inside a group, `Enter`, `t`, `Del` and `Ctrl+R` act on the cell under the cursor rather than the whole step. See *Splitting and merging* (2.5.2).

Mirroring lets several steps share one script, so you edit the source and the rest follow. See *Mirrored steps* (3.6) for how it works. A script you keep reaching for can be saved under a name and stamped back in by name; see *Snippets* (3.4).

2.5.3 Filling steps

Four small tools write a whole run of steps at once, each seeded from the step under the cursor. Press the key, set a number or two, and `Enter` lays them down:

Key	What it does
`e`	Spread hits evenly, a Euclidean rhythm
`u`	Play the metrically strongest steps, a Barlow rhythm
`r`	Repeat the step every few steps
`z`	Fill steps by chance, at the odds you set

2.5.4 Shaping the pattern

How fast it runs, how it relates to the others, and how long it is:

Key	What it does
`+` / `-`	Nudge the tempo up or down, in beats per minute
`T`	Set the tempo exactly; tap `Space` in the dialog to find it by ear
`S`	Set the speed exactly
`W`	Set the swing, from 50 (straight) to 80
`L`	Set how many steps the pattern runs, from 1 to 1024

A speed multiplier lets one pattern run twice as fast as another while they keep time together. Swing drags every second step late for the classic drum-machine shuffle, heard immediately even mid-loop, and the header flags a pattern that swings. See *Pattern properties* (4.3) for the swing values and the rest of a pattern's settings.

2.5.5 Mute and solo

Key	What it does
`m`	Mute the current pattern
`x`	Solo the current pattern
`M` / `X`	Clear every mute, or every solo

2.5.6 Moving between patterns and banks

Step across without leaving the view, so you can carry an idea from one pattern to the next:

Key	What it does
`Alt+←/→`	Previous / next pattern
`Alt+↑/↓`	Previous / next bank
`Alt+b` / `Alt+f`	Previous / next bank, for terminals that swallow the arrows
`/`	Find a pattern by name and jump the cursor to it

The arrows step one slot at a time, which is fine next door and tedious across a project. `/` opens a finder over every pattern you have named in any bank, typing narrows the list as you go, and `Enter` moves the cursor without touching what is playing.

The search covers the bank name too, so typing a bank name keeps only its patterns. Only named patterns appear, and you give a pattern its name with `r` on the *Patterns* (2.2) view.

2.5.7 Preludes

When several steps need the same setup, put it in a prelude, code that runs ahead of the pattern so every step can rely on it:

Key	What it does
`p`	Edit the bank prelude
`P`	Edit the project prelude
`d`	Evaluate the prelude now

The everyday keys for saving, loading, notes, and quitting work here too; they are gathered in *Keybindings* (1.2). `g` and `G` also live on this view, packing the current pattern out to your clipboard and pulling a shared one back in; see *Exporting and sharing* (4.6). This view is quite complex. As you can see, it will take some time to be perfectly comfortable using it. The complexity is normal: you will spend up to 90% in that view and it is the control room of your entire performance.

2.6 Engine

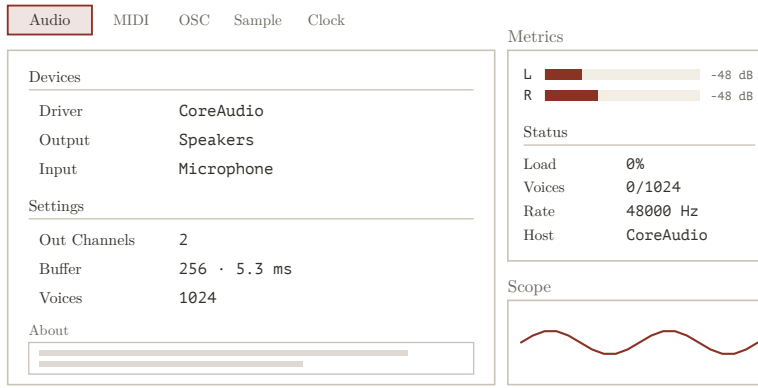


Figure 9: The Engine view: one tab per configuration area on the left, the monitor column tracking output levels and audio status on the right.

The Engine view is where Cagire meets the external world: audio, MIDI, OSC, synchronization, network, etc. Cagire also carries its own sound engine, **Doux**, built into the program. It means that you need a special place to configure it. This view gathers all of that in one place. Press `F6` to open the Engine view. A row of tabs runs along the top, one for each configuration area. The chosen tab's settings fill the space below. The right-hand side of the view monitors what is happening while you work: output levels, the voice count and CPU load, how hard the master limiter is working, the audio status, and a small scope. There are keys to move around that view:

Key	What it does
`Tab` / `Shift+Tab`	Move between tabs
↑/↓	Move through the current tab's rows
`PageUp` / `PageDown`	Move through them a screen at a time
←/→	Move through widget options
`Enter`	Open what the focused row holds: pick the audio device, edit the OSC endpoint, browse for a sample folder

Key	What it does
`Space`	Toggle the focused checkbox; on rows without one, start or stop playback
`F9`	Start or stop playback, whatever row you are on
`F10`	Stop and rewind

The mouse reaches the same controls. A click lands on whatever it points at, and clicking a value does what ←/→ would do on it:

Mouse	What it does
Click a tab	Switch to it
Click a row	Focus it
Click a value or an offset bar	Step it down on the left half, up on the right half
Click a checkbox	Toggle it
Double-click a row	Whatever `Enter` does on it: pick the audio device, edit the OSC endpoint, open the sample library
Scroll over the tabs	Move between tabs
Scroll over a tab	Move through its rows

A handful of quick controls are accessible through special keybindings. These are very useful and most of them are also hinted by the footer bar. Use them to diagnose and fix problems quickly:

Key	What it does
`h`	Hush everything at once
`p`	Panic, cutting any stuck voices
`R`	Restart the audio stream when a device changes underneath you
`t`	Play a short test tone
`D`	Rebuild the device list when you plug something in
`r`	Reset the peak-voice count in the monitor

On the Sample tab, two more keybindings are there to help you manage your audio sample library. `D` does double duty: it rebuilds the device list everywhere else, and removes a folder here.

Key	What it does
A	Add a folder to your sample library
D	Remove the selected folder from the list

2.7 Samples

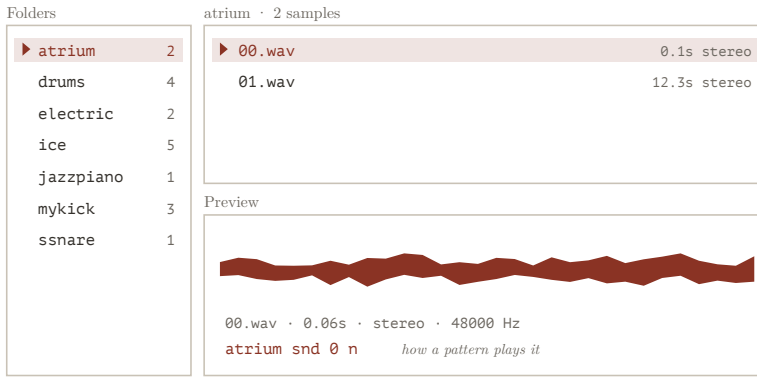


Figure 10: The Samples view: folders and their files side by side, with a preview showing the selected sample's waveform, its facts, and the words that play it.

The Samples view is your sound library: the audio files Cagire can play, gathered from the folders you point it at. A script plays a sample by its folder name. The library is the bridge between the files on your disk and the words in your patterns. This view is where you see what you have, hear it, and keep it in order. Press F8 to open it, or reach it from the navigation grid's bottom row or the Engine's Sample tab. The view is split in two. The folders of your library sit on the left, and the files inside the chosen folder fill the right. Tab moves focus between them. Here are the dedicated keybindings to move around:

Key	What it does
Tab	Swap focus between the folders and the files
→	Step into the files, then audition the file under the cursor
← (or h)	Back to the folders

Key	What it does
`Enter`	Step into the files
`↑`/`↓` (or `k`/`j`)	Move through the focused pane
`Ctrl+↑`/`↓`	Move through it ten at a time
`PageUp` / `PageDown`	Move through it twenty at a time

With the mouse:

Mouse	What it does
Click a folder	Switch to it
Click a file	Select and audition it
Scroll	Focus a pane and move through it

2.7.1 Managing the library

If you want to stay focused while playing live, you will have to tend your garden and manage your audio sample library. You will sometimes have to rename audio files or even folders. There are dedicated keybindings for all of this:

Key	What it does
`r`	Rename the file or folder
`d`	Delete it from disk, after you confirm
`y`	Duplicate a file, to keep a variant
`l`	Open the file in the Sampler, to trim or reshape it

`d` here is the real thing, not a tidying of the list. On a file it deletes that file. On a folder it deletes the folder and everything inside it. Cagire asks first, and there is no undo once you answer yes. The Engine's Sample tab has a similar-looking key that only forgets a path and leaves the disk untouched, so know which of the two you are standing in. See *Sample folders* (6.4).

A sample answers to `n`, its position in the folder counted alphabetically from zero. Adding or removing a file moves every position after it, so a pattern that asked for `n 3` now plays what used to be `n 4`. Cagire tells you when an edit has done this: the delete dialog

names the slot the shift starts at, and a rename or a duplicate says so afterwards in the status line.

2.7.2 Narrowing a pane

Each pane carries its own filter, so you narrow the folders and the files apart from each other. `/` starts one on the pane you are standing in.

Key	What it does
`/`	Narrow the focused pane by name
`Enter`	Keep the filter and step on: into the files, or audition the file
`Tab`	Keep the filter and hand the keys to the other pane
`Esc`	Drop the filter on the focused pane, leaving the other one

Type on the folders to find the kit, `Tab` across, then type again to find the sample inside it. The folder filter holds while you work in the files, and the file filter holds while you walk from folder to folder, which is how you find the same-named sample in each of them. The filters are fuzzy, so `bd` reaches `big drum` and `909kck` reaches `909 kick`. A file keeps the `n` it was born with however far the list is narrowed, and the row shows it.

The folders the library is built from are added in the Engine's Sample tab. The full key list sits under `?`, and *Sound resources* (5) covers loading, recording, and editing samples.

2.8 Sampler

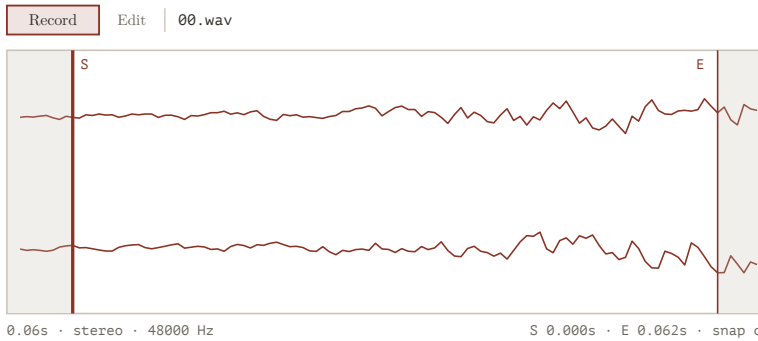


Figure 11: The Sampler with its Record and Edit modes; the stereo waveform fills the view and the start and end markers trim the region to keep.

The Sampler is Cagire's recorder and sample editor in one. It captures sound coming into the machine, lets you trim and reshape it, and saves the result as a file in your library to play back later. If the Samples view is for the sounds you already have, the Sampler is where new ones are made. Press `F7` to open it. It works in two modes, and `Tab` switches between them. `Record` is the arming screen where you set a source and capture a take. `Edit` is where the take lands, and where you trim, reshape, and save it. You never have to reach for `Tab` after a capture: a finished take drops you into Edit on its own. A few keys work whatever mode you are in:

Key	What it does
`Space`	Audition the trimmed region, to hear it before you commit; press it again to stop
`Esc`	Throw the take away, after you confirm
`Tab`	Switch between Record and Edit
`l`	Load an existing audio file directly into Edit; no recording is required

This view is very keyboard centric. Working quickly here means knowing where the keys sit, so the tables below are the map. Record

arms the take, and its keys answer only while nothing is being captured:

Key	What it does
←/→	Choose the input
↑/↓	Choose the record mode
[/]	Set the length: a seconds cap when free, a bar count when quantized
r	Record, or stop

Everything you do to a take you do in Edit, so that is where the waveform keys live:

Key	What it does
s / e	Choose the marker you move, the start or the end
←/→	Nudge the active marker
g then a letter	Jump the active marker to a zone of the waveform
+ / -	Zoom in and out
↑/↓	Exaggerate the peaks, or settle them back
z	Snap markers to zero crossings, or stop

With the mouse:

Mouse	What it does
Click or drag in the waveform	Position the active marker
Scroll	Zoom the waveform in or out

You can record by hand or in time with the bar, so a take drops cleanly into a pattern. Once you have a take, set the start and end markers around the part you want and reshape it, each edit undoable:

Key	What it does
p	Open the process picker: normalise, fades, reverse, gain, crop, and the rest
u	Undo the last edit
Ctrl+S	Save the result into a folder in your library

Every tool and export sits behind `p`, so there is one list to learn instead of a dozen shortcuts to collide with the markers. Type to filter it and `Enter` runs the highlighted process on your marked region. The two slice export entries leave the take unchanged: one keeps an equal grid, the other moves the cuts to local zero crossings, and both write a numbered WAV collection into a folder. Press `?` for the full key set. See *Sound resources* (5) for recording and editing samples in depth. If the audio engine is not running, the Sampler cannot record or play anything back, so the Record screen shows a red notice in its centre. Open the Engine view with `F6`, pick a working device, and restart the engine; the notice clears the moment sound is flowing again.

2.9 Visualizers

Cagire has many widgets used to draw the sound output. Because we run Cagire in a terminal emulator, don't expect to find pixel-level precision. Visualizations can give you the best approximation, which is often more than enough. There are a handful of visualizers spread across a few views. Each one gives you a different angle on the sound: how loud it is, what its shape looks like, where its energy sits, and how it spreads across the stereo field.

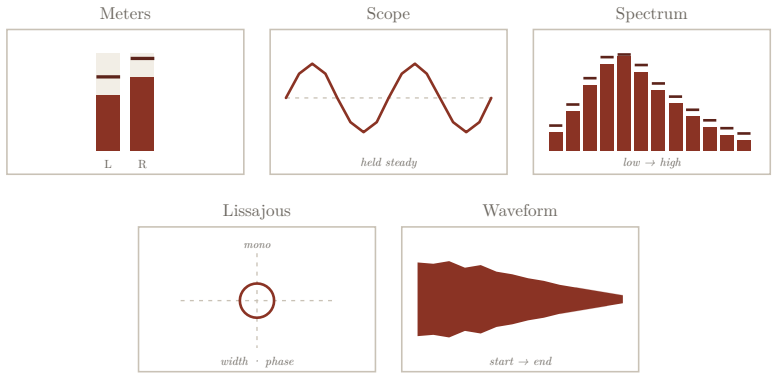


Figure 12: The five visualizers at a glance: stereo dB meters, a triggered scope, a frequency spectrum, a mid/side Lissajous, and a sample's filled waveform.

Widget	What it shows	Where you find it
Meters	Left and right level, in dB	Sequencer, Patterns, Engine, Sampler
Scope	The output waveform, held steady	Sequencer, Engine
Spectrum	Energy across the frequency range	Sequencer
Lissajous	Left against right: width and phase	Sequencer
Orbit levels	Level on each of the eight buses	Sequencer
Waveform	A sample's shape, start to end	Sampler, Samples, Sample finder

2.9.1 Options

You can customize the look of visualizers in the Options view, under *Visualizers* (2.9). Those rows are grouped by the page that draws them, so you always know where a change will land. On the main Sequencer view you can also reach for them with the mouse. Click a widget to cycle its style and double-click to flip how it is laid out. See *Options* (2.3) for turning each one on and shaping it, and *Engine* (2.6) for the meters and scope in the monitor.

The Sequencer and Patterns views each carry the master meter in a column down their right edge, each with its own switch. On the Patterns view it is the only visualizer there is: the rest of that page is the clip grid, so the meter is what tells you sound is still coming out.

The orbit levels are the one visualizer that answers a question the others cannot: which bus a sound is on, and how hard it is hitting. See *Monitoring orbits* (9.4).

Chapter 3

Using the sequencer

3.1 What is a step

A step is one slot of a pattern. A pattern is `n` steps long. Inside each slot sits a small script, a handful of words in the Cagire language. Cagire runs that script the moment the playhead reaches the step. One step might play a note, the next might change a sound, another might do nothing at all. A pattern is just a row of these steps run in order and looped. A step is either *on* or *off*. That is separate from what it holds. Press `t` to toggle it. An *off* step keeps its script but stays quiet until you switch it back *on*. An *on* step with an empty script simply passes. You can silence a step without losing what you wrote then bring it back later untouched. Edit a step's script, evaluate the changes, and they will take effect the next time the playhead comes round to it while the music keeps playing. If a script has an error, the step does not fall silent. It keeps sounding its last working version and shows a small error sign so you can find it. *The step editor* (3.2) covers writing the script itself.

The grid shows you where you are and where the music is. The cursor (drawn as `<` `>` around a step) is where your focus sits. The playhead, a ▸, is the step playing now, and a fading trail follows behind it like a comet. A pattern's length sets how many steps loop, from 1 to 1024. Shorten it and the steps past the end are hidden, not erased. Grow it again and they come back as they were. So you can keep a longer idea parked beyond the loop and reveal it when you want it.

Key	What it does
`Enter`	Open the editor on the focused step
`t`	Switch the step on or off

Key	What it does
←/→ ↑/↓	Move the cursor across the grid
L	Set how many steps the pattern runs

See *The step editor* (3.2) to write a step, *Filling steps* (3.5) to lay down many at once, and *Mirrored steps* (3.6) to share one script across several.

3.2 The step editor



Figure 13: The step editor beside the board, with the focused step's script under the text cursor and the pattern laid out as squares on the right.

Press `Enter` on a step and Cagire opens the step editor on it. This is where a step's script lives, the few words that run each time the playhead arrives. The same editor opens for preludes, so everything here applies to those as well. A script is not limited to one line: write as many as you like. Start a comment with `;;` to leave yourself a note. What you type takes effect the next time the step comes round but only if you have evaluated the code before. It means that you can edit while the music plays. As you type, Cagire suggests words that fit. Press `Tab` to take a suggestion, and `Ctrl+N` (*next*) or `Ctrl+P` (*previous*) to navigate between suggestions. When you need a sample, `Ctrl+B` opens the sample finder.

The editor does not float over the sequencer. It takes half the page and the pattern takes the other half, drawn as a board: one square per step, the squares parted by a gap and shaded light and

dark so you can count them. The board is not the sequencer grid in miniature. The grid packs eight steps to a row because that is how a beat wants to be read; the board lays the pattern out to fill its half of the page, four or eight or sixteen across depending on the shape of the space, so the whole pattern is in front of you while you write.

Every square is the same size as its neighbours, and the board keeps them big enough to read rather than fitting the pattern in at any cost. A pattern too long for that pages, following the cursor as the sequencer grid does, and the line above the board says which steps are on screen, like `01-32/64`.

Every square carries its step number, written the way the grid writes it: `07` plain, `·07·` when the step holds a script, `→03` when it mirrors step 3, ⚠ when its last run failed, and `< 07 >` when it is the one you are editing. The colours are the grid's too, dim for an empty step and lit for one holding a script, but the number is what tells you where you are. Colour alone is a poor answer to that. The cursor square is also inverted and ruled underneath, so it is three cues, not one. When a pattern is long enough that the squares shrink, the brackets go first and the number last.

The rest of what the grid says about a step, the board says in the same places. A step's name sits above its number and the first word of its script sits below, so a square tells you what it plays and not only that it plays something. A divided step draws its cells on that lower row instead, one block per cell, each as wide as its cell is long, with the cell sounding now lit and the cell you are on underlined. A selection tints every step it covers, and a merged group closes the gaps between its squares so the run reads as one bar. The line above the board names the cell you are on, like `step 09 · 3/5`, and so does the editor's own title.

3.2.1 The keys belong to one pane at a time

`Esc` hands the keys to the board and leaves the editor on screen. `Enter` hands them back. `Esc` again, with the board listening, closes the editor and gives the sequencer the whole page.

Nothing moves when the keys change hands, so three things say which pane has them. A ▸ sits at the head of the listening pane's title line, in the same place on both, and the other title goes quiet. Only the listening pane shows a live cursor: the editor's block cursor when the text has the keys, the inverted square when the board does. The parked pane keeps its position readable, the board's square still bracketed and the editor's text still there, but neither is lit. The footer names the keys that are live.

Key	Where it takes you
`Enter`	Into the text
`Esc`	Out to the board, editor still on screen
`Esc` again	Editor closed, sequencer full page

The point of that middle state is that **every key the sequencer has works in it, unchanged**. Copy a step, paste it, mirror it, harden it, divide it, fill it, mute it, walk the pattern with the arrows: the same keys in the same place doing the same thing, with the script you are working on still in view beside them. *What is a step* (3.1) and the pages after it cover those keys.

That is also why the editor takes no key the text has a use for. `Tab` indents and takes suggestions, `?` types a question mark, the digits type digits. Nothing is borrowed, because the board is one keystroke away.

Cagire saves what you typed before the keys leave the editor, so stepping out never costs you a word.

3.2.2 Moving without leaving the text

`Alt+Shift` and the arrows walk the bank and pattern grid: left and right along the patterns, up and down through the banks. The same four bindings the sequencer has, so the chord means one thing wherever you press it.

Key	What it does
`Alt+Shift+`← / `Alt+Shift+`→	Previous / next pattern
`Alt+Shift+`↑ / `Alt+Shift+`↓	Previous / next bank, on the pattern you were already on

Key	What it does
F9 / F10	Start or stop, and stop and rewind

A bank move keeps its pattern: B01 P03 down is B02 P03. A card shows where you landed, a fragment of that grid with the numbers around it, the patterns that hold something marked, and the one you arrived on bracketed.

Steps are plain arrows on the board, one Esc away with nothing repainting on the trip. The preludes are the board's p and P, the same trip.

Clicking either pane focuses it, and a click on the board also moves the step cursor, so one press does both.

The two panes sit side by side when the terminal is wide enough for both, and stack with the board on top when it is not. A pattern longer than the board can show pages to keep the step you are on in view.

3.2.3 Editing a mirrored step

A mirror holds no script of its own, so editing one edits the step it plays. The title says where the text really lives, as Step 09 → 03. Saving writes to step 3, and every mirror of it changes with it. The cursor stays on the mirror rather than jumping to the source, so you can carry on walking a run of them.

3.2.4 Keys inside the text

Key	What it does
Tab	Take the suggested word, or indent; on terminals that cannot distinguish it from Ctrl+I, open snippets instead
Ctrl+R	Run the script once, to hear it now
Ctrl+E	Evaluate it without leaving the text
Esc	Save and hand the keys to the board, or leave the search first
Ctrl+F	Search within the script
Ctrl+N / Ctrl+P	Next / previous suggestion, or search match
Ctrl+B	Find a sample and insert its name

Key	What it does
`Ctrl+I`	Insert a saved snippet at the cursor
`Ctrl+A / C / X / V`	Select all, copy, cut, paste
`Shift+←/→ ↑/↓`	Select text
`Ctrl+U`	Undo

`Ctrl+F` opens the search bar. Cagire highlights every match as you type and takes you to the nearest one straight away, so you never have to guess whether a word is in there. The match you are on is picked out in its own colour, and the right of the bar keeps count of your place: `2/7`, or `no match` when there is nothing to find. `Ctrl+N` and `Ctrl+P` walk the matches and wrap round at either end, and each jump lands in the middle of the view so you can see what surrounds it. `Enter` keeps the search alive, so you can carry on walking it while you edit. `Esc` abandons it and puts the cursor back exactly where it was before you went looking. Once a search is accepted, the first `Esc` clears it and the next hands the keys to the board.

While the text has the keys it keeps every one of them, `?` included, so a stray question mark lands in your script rather than opening the key list. The footer carries the hints you need instead, and changes with the pane that is listening. See *Dictionary* (2.1) for the words a script can use, *The sample finder* (3.3) for dropping a sample in by name, and *Snippets* (3.4) for saving a script to reuse by name.

3.3 The sample finder

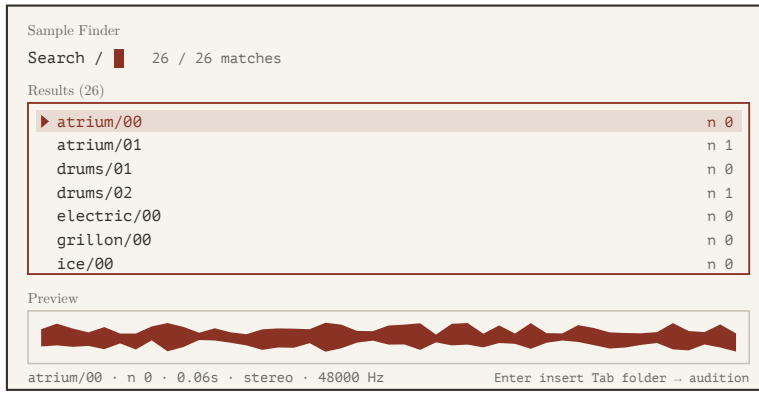


Figure 14: The sample finder: a fuzzy search line above folder/number result rows, with the highlighted sound's waveform and facts previewed below.

Writing a step, you often know the sound you want long before you remember which bank it lives in or what the number of the desired sample is. The sample finder closes that gap. Press `Ctrl+B` in the step editor and it opens over your script. Type a few letters of what you are after, find it in the list, and Cagire drops the words to play it straight in where your cursor was. The search is fuzzy and runs over the whole path of each sound. Type more to narrow further.

It searches the sample folders you have registered, the same library the *Sample folders* (6.4) tab builds. Anything you can play is anything you can find here. Picking a sound using the `Enter` key puts text into your script, inserting text in place. It will for instance write `hats snd 2 n`. When you would rather choose the number yourself, `Tab` writes just the bank word, `hats`, and leaves the rest to you. You can also listen before you commit. → plays the highlighted sound once. Turn on auto-audition with `Ctrl+A` and each sound plays the moment you land on it. You can browse the whole bank by ear with the arrow keys. A panel beneath the list shows the highlighted sound's waveform and tells you its length, its channels, and its rate.

Cagire remembers the finder between visits. Reopen it and your search text, your place in the list, and auto-audition come back as you left them. When your cursor already sits on a bank word, Cagire searches for that bank instead, so pressing `Ctrl+B` on `hats` opens straight into the hats sounds. `Ctrl+R` throws you at a random match, which is the quickest way to dig when you do not yet know what you want.

Key	What it does
`Ctrl+B`	Open the finder, from a step or the dictionary editor
`type / Backspace`	Narrow the list, or delete a letter
↑/↓ (or `Ctrl+P / Ctrl+N`)	Move through the results
`PageUp / PageDown`	Jump ten at a time
→	Audition the highlighted sound
`Ctrl+A`	Auto-audition: play each sound as you reach it
`Ctrl+R`	Jump to a random match
`Enter`	Insert the phrase to play it (`hats snd 2 n`)
`Tab`	Insert just the bank name (`hats`)
`Esc`	Close without inserting anything

The mouse reaches the list as well:

Mouse	What it does
Click a row	Highlight it, and play it when auto-audition is on
Double-click a row	Insert the phrase, the same as `Enter`
Click the waveform	Audition the highlighted sound
Wheel	Move through the results
Click outside the box	Close without inserting anything

3.4 Snippets

A snippet is a step's script saved under a name, so you can drop it back in later without retyping it. When a phrase proves useful, a bass figure, a drum trigger, a chord voicing you keep reaching for, save it once and reach for it by name from then on.

Cagire keeps your snippets with the application, not inside the song. They are yours, not the project's, so they follow you from one `.cagire` file to the next and survive across sessions. Saving under a name that already exists overwrites it, no questions asked; the list in front of you shows the name as you type, so you can see what you are about to replace. This is the same idea as the user dictionary on the *Dictionary* (2.1) view, which keeps your personal words the same way, and the same insert-by-name gesture as *The sample finder* (3.3).

Each snippet is a plain text file in a `snippets` folder next to Cagire's configuration file, one file per snippet, named after it. Cagire re-reads that folder every time you open the prompt, so you can add, edit, or remove snippet files with any editor while Cagire runs, and the change is there the next time the prompt opens.

3.4.1 Opening the prompt

Key	What it does
`i`	Open it on the sequencer; it acts on the selected step, or the whole selection
`Ctrl+I`	Open it inside the step editor; it inserts at the cursor

On terminals that cannot distinguish `Ctrl+I` from `Tab`, including tmux without extended keys enabled, `Tab` opens the prompt from the editor too.

3.4.2 Inside the prompt

Type to filter the list, move through it with the arrows, and the body of the highlighted snippet shows in the panel below so you see the script before you commit to it.

Key	What it does
`Enter`	Insert the highlighted snippet
`Ctrl+S`	Save the current script under the typed name
`Delete`	Remove the highlighted snippet, after you confirm
`Esc`	Close without changing anything

Where the snippet lands depends on where you opened the prompt. From the step editor it drops in at the cursor, like typing it yourself. From the sequencer it replaces the script of the step you are on, or of every step in the selection, all stamped with the same phrase. A mirrored step hardens into its own copy when a snippet lands on it, exactly as a paste would.

A snippet holds script text and nothing else. The step's name, its on or off state, and any mirror link are not part of it; the script is the essence, so that is all Cagire keeps. Save one from a step that plays a sine and what you get back is the line itself:

```
sine snd c4 note .
```

3.5 Filling steps

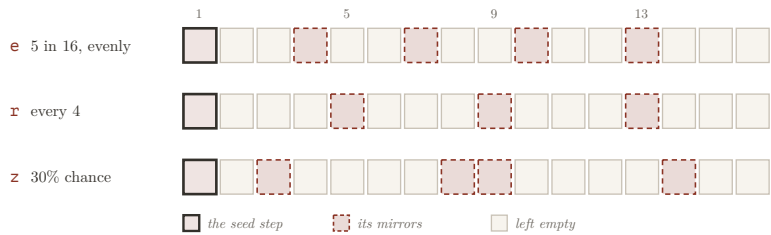


Figure 15: Three of the fill tools over the same steps, all seeded from the cursor: e spreads evenly, r repeats, z scatters.

Sometimes you want a whole run of steps at once, not one at a time. Four tools do that, each starting from the step under the cursor. Press the key, answer a small prompt, and `Enter` lays the steps down.

- **`e` spreads hits evenly**, making a Euclidean rhythm. You give it a number of hits and a number of steps, and it spaces the hits as evenly as it can across them. A rotation shifts where the pattern starts.
- **`u` picks the strongest steps**, a Barlow rhythm. Same two numbers, but instead of spacing hits evenly it ranks every step by how much the bar leans on it and plays the top few. Raising the hit count thickens the pattern rather than rearranging it. **Group** is how many steps make a beat, which is what tells `12` steps as three beats of four from four beats of three; leave it at `0` to let Cagire choose.
- **`r` repeats every few steps.** You give it an interval, and it drops a hit at every multiple of it.
- **`z` fills by chance.** You give it a percentage, and each empty step is filled at those odds. Run it again for a different scatter.

There is one thing to know about all four. They do not copy the step under the cursor, they mirror it. Every step they lay down shares the seed step's script. So fill from an empty step and you get silence. Fill from a step that plays a kick and the whole run plays that kick. Better still, edit the seed afterwards and all of them follow at once. When you want them to part ways, `h` hardens them into independent steps. *Mirrored steps* (3.6) explains the link in full.

Each dialog has a **Mode** that decides what happens to steps already in the way. **Fill empty**, the default, lays the new steps only on empty ground and leaves what you built alone. **Overwrite** writes straight over it. Mode is a row like the rest: reach it with `↑/↓` and flip it with `←/→`. On an empty stretch the two do the same thing, so it only matters when you fill across a part you already made.

Key	What it does
`e`	Spread hits evenly, a Euclidean rhythm
`u`	Play the metrically strongest steps, a Barlow rhythm
`r`	Repeat the step every few steps
`z`	Fill steps by chance, at the odds you set
`↑/↓`	Move between a dialog's fields, Mode included

Key	What it does
←/→	Adjust the focused field, or flip Overwrite / Fill empty
`Enter`	Lay the steps down
`h`	Harden the filled steps into independent ones

3.6 Mirrored steps

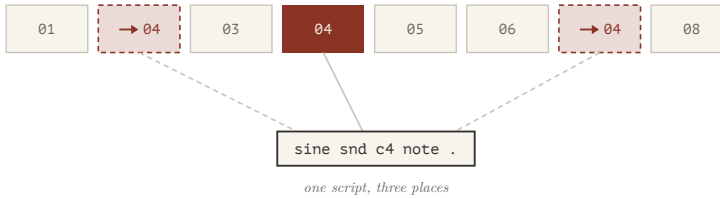


Figure 16: Step 04 is the bright source; 02 and 07 mirror it as dimmed steps marked →04, all sharing one script.

A mirror is a step that borrows another step's script instead of keeping its own. The two play the same thing, because there is only one script between them. Change the source and the mirror changes with it, the moment the change is made. It is one part shared across several places in the pattern, not a copy that drifts. To make one, copy a step with `c`, move the cursor to where you want the mirror, and paste it linked with `b`. Where a plain paste drops an independent copy, `b` drops a mirror that points back at the source. Mirroring stays within a single pattern; you cannot link a step in one pattern to a step in another.

Cagire marks a mirror so you can read it at a glance. It shows an arrow to the step it follows, like →`04`, and wears a dimmed shade of that step's colour. The bright step is the source, and the dim ones lean on it. When you want a mirror to go its own way, `h` hardens it. Cagire copies the shared script into the step and cuts the link, leaving an ordinary step you can edit on its own. And if you delete a source, its mirrors do not keep playing without it; they empty out alongside it. The fill tools work the same way underneath. The runs that `e`, `r`, and `z` lay down are mirrors of the step you filled from,

which is why editing that one step changes them all. See *Filling steps* (3.5).

Key	What it does
`c`	Copy the step under the cursor
`b`	Paste it linked, as a mirror of the source
`h`	Harden mirrors back into independent steps

3.7 Splitting and merging

Every step owns one slot of the grid and plays one script in it. Splitting hands that slot to several cells instead, each with its own script, each sounding at an even fraction of the time. Press a digit from `1` to `9` on the step under the cursor and Cagire cuts it that many ways. A step split in three plays three different things where it used to play one, so a roll, a flam, or a quick answering phrase fits inside a single slot.

Merging is the same gesture over more time. Select a run of steps with `Shift+`← or `Shift+`→, press a digit, and Cagire fuses them into one group cut that many ways. The digit means one thing everywhere: cut the time you have chosen into that many equal cells. Select three steps, press `4`, and four cells spread evenly across three slots, which is a four-against-three figure sitting inside a straight grid. Nothing about the pattern's length, its speed, or the tempo changes. The group still occupies its three slots; it just fills them differently.

`0` is the way back. It gives the time to the grid again, one slot and one cell, and keeps every script. Press `a` to select the whole pattern, then `0`, and you have a plain grid again however far you went.

`1` does the same to the group under the cursor. One cell is no division, and a group with nothing left to divide is just its slots back, so the key that undoes a split undoes the merge that came with it. Select the run first if you want to keep the group, because `1` over a selection welds it instead, into one step that fires once across

all of them. Cagire applies a split or a merge the moment you press the key, mid-loop, the same way it applies swing.

The slots a group covers keep their scripts. They lie dormant while the group owns their time, so merging never throws anything away and dissolving a group gives you the old rhythm back exactly as it was. Their tiles keep their numbers too, and the mark that says they hold something, so nothing goes quietly missing under a merge.

3.7.1 Reading the grid

One step is always one tile. A merge never moves a tile, resizes one, or changes the shape of the grid, so what you learned to read stays where it is.

A group is drawn by closing up. Its tiles take one colour, and the gaps between them fill with it, so three separate tiles become one continuous bar. Nothing is drawn over the grid to explain this. The steps simply stop having air between them, the way they would if they really were one. The bar ends where the group ends, and a group that runs over the end of a row is drawn as one bar per row.

Along the bottom of the bar runs a band, one block per cell, each block as wide as its cell is long. Every column of the band belongs to exactly one cell, so how wide a cell looks is how long it lasts. The band is built from the group's own colour rather than from new ones. A cell that sounds stands well clear of the tile, and an empty or switched-off one barely lifts off it. Every other block is a shade lighter than its neighbours, so you can count a division by eye without cluttering the grid. The cell playing now takes the playhead's colour, and the cell under the cursor is underlined, so neither can hide the other. Click a block and the cursor lands on that cell.

Four cells across three slots is four equal blocks spanning the whole bar, which is the four-against-three figure itself, drawn at the size it really occupies. The panel beside the grid names what you are on, like `Cell 05.2 of 4:3`, read the way a musician writes a tuplet: four cells in the time of three slots.

On a terminal too narrow to give every cell a column of its own, Cagire refuses to draw a band that would not be proportional. The first tile shows how many cells there are instead, like `:9`, and you walk them with the arrow keys as usual.

3.7.2 Working inside a group

Cells behave like small steps in every way that matters. The arrow keys walk into them left to right, then carry on past the whole group, so the cursor always follows the timeline and never lands on a slot a group has covered. `Enter` opens the editor on the cell under the cursor, `t` switches that one cell on or off while its neighbours keep playing, and `Del` empties it and leaves the division alone. `Ctrl+R` runs the focused cell once.

Selecting is always about whole steps, so `Shift+←` and `Shift+→` step over a group in one move and leave the cell cursor where it is. A group is one thing, so a selection that reaches any slot of one takes all of it: the highlight covers the whole group, and so does whatever you do next. With a selection, `t`, `c` and `Del` act on whole steps again. Without one they act on the cell you are looking at. That is the only rule.

The clipboard follows it on the way in and then remembers. `c` on a cell puts that one cell aside, script and switch, and `v` drops it into whichever cell the cursor is on, in this group or another, leaving the division it lands in alone. What you copied decides the shape of the paste, so a cell stays a cell until you copy something else, whatever is selected when you paste it. Copy a whole step instead by selecting it first, and the copy carries the step's shape with it, so pasting it somewhere else brings the merge and the division along. A cell pasted over a mirrored step hardens it first, exactly as typing into one does. `b` and `h` stay about whole steps, because a mirror is a whole step pointing at another.

Each cell counts its own loops, so `cycle`, `walk` and `urn` advance separately in each one and two cells can run down two different lists at their own pace.

Dividing moves notes closer together. It does not make them shorter. A cell's gate and envelope times stay measured against

a whole step. Its `delaytime` setting stays a fraction of a whole note, so a pad divided in four is still a pad, played four times. This matches what `at` already does inside a script, and it is usually what you want. If you also need the notes clipped, set `gate` yourself or reach for `stut`. When the pattern swings, the cells swing with it: swing steals time from the off-beat, so a group's cells compress or stretch to fit the time it really owns rather than spilling into the step behind it.

Cells take equal shares, always. For an uneven figure, use two groups side by side: merge the first two steps into one cell and cut the next two in three, and you have a half followed by three across the other half. Cagire draws them as two groups because that is what they are, and the grid still shows exactly where everything sounds.

A mirror plays its source's cells, spread across its own slots, so one script can be a triplet in one place and the same triplet somewhere else. Cutting a mirror is an edit to the mirror, so Cagire hardens it first and gives it cells of its own, exactly as typing into one already does. The step it was following is left alone, and so is every other mirror of that step.

3.7.3 Moving around

Key	What it does
`2`-`9`	Cut the chosen time into that many equal cells
`1`	Over a selection, weld it into one step; over a group, give its slots back
`0`	Give the time back to the grid: one slot, one cell
`a`	Select every step of the pattern
`a` then `0`	Flatten the whole pattern, scripts kept
`Shift`+← `Shift`+→	Extend the selection by whole steps, then a digit merges them
← →	Walk the cells, then on to the next step
`Enter`	Open the editor on the cell under the cursor
`t`	Switch the cell under the cursor on or off
`Del`	Empty the cell under the cursor, keeping the division

Key	What it does
`c v`	Copy the cell under the cursor, paste it into another
`Ctrl+R`	Run the cell under the cursor once

See *What is a step* (3.1) for the step itself, *Mirrored steps* (3.6) for sharing one script across several places, and *Periodic execution* (3.14.1) for `at` and the other ways to place events inside a single step.

Chapter 4

Project & session

4.1 Session organization

A project is the whole of your work in Cagire. It nests in four layers. A project holds banks, a bank holds patterns, and a pattern holds steps. You fill a step with a short script, and that is where the music is made.

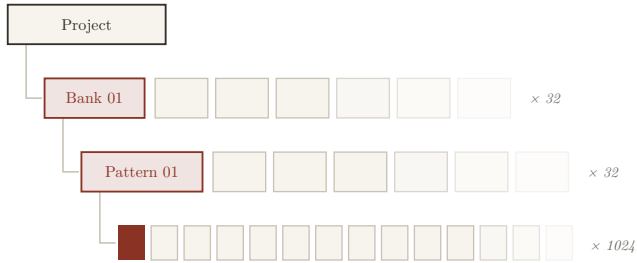


Figure 17: A project nests in four layers: banks hold patterns, patterns hold steps, and a step holds the short script.

A project holds up to 32 banks. Each bank holds up to 32 patterns. Each pattern runs up to 1024 steps. The *Patterns* (2.2) view shows the banks and patterns as one grid. The *Sequencer* (2.5) shows one pattern up close, the one you have selected, step by step. One tempo governs the whole project. What changes from pattern to pattern is the playing speed. Each pattern carries its own speed multiplier, so one can run twice as fast as another while both stay locked to the same beat. Tempo is the shared pulse; speed is how each pattern divides it. See *Pattern properties* (4.3) for a pattern's speed and *Playing with patterns* (4.2) for how patterns run side by side.



Figure 18: Both preludes run ahead of the steps, the project prelude first, then the bank's, so shared setup is ready before the pattern plays.

Two scripts run ahead of your steps. The project prelude runs first, for the whole project, and each bank has a prelude of its own as well. Put your shared setup there, the words and values every step relies on, and the steps stay short. You edit both from the Sequencer. The language section explains what a prelude is good for. Notes belong to the project too. They travel inside the file and come back when you load it. See *Taking notes* (4.7).

Key	What it does
`F2`	Open the Patterns view, the whole song as a grid
`F5`	Open the Sequencer, one pattern up close
`Alt+←/→`	Previous / next pattern
`Alt+↑/↓`	Previous / next bank

4.2 Playing with patterns

The *Patterns* (2.2) view is where a project comes alive. More than one pattern can play at once, so this is where you start them, stop them, and combine them into something larger. One question runs through all of it: when does a change take effect? Cagire gives you two answers.

4.2.1 Acting at once

By default a key fires the moment you press it. Press `Space` and the pattern under the cursor starts or stops. Press `m` to mute it, `x` to solo it. Each takes effect straight away. A launch is not quite instant, though. A starting pattern waits for its own boundary first, by default the top of the next bar, so it falls into time with everything

already running. That boundary is the pattern's quantization. See *Pattern properties* (4.3) to change it.

There is nothing to fall into time with when the sequencer is halted, so launching a clip then does two things at once: it starts the sequencer and brings the clip in with it, on the downbeat, without waiting out a bar first. This is why you never have to start the sequencer before you start a pattern. The same is true of `t` in HOLD: firing an armed scene from a halted sequencer starts everything together.

Falling into time has a quieter side. A pattern started while others play enters in phase, as if it had been running since the sequencer started, so anything that varies from step to step or loop to loop, an alternation, a `bjork` gate, an `every` or a `once`, picks up where the running clock already is instead of counting again from the top. When you want those counts to begin at the beginning, stop and start from the top, or arm a follow action, which always starts its clip from the top.

4.2.2 Building a scene: hold

The other way is to set up several changes and fire them together. Press `h` to turn `HOLD` on. Now `Space`, `m`, and `x` arm a change instead of firing it. Move across the grid and arm as many as you like, launches and mutes and solos all at once. Nothing sounds until you press `t`, and then the whole scene lands together. Instead of hearing each change as you make it, you sculpt the next moment in silence and drop it whole on the downbeat. Press `Esc` to clear an armed scene before it fires, and `h` again to leave HOLD, which also drops anything still armed.

4.2.3 Mute and solo

Mute and solo shape what you hear, in either mode. A muted pattern keeps running but stays silent. A soloed pattern silences every pattern that is not also soloed, so solo is how you lift one idea out of the mix. `M` clears every mute at once, and `X` clears every solo.

4.2.4 Follow actions

A pattern can pass the baton on its own, handing off to another when it finishes a loop. These are follow actions, and you arm them right on the grid. Press `f` on a clip and the follow editor unfolds as a panel at the right of the grid, folding away again when you leave; the cell under the cursor is the one you are arming, and you can walk the grid with the arrows to arm several patterns without leaving.

A clip carrying a follow action does not fall into phase the way a plain loop does. It still waits for its quantization boundary, but it enters on its own first step and counts from there, because the loops that decide the handoff are counted from the launch. That is what makes a chain play its head pattern whole before passing on.

Single letters pick what plays next. `n` and `p` chain to the next or previous filled pattern in the bank, `f` and `l` to the first or last, `a` and `o` to any filled pattern or any other than this one, at random. `r` replays the clip and `s` stops it. For a named target press `j`: the cursor roams the grid as a destination, `Enter` sets the clip you land on, and `Esc` cancels back to the editor.

The four grid directions live on the arrows with `Shift` held, so the same keys that walk the grid also arm a move across it. `Shift+Up` and `Shift+Down` hand off to the bank above or below, which is the only way besides a jump to leave the bank you are in. `Shift+Left` and `Shift+Right` walk the bank itself. A direction crosses empty slots to reach the nearest clip that way, so a blank row between two ideas does not break the chain, and it does not wrap: run out of grid and the clip simply keeps looping. That is what separates them from `n` and `p`, which wrap around their own unbroken run of clips and never cross a gap.

Type a number to set the timing, how long the clip waits before it hands off. `u` switches that count between loops and sixteenth notes, so a follow can fire after four loops or after sixteen sixteenths whatever the loop length. `Backspace` erases a typed digit, or clears the action when nothing is typed, and `Esc` leaves the editor.

Follow actions answer the same question as everything else here. In the immediate default each keystroke lands live, and the clip's badge updates as you arm it. In `HOLD` the change joins the armed scene and fires with the rest on `t`. An armed clip wears a small glyph in its cell: an arrow for the relative targets, a square for stop, a hook for a jump.

Key	What it does
`f`	Open the follow editor on the pattern under the cursor
`n` / `p`	Target the next / previous filled pattern
`Shift+Up` / `Shift+Down`	Target the nearest clip in the bank above / below
`Shift+Left` / `Shift+Right`	Target the nearest clip left / right in this bank
`f` / `l`	Target the first / last filled pattern
`a` / `o`	Target any filled pattern / any other than this one, at random
`r` / `s`	Target play again / stop
`j`	Pick a Jump destination, then `Enter` to set it
`0`-`9`	Type the timing count
`u`	Switch the count between loops and sixteenths
`Backspace`	Erase a typed digit, or clear the follow action when nothing is typed
`Esc`	Leave the follow editor

The keys below run the grid itself.

Key	What it does
`Space`	Start or stop the pattern under the cursor
`m` / `x`	Mute / solo it
`h`	Turn HOLD on or off; in HOLD, Space/m/x arm a scene
`t`	Launch the armed scene
`p` (or `F9`)	Start or stop the sequencer
`P` (or `F10`)	Stop the sequencer and rewind to the top

Key	What it does
`M` / `X`	Clear every mute / every solo
`Esc`	Drop a selection, then clear an armed scene, then step out

See *Patterns* (2.2) for the grid itself, and *Pattern properties* (4.3) for quantization and the rest of a pattern's settings.

4.3 Pattern properties

Every pattern carries a handful of settings of its own. Press `e` in the *Patterns* (2.2) view and Cagire opens its properties. Move between the fields with `↑/↓` or `Tab`, change a value with `←/→`, and type where a name or a number is wanted. `Enter` applies your changes and `Esc` leaves them. In `HOLD` mode `Enter` arms the change instead, to land with the rest of your scene on `t`. The settings are:

Property	What it sets	Values
Name	A label for the pattern	free text
Description	A longer note, shown where you browse and launch	free text
Length	How many steps the pattern loops	1 to 1024
Speed	A multiplier on the project tempo	1/8x up to 8x
Swing	How late the off-beat steps land	50 (straight) to 80
Quantization	The boundary a launch waits for	immediate, beat, or bars

A pattern can also hand off to another when it finishes a loop, but you arm that right on the grid, not here. See *Playing with patterns* (4.2) for follow actions.

Length, speed, and swing have quick keys in the Sequencer when you do not need the full panel. `L` sets the length, `S` sets the speed, and `W` sets the swing. The `S` dialog also takes an exact fraction you type, like `2/3` or `3x`. Swing shuffles the pattern by dragging every second step late: 50 is straight, 66 lands the off-beats in triplet

time, and 80 is the strongest drag. The pattern's first step keeps the boundary it launched to, and the second is the first one dragged. It takes effect immediately, even mid-loop, and the MIDI clock Cagire sends stays perfectly straight while the notes swing. This pattern swing moves the steps themselves, and it is not the `swing` word, which shuffles the subdivisions inside a single step. Tempo is not here, because it belongs to the whole project, not one pattern. You set it with `T`, or nudge it with `+` and `-`. See *Session organization* (4.1) for why tempo is shared and speed is not.

Key	What it does
`e`	Open the pattern's properties, from the Patterns view
`↑`/`↓` or `Tab`	Move between fields
`←`/`→`	Change the focused field
`Enter`	Apply the changes, or arm them in HOLD mode
`Esc`	Close without applying
`L` / `S` / `W`	Set length / speed / swing quickly, from the Sequencer

4.4 Saving and loading

4.4.1 Saving

A song lives in a single file, ending in `.cagire`. Press `s` and Cagire opens a browser to save it. Type a name and press `Enter`, and the file is written; the `.cagire` ending is added for you. There is no autosave. Your work is kept when you ask for it, so save as you go. You can save `.cagire` files anywhere on your computer and retrieve them later. There is no central folder where to put things.

4.4.2 Loading

Loading starts somewhere you might not expect. Press `l` and Cagire opens the Load hub, a shelf of bundled demos and tutorials to learn from and take apart. Switch between the two with the tabs, move through them with the arrows, and press `Enter` to load one. To open a song of your own, press `f` from inside the hub. That swaps to a file browser, where you find your `.cagire` and press `Enter`. The file holds the whole project. Every bank, pattern, and step is

in there, along with your preludes, your tempo, and your notes. It records where your samples came from by path, not the audio itself, so see *Exporting and sharing* (4.6) for how samples travel with a song you hand to someone else.

Your saved snippets are the one thing that does not live in the file. Cagire keeps them with the application so they follow you across every song; see *Snippets* (3.4).

Cagire can also keep up with the file while you work, picking up edits made in your own editor or by someone else without disturbing what is playing. See *Watching the file* (4.5).

4.4.3 New

Starting fresh lives in the same hub. Press `n` from inside it and Cagire stops what is playing and hands you a blank project: empty banks, the default tempo, and your sample folders kept as they are. Like a demo, the blank project has no file until you save it, so the next `s` asks for a name. The command palette reaches the same action under **New**. At launch Cagire still opens on one of its demo songs; turn off *Demo on startup* in the Options view to start empty instead, see *First run* (1.1).

Key	What it does
`s`	Save the song to a `.cagire` file
`l`	Open the Load hub of demos and tutorials
`Tab`	Switch between demos and tutorials, inside the hub
`f`	Open your own songs, from inside the Load hub
`n`	Start a blank project, from inside the Load hub
`w`	Watch the file for outside changes, from inside the Load hub
`Enter`	Confirm the save, or load the selection
`Esc`	Close the hub without loading anything

4.5 Watching the file

A `.cagire` file is plain text, so anything that edits text can edit your song. Cagire can watch the file it has open and pick up changes the moment they land on disk, which means you can keep your

patterns in your own editor, put a song under version control and pull a colleague's commit while it plays, or leave the file in a synced folder and work on it with someone on another machine. What you get back is not a reload. Cagire compares the file against what it already has and applies only the difference, so the transport keeps running, your undo history survives, and patterns you did not touch are never resent.

4.5.1 Turning it on

Watch project file in the Options view is a **Yes** or **No** preference that you can change at any time. When it is **Yes**, Cagire watches the current file whenever the project has one. With a demo, tutorial, or unsaved project, the preference stays on and waits for a file to be opened or saved.

You can also reach it with `w` from inside the Load hub, which is useful when you are already there opening something. Toggling from the hub does not close it, so you can flip the watch and carry on choosing a song; the hub's title says which file it is watching. The command palette reaches the same switch under **Watch**.

Cagire remembers the preference between sessions. A **W** badge sits at the left of the tempo readout while a file is actively being watched, next to where the Fill badge appears, so a watch you left on a week ago is never a surprise.

The watch always follows the current file. Save under a new name and it moves to the new file; load a different song and it moves with you. Loading a demo or starting a blank project pauses the watch because there is no file behind that project. The preference remains **Yes**, so opening or saving a file starts the watch again.

4.5.2 What you see when something changes

Cells the file rewrote light up for about a second and fade back, in the sequencer grid and in the pattern grid both, so a change you did not type has somewhere to be seen. The status line names what arrived: how many patterns, and whether the prelude, the tempo, or the sample folders came with them. A change to a prelude rebuilds

your vocabulary the same way saving the prelude editor does, so words your steps already call take their new meaning immediately.

Your own saves are silent. Cagire recognises the text it just wrote and raises nothing, so the only time the screen moves is when the change came from somewhere else.

One `Ctrl+z` puts back every pattern a reload touched, rather than one of them, because reverting half of somebody else's edit would be worse than not reverting it at all. What came with them stays: a tempo or a prelude that arrived in the same reload is not part of the undo, so set those back yourself if you want them back.

4.5.3 When Cagire refuses

If the file will not parse, nothing at all is applied and the status line says so. A half-written file is never half-loaded.

If the file rewrites the very cell you have open in the editor with unsaved text in it, Cagire keeps what you typed and tells you which step it held back. The rest of the change still lands. If the cell you have open was rewritten and you had *not* typed into it, the editor refreshes under you instead, so what you see is what is stored and pressing `Esc` cannot overwrite the newer text with the older.

4.5.4 Moving around

Key	What it does
←/→	Select **Yes** or **No** on the Watch project file row
`w`	The same switch, from inside the Load hub, without closing it
`Ctrl+z`	Put back every pattern the last reload touched

4.6 Exporting and sharing

There are three ways to get something out of Cagire: exporting a single pattern/bank as compressed text snippets, making an audio recording or exporting the whole project as a `.cagire` file.

4.6.1 Sharing a pattern or a bank

Press `g` from the Sequencer or the Patterns view. Cagire will pack the current pattern into a short text string and put it on your

clipboard. Paste it into a chat and someone else can press `G` to drop it straight into their own pattern slot. A whole bank goes the same way through the command palette, with **Share Bank** to copy and **Import Bank** to paste. The strings are kept small on purpose. The goal is to be able to send small snippets through messaging apps and through emails. Cagire warns you when a pattern grows too long to be shared easily. What travels is the step scripts and their names, not any audio, so a shared step that plays a sample only sounds right if the other person already has that sample under the same name.

4.6.2 Saving a recording

To get audio out as a file, record it in the Sampler. Press `F7`, set the source to the engine's master output and capture a take while the music plays. Cagire records the live output in real time. There is no faster offline bounce of the whole song. Once you have a take, `Ctrl+S` in Edit mode saves it as a `.wav` into one of your sample folders. The Sampler page covers the full recording and editing workflow. Saving recordings can be quite useful if you intend to edit your takes later on in a DAW or any other software. However, please note that Cagire is made for live improvisation and I will likely never support exporting stems or adopting a logic where things are laid down nicely track by track.

4.6.3 Sharing a whole song

A saved song is a single `.cagire` file. To share it, send that one file. The other person opens it from the Load view with `f`. See *Saving and loading* (4.4) for how saving works. The file holds everything you wrote, your banks, scripts, prelude, tempo, and notes, but it does not hold your samples. What it keeps of them is only a trail: the folder paths, never the audio. A song built only from synthesis travels perfectly in that one file. A song that leans on samples needs the samples sent alongside it, and the person opening it has to either put them where the file expects them or keep them in their own sample folder under the same filenames and point Cagire at it.

Cagire scans the folders it is given and matches samples by name, so the names are what matter.

Key	What it does
`g`	Share the current pattern to the clipboard, from the Sequencer or Patterns view
`G`	Import a shared pattern from the clipboard
`:` (**Share/Import Bank**)	Share or import a whole bank
`Ctrl+S`	Save the recorded take as a `.wav`, in the Sampler's Edit mode
`f`	Open a `.cagire` someone sent you, from the Load view

4.7 Taking notes

Every project carries a page of notes of its own. Press `n` from the Sequencer or the Patterns view and the notes open. Notes show up like a small text editor in the middle of the screen. Type as you would anywhere, across as many lines as you like. When you are done, `Esc` folds the panel away and writes the text back into the project. There is no separate save step inside the notes: closing them is saving them. That text lives inside the project itself, not in a file off to the side. It is kept the moment you save the project and comes back with it when you load it again. When a project ships as one of Cagire's bundled demos or tutorials, those notes appear as its description in the Load hub. You can also reach the notes from the command palette. Press `:`, look for **Open Notes**, and Cagire opens the same panel.

Key	What it does
`n`	Open the notes, from the Sequencer or Patterns view
`Esc`	Save the notes and close the panel
`:` then **Open Notes**	Open the notes from the command palette

While the notes are open every key goes to the text, so write freely; `Esc` is always the way back out.

Chapter 5

Sound resources

5.1 Loading samples

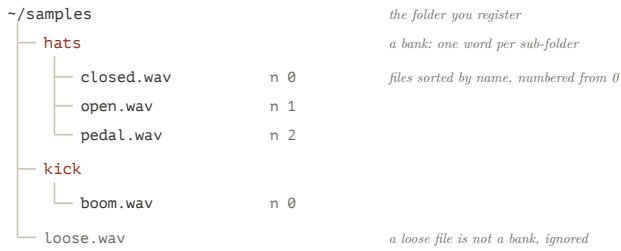


Figure 19: The layout Cagire expects: each sub-folder of a registered folder is a bank, its files sorted and numbered from zero.

An audio sample is just an audio file on your disk and Cagire can play any audio file you have (`.mp3`, `.wav`, etc). Before that can happen, Cagire has to know where your samples live to index that space. You register one or more folders, Cagire scans them, and every sub-folder of samples it finds becomes a word you can write into a pattern. Samples are lazily loaded: they are only loaded in memory when they are really required. This page is about that path, from a file on disk to a word in a script.

5.1.1 Registering a folder

Cagire does not go looking through your whole machine. You point it at the folders you want on the Engine view's Sample tab. Press `F6` to open the Engine, `Tab` to the Sample tab, and `A` to add a folder. A small browser opens for you to pick a directory, and the count of audio files it holds is shown as you go. The folder joins the list and is remembered with your settings, so it is there the next time you start. `D` removes the folder under the cursor from the list

again. Removing it here only forgets the path; nothing on disk is touched. You can register as many folders as you like, and they are scanned the moment you add them, so a sound is ready to play right away.

5.1.2 What Cagire loads

Cagire reads the common audio formats: WAV, MP3, OGG, FLAC, AAC, and M4A. Anything else is ignored. Samples come in banks, and a bank is simply a sub-folder of a folder you have registered. The bank takes the sub-folder's name, so a `hats` folder becomes the word `hats`. Cagire sorts the files inside by name and numbers them from zero, so three files in `hats` give you slots `0` to `2`, and `n` is how you pick between them. Cagire looks one level down, at the sub-folders of a registered folder and no deeper, so a loose file sitting straight in a registered folder is not a bank. Tuck your samples into a sub-folder and that folder becomes the word you play.

5.1.3 In a script

A bank is a sound source like any other: name it with `snd`, play it with `.`, and pick between its files with `n`, where index `0` is the first. The value comes first and the word after, the same postfix order as the rest of the language. See *Selecting samples* (5.1) for choosing and shaping which file in a bank plays.

5.1.4 Keeping the library in order

You do not have to leave Cagire to tend your sample library. The *Samples* (2.7) view is built for it. There you browse the registered folders, audition any file by pressing it, rename, duplicate, and delete, and search the whole library by name. The folders you add on the Engine's Sample tab are the same ones the Samples view shows.

5.2 Recording samples

New samples can be recorded in the Sampler. The sampler view captures sound coming from various sources. It lets you trim the take down to the part you want and save it into your library. Press `F7`

to open it from anywhere. The Sampler has two tabs, `Tab` switches between them:

- **Record**: where you arm a source and capture.
- **Edit**: where the take lands, and where you trim, reshape, and save it.

5.2.1 Choosing what to capture

Before a take exists, the Record tab is a small arming screen. The arrow keys (←/→) pick the source. Cagire offers the front stereo pair, each input channel on its own as mono, a mixdown of them all, and the engine's own master output. The sampler can bottle up what Cagire is playing just as readily as what comes in through a microphone or instrument. The meter follows whatever source you pick, so you can set your levels before you commit.

You can also skip capture entirely and load an existing audio file directly into Edit from the arming screen.

Key	What it does
←/→	Pick the source to capture
`l`	Load an existing audio file directly into Edit

5.2.2 Free and quantized

↑/↓ on the arming screen switch between two ways of recording, and the choice changes what the length control does. In free mode the take starts the instant you press `r` and runs until you press `r` again. `[` and `]` set a safety length the take will not exceed, stepping in seconds up to a ten-minute ceiling.

In quantized mode the take snaps to a bar grid. Start the transport first, then `r`: Cagire waits for the next bar, captures a whole number of bars, and stops on its own. `[` and `]` cycle that length through 1, 2, 4, 8, and 16 bars. A second `r` cancels a pending take or stops a running one early. If the transport is not playing, Cagire reminds you to start it.

Key	What it does
↑/↓	Switch between free and quantized recording

Key	What it does
`r`	Start the take, then stop it (or cancel a pending quantized take)
`[ / ]`	Set the length: a seconds cap in free mode, or the bar count (1, 2, 4, 8, 16) when quantized

5.2.3 Trimming the take

When the capture ends, Cagire hands the take to the Edit tab and puts you there, so the arming screen never has to double as a waveform editor. The waveform fills the page and the arrows now move the trim markers. `s` and `e` pick which marker you are moving, the start or the end, and ←/→ nudge it. `z` toggles snapping to the nearest zero crossing, which keeps the cut edges from clicking. `+` and `-` zoom in and out for a closer look. `g` followed by a letter jumps the marker straight to a labelled zone of the waveform. `Space` plays just the region between the markers so you can hear the trim before you keep it. `Esc` throws the take away and returns you to the arming screen, asking first.

Key	What it does
`s / e`	Choose the marker you move, the start or the end
←/→	Nudge the chosen marker
`z`	Toggle snapping to the nearest zero crossing
`+ / -`	Zoom in and out
`g` then a letter	Jump the marker to a labelled zone
`Space`	Play just the region between the markers
`Esc`	Discard the take and return to the arming screen, after you confirm

5.2.4 Saving

The trimmed take is saved from the Edit tab, where it already sits. `Ctrl+S` opens a browser to save the selected region as one WAV file. For slices, press `p` and choose either the equal-grid or zero-crossing export process; Cagire writes the result into a new sample-bank folder. Both paths rescan the sample library after writing, so the result plays like any other sample. Reshaping and slice export are covered in *Editing samples* (5.3).

Key	What it does
`Ctrl+S`	Save the trimmed take to a library

5.2.5 From a script

The Sampler captures into a buffer you trim by hand. To record straight from Forth instead, the `rec` family bottles up the engine's own output into a sample as it plays: `"name" rec` starts a take and `endrec` stops it, `dub` overdubs onto an existing one, and `orec` captures a single orbit. See *Live recording* (5.6) in the engine docs for the full set.

5.3 Editing samples

The Sampler's Edit tab is where a take becomes the sound you want. It gathers a set of destructive audio tools behind a single picker that acts on the take in place. A finished take lands here on its own, `Tab` reaches it from the Record tab, and `l` loads an existing sample from your library to work on. Editing, thus, can also be a way to work on your existing samples.

Every tool works on the selected region, and not always the whole sound. If you have set the start and end markers, the edit applies between them. With no selection, it applies to the entire take. Place the markers around the part you mean to change, then reach for the tool.

Marking that region is Edit's own work, so the marker keys live here alongside the tools. `s` and `e` name the marker you are moving, `←/→` nudge it, `g` followed by a letter jumps it to a labelled zone, `+/-` zoom in for a closer cut, `↑/↓` exaggerate the peaks, and `z` snaps to zero crossings.

Key	What it does
`s / e`	Choose the marker you move, the start or the end
`←/→`	Nudge the active marker
`g` then a letter	Jump the active marker to a labelled zone
`+ / -`	Zoom in and out

Key	What it does
↑ / ↓	Exaggerate the peaks, or settle them back
z	Snap markers to zero crossings, or stop

5.3.1 The tools

Every edit and export sits behind one key. `p` opens the process picker, a list of everything Edit can do with a take. Type to narrow it, move with the arrows, and `Enter` runs the highlighted process on your region. Gain and resize ask for a value; slice exports ask for a grid size and destination folder.

Key	What it does
p	Open the process picker
u	Undo the last edit, stepping back through several

Process	What it does
Normalize	Lift the loudest peak in the region to full scale
Remove DC offset	Centre each channel's waveform on zero within the region
Fade in / Fade out	A linear fade up or down across the region
Reverse	Play the region's frames backwards
Invert phase	Flip the polarity of the region
Swap L/R	Swap the stereo channels, across the whole take
Fold to mono	Average the channels down to mono, across the whole take
Silence	Zero the region
Crop to region	Keep only the region, throwing away everything outside it
Delete region	Cut the region out and splice the ends back together
Gain… (dB)	Scale the region by a number of decibels you type
Resize region to length…	Trim the region to a length you type, like `2s` or `4 bars`

Process	What it does
Export equal slice grid…	Write the region as equal-length WAV slices
Export zero-crossing slices…	Start from an equal grid and move cuts to local zero crossings

Destructive tools can be taken back. `u` undoes the last edit, and you can step back through several. Slice exports leave the take unchanged.

5.3.2 Hearing your work

The region between your markers is the region your tools act on, and it is also what you hear. `Space` plays the trimmed region as it stands, with every edit so far baked in, so you can judge the result before you commit it. A second press stops the playback.

Key	What it does
`Space`	Play the trimmed region, with every edit so far baked in

5.3.3 Keeping it

When the sound is right, `Ctrl+S` opens a browser to save the selected region as one file in your library. The process picker also exports the region as a folder of slices. `l` loads an existing file directly into Edit from either Sampler tab, so recording a take first is never required.

Key	What it does
`Ctrl+S`	Save the sound into a folder in your library
`l`	Load an existing file from either Sampler tab

5.3.4 Exporting a slice grid

Slice export divides the region between the start and end markers. Press `p` and choose **export equal slice grid...** to keep every slice the same length, or **export zero-crossing slices...** to begin with the same grid and move each internal cut to the nearest local zero crossing. Then choose 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, or 64 slices.

After choosing the method and size, name a new or empty folder in your sample library. Cagire writes `slice-00.wav`, `slice-01.wav`, and so on, then rescans the library. The folder becomes a sample bank immediately, with the file order matching `n 0`, `n 1`, and the following indices. A non-empty destination is left untouched.

5.4 Loading soundfonts



Figure 20: An .sf2 dropped into a registered sample folder replaces the bank Cagire ships with. The first one found wins, and only one plays at a time.

A soundfont is a single file that holds a whole set of instrument sounds, a piano, strings, a drum kit, and the rest, each one built from recorded samples and ready to play. It is an old school format that was used quite a lot in the past. The file ends in `.sf2` and follows the General MIDI layout, so a given instrument always sits at the same number.

Cagire carries one already. GeneralUser GS, by S. Christian Collins, is built into the binary, so the `gm` words sound the moment you start Cagire. There is nothing to download, nothing to install, and nothing to point Cagire at.

5.4.1 Using your own

Drop an `.sf2` file into one of your registered sample folders and Cagire plays that one instead. It looks when the software starts and again the moment you register a new folder, so a soundfont you have just added is ready without a restart. Only one is ever active. If several of your folders hold one, the folder you registered first wins. See *Loading samples* (5.1) for registering folders in the first place.

The built-in bank lives inside the binary rather than in a file, so there is nothing on disk to edit or delete. Replacing it means giving Cagire a soundfont of your own. Take that file back out of your sample folders and the built-in bank returns.

A file Cagire cannot read falls back to the built-in bank, so a truncated or half-copied `.sf2` leaves you with working instruments rather than silence.

5.4.2 Playing it

You name an instrument by gluing `gm` to its name or to its General MIDI number from 0 to 127, like `gmpiano` or `gm48`. Bare `gm` is the piano:

```
gmpiano snd .
```

The drum kit lives on its own bank, under `gmdrums` or `gmpercussion`, where each note is a different piece:

```
gmdrums snd 36 note .
```

See *Soundfonts* (2.9) in the engine docs for playing GM in full. See the *General MIDI presets* (B) appendix for the table of instrument names.

Chapter 6

Configuration

6.1 Audio input

The **Input** column of the **Audio** tab (`F6`) chooses the device sound comes in from. It takes whatever your driver offers: a microphone, a line input, an audio interface. It sits beside the Output column and shares the same driver. The backend you pick for playback is the one Cagire records through. ←/→ move focus between the Output and Input columns. ↑/↓ move the cursor within Input. `Enter` selects. A freshly picked input takes hold the next time the audio engine restarts. Press `R` to bring it online. As with the output, the choice is remembered between sessions. What flows in the audio input feeds the **Sampler** (`F7`). Set the Sampler's source to the input, and a recording captures whatever this device hears into a working take.

How many channels Cagire opens is your decision. **In Channels** in the Settings list sets it and defaults to two. A four-output synthesizer or a multi-input interface needs that number raised before Cagire can see past the first pair, because a driver asked for nothing in particular hands back a stereo pair every time, whatever the hardware carries. Raise it, press `R`, and the rest of the channels arrive. From there `inchan` picks between them inside the language, so the setting decides how many channels exist and the language decides what listens to which (see *Live audio* (2.8)).

Cagire asks for the width you set and reports what it was given. When the two differ the Metrics column says so in red, reading *asked 8, got 4*, and the setting keeps the number you chose rather than quietly rewriting itself to the smaller one. Your request outlives the device that refused it, so swapping a stereo interface for a wider one

needs no retyping. Output channels and the sample rate work the same way.

One important note: If the input device runs at a different sample rate than the output, the Metrics column flags it in red as a *rate mismatch*. Cagire keeps working, but matching the two rates is the cleaner setup. Reconcile them in your driver or interface when you can, or move the output side to meet the input with the **Sample Rate** setting (see *Audio output* (6.2)). The key list is under `?` as always. See *Audio output* (6.2) for the playback side, and *Sound resources* (5) for recording and editing what you capture.

6.2 Audio output

Cagire needs a real audio device to make sound. The **Audio** tab of the Engine view (`F6`) is where you set the audio output and the settings that shape the stream.

6.2.1 Driver and device

The **Driver** is the audio backend Cagire talks to. Many machines have only one worth using. Where there is a choice, ←/→ step through them. Some drivers manage the buffer themselves. When one does, the Buffer reads *Host managed* and its size leaves your hands. Below the driver, the **Output** column lists every device that driver offers. ↑/↓ move the cursor. `Enter` selects. The choice is remembered the next time you open Cagire.

6.2.2 Settings

Setting	Default	Range	Step
Out Channels	2	2–128	1
In Channels	2	1–128	1
Sample Rate	Device	Device to 192000	one rate per press
Patching	Auto	Auto or Manual	toggles
Buffer	512	64–4096	doubles / halves
DSP Block	32	1–256	doubles / halves
Voices	64	1–1024	1

Setting	Default	Range	Step
Audio Nudge	0 ms	±200 ms	1 ms (10 with Shift)
Master	0.0 dB	OFF to +6 dB	0.5 dB
Bass mono	off	off to 300 Hz	10 Hz
Limiter	On	On or Bypass	toggles

- **Out Channels**: how many output channels Cagire asks the device for. Two is one stereo pair, eight is four pairs, and it is the pair count that decides how orbits spread across a rig (see *Multichannel output* (9.1)). The floor is two because the engine places voices onto pairs.
- **In Channels**: the same request on the capture side. If either count comes back smaller than you asked, the Metrics column says so and the setting keeps your request (see *Audio input* (6.1)).
- **Sample Rate**: the rate Cagire asks the output device to run at. *Device* is the default and takes whatever the device already runs at, which is the setting to leave alone unless you have a reason. Naming a rate is worth it when the device would otherwise sit at 44100 and your material is 48000, since Cagire does not resample the live input and a mismatch there drifts.
- **Patching**: whether Cagire's ports arrive connected. *Auto* lets the session manager wire them to the system devices, which is what you want on a laptop. *Manual* registers the ports and connects nothing, leaving the routing to you in a patchbay like qpwgraph, Carla or qjackctl. It reads *Auto (host)* and cannot be changed on drivers with no such switch, which is everything except PipeWire and JACK on Linux.
- **Buffer**: the buffer size in samples. The tab shows the latency each size works out to at the current sample rate.
- **DSP Block**: the block size Cagire's sound engine processes at a time.
- **Voices**: the ceiling on notes sounding at once.
- **Audio Nudge**: shifts all of Cagire's audio earlier or later. It moves the sound itself. The MIDI and OSC nudges shift those

streams separately. Leave it at zero unless you mean to slide the whole engine against something external.

- **Master**: the output level for everything Cagire makes. It sits before the master limiter, so turning up drives the safety chain rather than escaping it.
- **Bass mono**: collapses everything below this corner to the centre, off by default. It is what stops a stereo low end from cancelling in a summed sub array.
- **Limiter**: the master ceiling. `Bypass` takes it out of the chain whole, which also takes out the safety clip behind it, so the output can then leave Cagire past full scale. Leave it on unless you are feeding something that does its own limiting.

The last three belong to the machine rather than to the music, so they are saved with your audio settings and never travel in a project file. *Master output* (9.2) covers what they do to the sound and how to read the limiter while you set them.

6.2.3 When changes take hold

Only a driver change restarts the engine on the spot. The device, both channel counts, sample rate, patching, buffer, DSP block, and voice count are staged as you set them and take hold on the next restart. Dial them in, then press `R`. The nudge, the master level, bass mono and the limiter are the exceptions: they move the moment you turn them.

6.2.4 Stage controls

The footer carries the controls for those restarts and for when a device changes underneath you.

Key	What it does
`R`	Restart the audio stream
`t`	Play a short test tone, to confirm sound is getting out
`h`	Hush everything at once
`p`	Panic, cut any stuck voices
`D`	Rescan the device list when you plug something in or pull it out
`r`	Reset the peak voice count once you have read it

`?` lists this tab's keys. See *Audio input* (6.1) for the capture side, *Synchronization* (6.3) for keeping the engine in time, and *Master dynamics* (9.3) for the compressor on the `DYNAMICS` tab, which runs ahead of everything described here.

6.3 Synchronization

Cagire rarely plays alone. The **Clock** tab of the Engine view (`F6`) keeps it in time with other software and hardware through **Ableton Link**. It is the industry standard protocol that lets programs on the same network share one tempo and a common beat grid. Turn it on and Cagire joins the session as a peer. When anyone changes the tempo, everyone follows, and all of them agree on where the beat falls. The tab holds three settings, with `↑`/`↓` moving between them and `←`/`→` changing the one in focus:

- **Enabled**: turns Link on or off. With Link off the header reads `DISABLED`; once on, it reads `LISTENING` until another peer appears and `CONNECTED` once one does. The metrics column counts how many peers are in the session.
- **Start/Stop**: ties transport to the session as well. Starting Cagire playback starts the others and stopping it stops them, and the other way round.
- **Quantum**: is the bar length (in beats) that Cagire phase-aligns to. It is `4` by default and adjustable from `1` to `16`. It decides how playback snaps into the shared grid when you start.

With Start/Stop on, Cagire starts up halted rather than playing, and waits for you to press play. Starting the transport is shared, so booting into playback would drag every peer in the session along with it, and joining a jam should never do that to the people already in it.

Link shares a tempo and a beat grid, never a song position, so a peer stopping the session stops Cagire and rewinds it to the top. On the next shared start everyone relaunches together from the downbeat, in phase. There is no held position to keep, so a stop is always the same fresh rewind, whoever calls it: a peer, `F9`, `F10`, or the stop button.

Tempo itself is not set here. You set it from the main view, where the BPM readout and a tap-tempo entry live. The Clock tab shows it back to you live, alongside the current **Beat** and **Phase**, so you can watch Cagire ride the grid. When the session is connected, moving the tempo anywhere moves it everywhere. If the gear you want to sync does not speak Link, the **MIDI** tab can send clock and transport instead.

6.4 Sample folders

A script plays samples by identifying samples through the folder they belong to. For instance, you can have a folder named `mykicks` full of kick samples, and that is the name that Cagire will know about. The **Sample** tab of the Engine view (`F6`) is where you build or re-organize that library of folders containing audio files. It is materialized as a list of root folders Cagire scans for audio. The header counts how many folders you have given it and how many files it found inside them.

Press `A` to open a browser and add a folder; `D` removes the one under the cursor. A folder you add is scanned and ready at once. No restart needed, and your list of folders is remembered between sessions. You can also point Cagire at folders when you launch it, with the `-s` flag, repeated as many times as you like:

```
cagire -s ~/samples -s ~/drums
```

How Cagire turns those folders into playable banks, scanning one level deep and numbering the files inside, is covered in *Loading samples* (5.1). What matters here is that adding a folder only indexes it: scanning records the files without decoding them in full. Cagire reads a short head of each sample ahead of time so the first hit plays without a gap, but the body of a file stays on disk until you actually reach for that sound, so you can register a library of thousands and spend only a sliver of memory until you play from it.

6.5 MIDI (I/O)

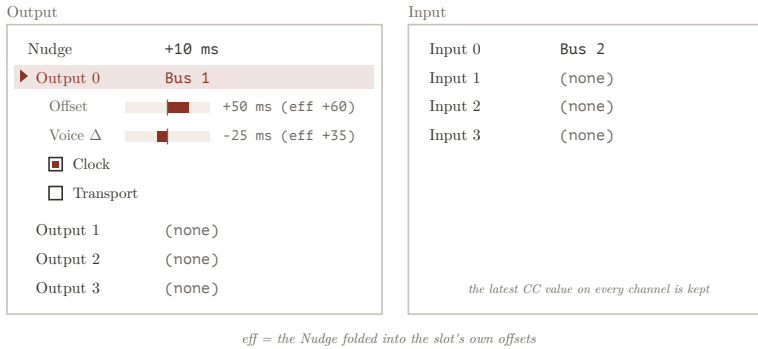


Figure 21: The MIDI tab: the Nudge row over the four Output slots and the four Input rows, with a connected slot's timing and clock controls open.

Cagire can speak MIDI to your gear/software over four dedicated output slots. It can also listen back over four dedicated input slots. These are wired up on the **MIDI** tab of the Engine view (`F6`). A slot is only a physical or virtual MIDI port. Choosing one in this view connects it. The words that actually send and read MIDI live in the engine reference. All settings will be remembered between sessions.

6.5.1 Connect to MIDI

Each of the four **Output** rows points at one MIDI port. ←/→ cycle through the ports your system offers. A given port can be held by only one slot at a time. Once a slot is connected it gains two timing controls. The **Offset** shifts everything that slot sends, for trimming one device that runs early or late. **Voice Δ** shifts only the musical messages (notes, control changes, bend, pressure, program changes) and leaves clock and transport alone, so MIDI-triggered voices can be seated against the audio without dragging the clock with them. Both adjust with ←/→ by a millisecond at a time, or ten with `Shift`, up to ±200 ms. Each row shows the effective figure once the section nudge is folded in. That section-wide **Nudge**, on the row above the slots, shifts every MIDI event at once against the audio; the per-slot offsets stack on top of it.

A connected slot also carries a **Clock** and a **Transport** checkbox. Tick **Clock** and Cagire sends MIDI clock, 24 pulses per quarter note, locked to the tempo while it plays. Tick **Transport** and it sends start, stop, and continue on the matching edges. Both are opt-in per slot, so you can clock several devices independently.

6.5.2 Listen to MIDI

The four **Input** rows work the same way in reverse, each picking a port to read from. Cagire keeps the latest control-change value on every channel of every input. What you do with those values belongs to the engine reference. Read these sections to learn more about it.

A control change can also drive a sound while it plays. The word `cc>bus` routes one controller on one input to a control bus, a number the audio engine holds and any parameter can follow. That routing is written in a script, usually a prelude, and not in this tab (*Control buses* (3.18.6)).

6.5.3 Panic and note-off

When playback stops, Cagire sends All Notes Off on every channel of all four outputs, and `p` (Panic) does the same on demand, so a note left ringing on external gear is always released.

Every key this tab answers to is one `?` away. See *OSC (I/O)* (6.6) for the software side and *Synchronization* (6.3) for clocking over Ableton Link.

6.6 OSC (I/O)

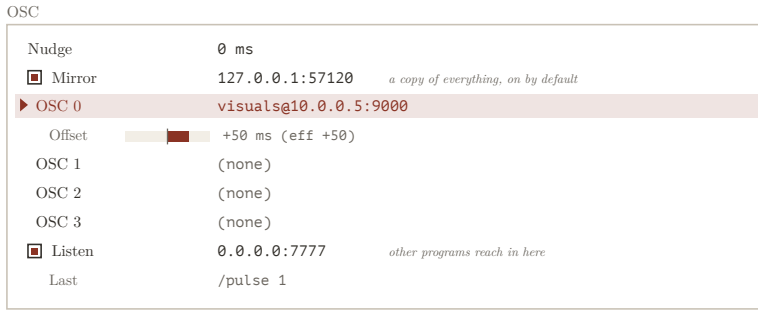


Figure 22: The OSC tab: the Nudge row, the Mirror slot, the four endpoint slots, and the Listen row with its Last line.

Cagire sends and receives OSC. Check out the **OSC** tab of the Engine view (`F6`). You can have up to four output slots, a mirror slot, and a listener. As with MIDI, a slot is only a destination. For the words that emit and read OSC, head to the engine reference. In this section, we teach you how to set the destinations up and what you set is remembered between sessions.

6.6.1 Configuring a port

Each **OSC** row holds an endpoint, written `name@host:port` (or just `host:port` if you leave it unnamed). `Enter` opens an editor to type or change it. Once a slot points somewhere it gains an **Offset**: ←/→ walk it in 1 ms increments, `Shift` jumps by 10, and ±200 ms is the ceiling. The **Nudge** row at the top is the global lever, moving all outgoing OSC against the audio in one motion, with each slot's own offset added on top.

6.6.2 Mirror slot

The **Mirror** row forwards a copy of everything Cagire plays: every sound and every MIDI or OSC emit, as a `/cagire/mirror` packet carrying the same parameters. The one thing it cannot carry is a motion you composed yourself: the packet reports it as `<motion>`, because the notation has no spelling for it (see *Motion notation* (3.18.3)). The engine still plays the real curve, and a

motion the notation can write is mirrored as itself. It is on out of the box, pointed at `127.0.0.1:57120`. This makes it easy to feed another application like SuperCollider or a message recorder running alongside. `Space` toggles it and `Enter` edits where it points.

6.6.3 Listen to incoming OSC

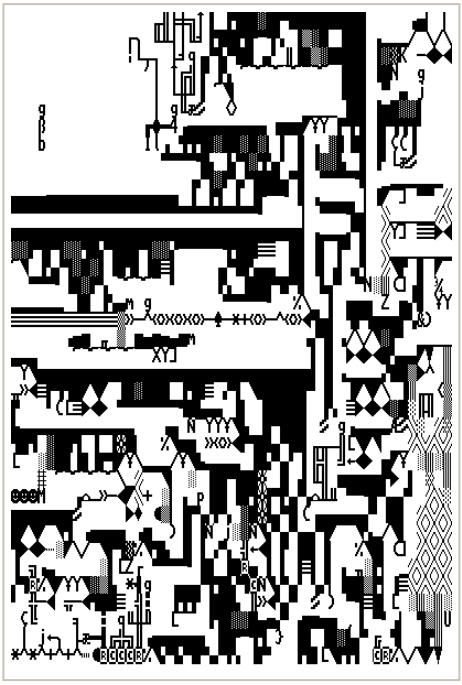
The **Listen** row binds an incoming port so other programs can reach Cagire. It starts off; `Space` turns it on, binding to `0.0.0.0` on port 7777 so packets arrive from any interface on the machine. `Enter` sets the port. A **Last** line beneath the row shows the most recent message that came in. That way, you can visually confirm something is arriving before you wire it into a patch.

An argument of an incoming message can also drive a sound while it plays. The word `osc>bus` routes one argument at one address to a control bus, a number the audio engine holds and any parameter can follow. That routing is written in a script, usually a prelude, and not in this tab (*Control buses* (3.18.6)).

As everywhere, `?` shows the keys. *MIDI (I/O)* (6.5) is the hardware twin of this tab, and *Synchronization* (6.3) keeps everything in time.

Part III

The language



Chapter 1

Introduction

Cagire is programmed in a small concatenative language of the Forth family. It is built for one purpose: describing sound and revising the description while it plays. A program is a flat sequence of words separated by whitespace. Cagire evaluates them from left to right. Each word is a function from stack to stack: it takes the values it needs from the top of a shared data stack and leaves its results in their place. Writing two words side by side composes them. That is what *concatenative* means. Arguments come before the words that use them, so addition is written `1 2 +`. There is no operator precedence, no grouping, and no statement syntax to track. How this dialect departs from classic Forth is the subject of *Is Cagire a Forth?* (A) in the appendices.

```
sine snd c4 note .
```

Four words, one sound. `sine` is a bare source name. A word Cagire does not recognise becomes a string, and this is how names travel to the words that consume them. `snd` takes that string and opens the sound register, a record separate from the stack that holds the description of one sound. `c4` pushes a pitch and `note` stores it in the register. `.` emits: the register is read out to the audio engine, which builds a voice from it. This is the semantic core of the language. A program's result is not text but a sound description. Parameter words accumulate in any order, and nothing sounds until you emit. One script can fill and fire the register as many times as it likes.

Two further mechanisms carry most of the rest. Words live in a dictionary, and `: name ... ;` adds your own, used exactly like the built-ins. Code is itself a value: `( ... )` pushes a quotation, a deferred program that the combinators run, repeat, or choose

between. Most of the language's control flow is built this way. The values that ride the stack form a small closed set: integers, floats, strings, quotations, groups of values, and a few musical kinds. They are typed dynamically, and a word checks its arguments at the moment it executes.

The last thing to know is that a program rarely runs once. A line evaluated from the editor runs when you fire it. The same line placed in the sequencer runs again on every pass of the playhead, hundreds of times in a minute of playing. Read-only context words such as `step`, `runs`, and `iter` push the current position in time, so a fixed script can answer differently on every run. The code stands still, the moment moves, and the sound evolves. This is the live model, and it is why the language is shaped the way it is. Postfix lines are short to write and safe to edit mid-performance. This part of the documentation has three layers:

- **Basic concepts** is the foundation and reads in order. It starts by making a sound, then takes that line apart: the stack, what counts as a word, the values that ride it, the sound register you fill and fire, the dictionary of words, the quotations and combinators that give the language its shape, the live model, variables, and how errors are reported.
- **Language reference** documents the vocabulary family by family, for looking a word up once you know your way around: lexical structure, stack and math, lists, context, control flow, combinators, definitions, selection and cycling, randomness, timing, generators, patterns, pitch and harmony, modulation, and the MIDI, OSC, and recording outputs.
- **Your own language** covers extending the dictionary and keeping what you build: your user dictionary and the preludes that load with a bank or with Cagire itself.

The words that shape the sound itself are documented in *The engine* (IV): the oscillators, filters, and effects behind names like `lpf` and `verb`. New to all of this? Start at *First sounds* (2.1) and read *Basic concepts* (2) straight through. Looking for a single word? Open the *Dictionary* (2.1) or jump to its family in the reference.

Chapter 2

Basic concepts

2.1 First sounds

The best way to start with Cagire is to play something, even if you don't understand the syntax yet. You do not need to understand how the language works in detail to make interesting music. Knowledge can be acquired on the way. Type a line, evaluate it, and listen. Everything else in this chapter will slowly explain every detail there is to know. Start with a single tone:

```
sine snd .
```

`sine` is the name of a sound source. `snd` (or `sound` for the long form) starts a sound from it. The dot at the end (`.`) is a word whose function is to play that sound you just defined. You should now hear a plain sine wave. Remove `.` and you get a valid program that describes a sound but never plays it. We have used `sine` which is the name of an oscillator. We haven't specified any pitch for it. Cagire will assume that you intended to use a default frequency. The same goes for the length of the tone you just played or for its volume. By default, Cagire tries to infer how to form a complete sound event out of the information you provide it with. You can start with something simple and refine along the way by playing. Let's add a `note` now:

```
sine snd
c4 note
.
```

`c4` is the middle C note. Adding this parameter will make the sine play a middle C. Read this program line by line, each line from left to right. It names the source, sets the note, and fires it. Change the source and you will change the sound timbre:

```
saw snd
c4 note
.
```

A sawtooth at the same pitch, brighter and buzzier than the sine because it is full of harmonics. Any source word slots into the same place. Most of what shapes a sound is a parameter you add before the `.`. Let's go further by adding a lowpass filter that rounds off the top of the saw:

```
saw snd
c4 note
800 lpf
.
```

`800 lpf` sets the cutoff frequency of our lowpass filter to 800 Hz. The tone comes out softer. You can keep stacking parameters this way, one value and one word at a time. Give `note` several pitches at once and they all play together:

```
c4 e4 g4 note
sine snd
.
```

Three notes, one chord: a C major triad with one note per sine oscillator. You did not write a loop or a list, you just put three values where one would go. Cagire is quite simple to pick up, at least for basic operations and sounds. You will add more complexity when you feel ready! Like any other musical instrument, it takes time to learn how to use it and to know what you can get out of it.

2.1.1 Reading a line

Every line you have written follows one shape: values first, then the words that use them. `c4` is a value. `note` is a word that takes that value. The next page, *The stack* (2.2), explains why this postfix order is all there is to it. Not every line makes sound, though. Some just compute:

```
2 3 + print
```

This adds two numbers and prints `5`. `print` shows a value on screen; `.` plays the sound register. They are the two ways a line can tell you what it did. They are easy to mix up at first, so keep them apart: `.` you hear, `print` you read.

2.2 The stack

A stack is a pile of values: you add to the top and take from the top. The last value you put down is the first one back. In Cagire, every value flows through one such stack. Every word takes its inputs from the top and pushes its results back. You write words from left to right in the order they run. The data comes first, the word that uses it comes after. That is what postfix and stack-based mean. The stack is a pretty classic data structure in computer science. There are a lot of languages built around the idea of *manipulating the stack* as a first-class operation. Cagire is one of them.

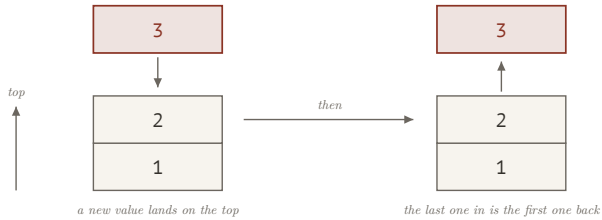


Figure 23: Values land on the top and are taken from the top, so the last one down is the first one back.

```
5 dup + print
```

`5` goes on the stack. `dup` copies it so there are two fives. `+` adds them. `print` shows the result: `10`. That line is the whole grammar of Cagire in miniature: values first, then the words that consume them. The rest of the language is the same shape. Each word is described in the documentation / dictionary by its effect on the stack. It is written as such: `(before -- after)`. The names on the left are what the word takes off the top. The names on the right are what it leaves. `+` is `(a b -- sum)` and `dup` is `(a -- a a)`. A stack

effect is enough to see whether two words will fit together, no trial run needed.

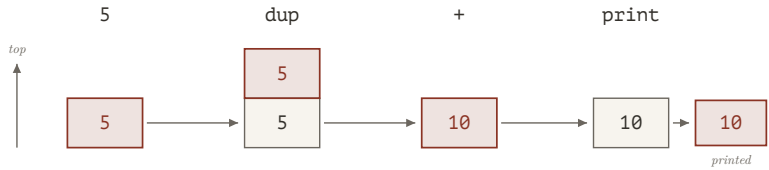


Figure 24: Each column is the stack after that word runs in 5 dup + print; print shows 10 and leaves the stack unchanged.

If you are already familiar with programming, the stack will look strange at first: no variables to name, no arguments in parentheses, no precedence to remember. What replaces all of that is order. Values go down, words pick them up, and a line reads as the sequence of things that happen. It is a small idea with a surprising amount of reach.

2.2.1 A program is one evaluation

When you evaluate, Cagire takes the whole script at once and reads it as a single program. New lines are not special, they are just whitespace. A step written over several lines runs as one continuous sequence of words. We could rewrite the program we just saw as:

```
5
dup
+
print
```

The stack starts empty at the start of each evaluation. Whatever the program leaves on it is cleared before the next run. Within a single run, a value flows from one word to the next, and from one line to the next, in the order you wrote. Take a look at this example:

```
1 2 3 print
```

This leaves `1`, `2`, `3` with `3` on top, and `print` shows the top: `3`. Evaluate it again and you get `3` again, never `3 3`, because each evaluation begins from empty. Because lines are not boundaries, a real step script usually spans several lines while building one sound:

```
saw snd
c4 note
0.3 verb
.
```

The four lines are one program: `saw snd` opens the register (see *The sound register* (2.5)). The next two add a pitch and some reverb. `.` finishes the work by firing the program. Written on a single line, it would behave exactly the same. Let's prove it by evaluating the next example:

```
saw snd c4 note 0.3 verb .
```

2.2.2 Moving things around

A handful of words do nothing but rearrange the top of the stack. They are here to help you manipulate the stack and perform useful common operations. You will not need them all the time, but they are there to make your life easier. Some of them can also be used to great effect when making music. `swap` is a good introduction to such words:

```
1 2 swap print
```

`swap` exchanges the top two values on the stack. `1` is now on top and this program prints `1`. Alongside `swap` are `over`, `dup`, `drop`, `rot`, `nip`, `tuck`, and the double-width versions like `2dup`. A second group acts on a whole list of values at once, so `[ 1 2 3 ] sum` adds the three together. All of them, with their stack effects, are gathered in *Stack & math words* (3.3).

2.2.3 Arithmetic is postfix too

The arithmetic words follow the same rule: they take their operands off the stack and leave the result. `+ - * /` each take two numbers. Order matters for the ones that are not symmetric, since the deeper value is the left operand.

```
5 3 - print
```

`5` is under `3`, so this is `5 - 3` and prints `2`. A result that lands on a whole number comes back as an integer. Otherwise it stays a float.

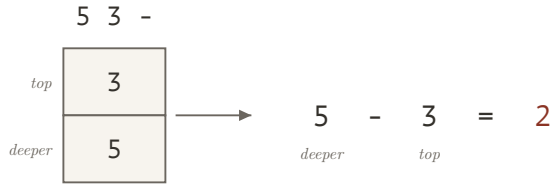


Figure 25: In 5 3 -, the deeper value 5 is the left operand and 3 the right, so the line means 5 - 3 = 2.

To steer a program by putting blocks of code on the stack and running them, see *Quotations & combinators* (2.7). To turn several values into several voices at once, see *The sound register* (2.5).

2.3 Words & comments

Cagire reads your source code as a stream of words separated by whitespace. This section teaches you how that reading works. What counts as a word? What happens to a word Cagire does not know? How to use comments to ignore parts of the source code?

2.3.1 What is a word?

A word is the smallest piece of a Cagire program that means something on its own. Cagire belongs to the Forth family of languages, and in a Forth everything is a word: a number is a word, a name is a word, an operation is a word, and every definition you write becomes a new word too. There is almost no grammar beyond this. You do not write statements or expressions, you write words, one after another, and Cagire reads them from left to right and acts on each one in turn. Some words are already known and do something the moment they are read, others are names Cagire has never seen and become strings, and the rest are numbers. Learning Cagire is mostly learning words and the order you put them in.

2.3.2 Spacing

Words are separated by whitespace and nothing else. The whitespace is the most useful token to know about in Cagire. Spacing is not optional. Everything that is not a whitespace has some value, and whitespaces are delimiters. Line breaks also count as a whitespace and using line breaks is a great way to make your program more readable to other people but also to yourself.

2.3.3 Unknown words become strings

A word Cagire does not recognise becomes a string. Using an unknown word does not error, with the one exception described below, which is highly unusual for a programming language. You will see that there is a reason to allow that, especially when you are live coding on stage! In this documentation, you will find many instances of unknown words being translated to strings such as `sine`, `kick` or `saw`. Cagire doesn't know about them, they have no meaning when the script is evaluated. These are not special words that were already registered. They are barewords that a sound word reads as the name of a sound source to use.

```
sine snd .
```

`sine` is the string `"sine"`. The flip side is that a misspelled word usually shows up as a stray string rather than a failure. If a line seems to do nothing, check the spelling. You might have a misspelled word somewhere that ended up being transformed into a string.

The exception is a near miss. A name of five characters or more that sits a single edit away from a built-in, or of four or more that sits a single edit away from one of your own words, is read as a typo instead of a string, and the line stops with a suggestion: `choos` gives `unknown word 'choos'; did you mean 'choose'?`. Anything further from a real name still becomes a string, which is why a source name like `sine` travels safely.

2.3.4 Comments

Anything after `;;` is a comment. Everything that comes after it on the same line is ignored:

```
saw snd c4 note . ;; a plain saw at middle C
;; this is also a comment
```

The `;;` and the words after it are skipped, so the saw plays as if they were not there.

You can also use a multiline variant of comments: `;*` and `*;`. Everything between them is ignored, across as many lines as you like, and the block does not nest, so the first `*;` ends it. They can prove useful sometimes but in practice are clunkier to use, especially for live coding:

```
sine sound
;*
2 fm 0.5 fmh
.2 chorus
*;
2 decay
.
```

The full lexical rules, including how numbers, strings, and the comment forms are tokenised, are in *Lexical structure* (3.2).

2.4 Values & types

Every value that flows through Cagire has a type: the kind of thing it is. A number is one type, a piece of text another, a block of held-back code another still. You never declare a type by hand and you can't create your own types. Cagire works it out from the way you write the value. Digits make an integer, quotes make a string, parentheses make a quotation. A value's type is what decides which words will accept it and what they do with it, so reading the type tells you how a value will behave before you run anything. Cagire has a small fixed set of value kinds, and they all ride on the same stack.

Kind	How you write it	Example
Integer	digits	`60`
Float	decimal point or ratio	`0.5`, `3/4`

Kind	How you write it	Example
String	quotes, or a bareword	`"kick"`, `sine`
Quotation	parentheses	`( 2 * )`
List	square brackets, or several values on one parameter	`[ 60 64 67 ]`
Silence	`rest`	`rest`
Tuning	`edo`, `tuning`	`12 edo`
Scale	`scale`, or a name	`major`
Pattern	`pat`	`"x.x." 4 pat`

The list is the one kind with two ways in. You write one directly with `[ ]`, and you also get one by handing several values to a single parameter, where the register keeps them together as one value per voice; that second form reaches the stack again when you read the parameter back with `'name`. Each kind has its own reference: numbers and strings in *Lexical structure* (3.2), quotations in *Quotations & combinators* (2.7), groups in *Polyphony* (2.5.5), the `[ ]` and `< >` brackets that build and pick from runs of values in *Lists* (3.6) and *Selection & cycling* (3.12), tunings and scales in *Pitch & harmony* (3.17), patterns in *Patterns* (3.16).

2.4.1 When types are checked

Cagire checks types as it runs, not before. A word looks at the values on top of the stack the moment it executes, and if they are the wrong type it stops the line there and reports it: hand a string to `+` and you get `expected number`, ask for text where a number sits and you get `expected string`. Nothing is verified ahead of time, so a line can read perfectly and still fail the instant a word meets a value it cannot use. The messages, and what happens to a line that fails, are in *Error handling* (2.10).

2.4.2 Numbers convert, other types do not

Numbers are the one type that bends. Integers and floats mix freely: in arithmetic, comparison, and equality an integer is promoted to a float when it meets one, so `2 0.5 +` is `2.5` and `60 60.0 =` is true. When a word needs a whole number it truncates a

float toward zero, and `floor`, `ceil`, and `round` let you make that conversion on purpose. No other type converts on its own. A string never turns into a number, a list never becomes a quotation; a value keeps the type it was written as, and the only way to a different type is to build a new value.

2.4.3 Truth and comparison

Cagire has no boolean type. In a test, `0`, `0.0`, an empty string, `rest`, and an empty list or pattern count as false. Everything else is true, including every quotation, tuning, and scale, even an empty one. Comparisons and logic answer with `1` for true and `0` for false.

```
3 4 lt print
```

`lt` is "less than", and `3` is less than `4`, so this prints `1`. The bare `<` and `>` signs are taken by cycles, so comparisons read as words: `lt`, `gt`, `<=`, `>=`, `=`, and `!=`. The full set of comparison and logic words is in *Stack & math words* (3.3).

2.5 The sound register

The sound register is a record that holds the description of one sound. It keeps a source and a set of named parameters: `note` for pitch, `gain` for level, `lpf` for a filter cutoff, `verb` for reverb, and so on. It is separate from the stack and does a different job. The stack carries the loose values you compute with; the register collects the ones that belong to the sound you are building, each stored under the name of the parameter it sets. You fill it with `snd` and parameter words, and `.` reads the whole register and turns it into sound.

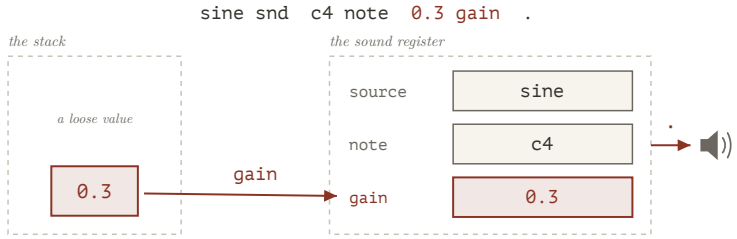


Figure 26: A parameter word like gain takes a value off the stack into the register under its own name; . reads the whole register out to the engine.

2.5.1 Starting a sound

`sound` (or its short form `snd`) takes a source name and opens the register. The name is a string, and since a word Cagire does not recognise is already a string (see *Words & comments* (2.3)), you usually just write it bare:

```
sine snd .
```

This builds a sine and plays it. When we say that we are *emitting* a sound, it means that we have fed the content of the sound register to the audio engine. An audio event is played as a result of this operation.

2.5.2 Adding parameters

A parameter word takes a value from the stack and stores it in the register:

```
sine snd
0.3 gain
.
```

`0.3 gain` sets the level. The resulting sine will be quieter. Order does not matter to the register, only to the stack. `0.3 gain sine snd .` builds exactly the same thing. Pitch is the `note` parameter, and note names work as values. We have already seen this example before:

```
sine snd
c4 note
.
```

This is another example of adding values to the sound register. You can layer as many parameters as you like, to the point where you will end up with sounds having 20~30 parameters in extreme cases:

```
square snd
c4 note
2000 lpf
.
```

This is now playing a square wave at middle C going through a lowpass filter at 2000 Hz.

Parameter	Sets
`note`	pitch, a MIDI number or note name
`gain`	level, 0 to 1
`gate`	note length
`pan`	stereo position, 0.5 is centre
`lpf`	lowpass cutoff in Hz
`delay`	delay mix
`verb`	reverb mix

The table above is describing a handful of common parameters that the sound register can accept. The full set is in the *Dictionary* (2.1) under the Sound, Envelope, Filter, and effect categories. Each is described in *The engine* (IV). A parameter word takes every loose value on the stack, not just the top one. That is how a chord works, several values to one word, as in Polyphony below. So set one parameter at a time, and do not leave stray values on the stack before you name one: a number left sitting before `snd`, for instance, becomes part of the sound and raises an error.

2.5.3 What the register keeps

. sends the register but does not empty it, so you can fire the same sound again.

```
sine snd c4 note . .
```

This program above will play the sine twice as two events starting at the same time. The source and parameters stay until you change them, so a later `.` repeats whatever is currently in the register. If you set the same parameter twice, the last value wins. The register lasts for one evaluation only. It starts empty each time a step fires and is gone when the program ends, so nothing you build leaks into the next fire. This is the opposite of variables, which keep their values for the whole session (see *Scope and variables* (2.9)). The one exception is what you set with `all`, which carries over from one fire to the next; see *Applying parameters to everything* (2.5.6) below.

Cagire chooses absent envelope defaults according to the source. A synth gets a four-step `gate` and one-step `release`; a sample gets a pattern-long gate and twelve-step release; a GM instrument gets the gate but keeps its SoundFont envelope; and a native drum keeps its kit envelope and engine gate. Values that Cagire adds are scaled to the tempo. User-set time parameters such as `gate` and `release` are also measured in musical time rather than fixed seconds.

2.5.4 Clearing and replacing

You will sometimes need to clear the sound register manually. To do so, you can use the `clear` word. Nothing stays in the sound register after it and you can start fresh. You can even use it in the middle of a running program:

```
sine snd
0.3 gain
clear
saw snd
.
```

The `clear` wipes the sine and its gain, so only a plain saw plays. Starting a new `sound` replaces the source but keeps the parameters you already set, so reach for `clear` when you want a clean slate.

2.5.5 Polyphony

Give a parameter several values and the register fans out into one voice per value, so a single sound command can play a whole chord; a lone value is shared by every voice, and a `rest` among the values drops one. How the lists wrap and how a `chord` quality expands is in *Polyphony* (2.5.5).

2.5.6 Applying parameters to everything

`all` takes the parameters currently in the register, applies them to every voice already emitted on this step, and keeps them sticky for every sound that follows, on this step and the steps after it, until `noall` clears them or playback stops. It is how you set something once and have it hold across a whole group of sounds. The `all` and `noall` pair is detailed in *The sound command* (3.4).

2.5.7 Reusing a parameter

Sometimes you want one parameter to depend on another. A `'` in front of a parameter name reads its current value back out of the register onto the stack, so you can compute with it and feed the result into a second parameter. This `'param` read, and the rule that you must set a parameter before you read it, is described in *Variables* (3.7).

2.6 The dictionary

A word is a name Cagire knows how to run. `+`, `dup`, and `snd` are words, and so is every other command in the language. The dictionary is the whole collection of them, the book Cagire looks a name up in to find what it does. Cagire takes both the idea and the name from Forth, the language family it grows out of, where a program is just words and the dictionary that holds them. Naming the set of commands a dictionary is unusual, but it is the whole organizing idea here: the language is words, and you can add your own. The dictionary comes in two layers: the built-in words that ship with Cagire, and the words you define while you play. When

Cagire reads a word it looks through what it already knows first, the note names and the built-ins, and your own words second.

2.6.1 Built-in words

The built-ins are the words Cagire is made of: stack moves and arithmetic, oscillators and filters, envelopes and effects, sequencing, music theory, MIDI, and OSC. There are more than four hundred of them, grouped into a few dozen categories you can browse in the Dictionary view. You do not need to memorise them. The *Dictionary* (2.1) view lists every word with its category, an example, and a stack effect written as `(before -- after)`, which names what the word takes from the stack and what it leaves. For instance `+` is `(a b -- a+b)`.

2.6.2 Defining your own words

You are not limited to the built-in words. A word can be defined anywhere in any step script. Definition starts with `:`, then the name, then the body, then `;`, and afterwards the new word behaves like any built-in.

```
: double 2 * ; 5 double print
```

This defines `double` to multiply by two, then uses it, so it prints `10`. Definitions are the start of building your own language on top of Cagire. Redefining words, recursion with `recurse`, and removing words with `forget` are covered in the reference under *Defining words* (3.11.1) and *Removing words* (3.11.2).

2.6.3 Words that take arguments

A word's body runs on whatever the stack holds when it is called, so `double` already takes an argument: the `5` it multiplies. When a body uses a value more than once, or takes more than one input, name it. Write a brace block right after the word's name and list the argument names inside it; Cagire binds the stack to them when the word runs, and from then on writing a name pushes that value back.

```
: square { n }  n n * ;
7 square print
```

`7 square` binds `n` to `7` when the word runs, so `n n *` prints `49`. The binding is made fresh on every call and is gone when the body ends. Nothing leaks to the caller or into the next call. A definition also opens a fresh scope: a name bound outside it is not visible in the body, whose inputs arrive on the stack. List several names for several arguments and they bind in call order, the deepest value taking the first name. The details, including how `:>` names an intermediate partway through a body and why bindings come before `snd`, are in *Defining words* (3.11.1).

2.6.4 Built-ins win

Because Cagire checks the built-ins before your words, you cannot replace or shadow a built-in. A definition that reuses a built-in name is accepted, but it is never reached, and Cagire gives no warning.

```
: dup 99 ; 5 dup print
```

`dup` is the built-in that duplicates the top of the stack. The new definition is never reached, so `5 dup` is still `5 5`, and this prints `5`, not `99`. So give your words descriptive names, and be wary of single letters, since some are taken by built-in aliases; `a`, for example, is short for `attack`. Note names and interval names are spoken for as well, and they resolve even before the built-ins, so a word you call `e4` or `M3` is never reached either. Numbers are the one exception: a numeric literal is not a built-in, and Cagire checks your words before reading a token as a number, so a definition named `2` is reached. *Defining words* (3.11.1) covers that curiosity.

A local binding bends the rule, in the other direction. While it is in scope, whether it names an argument in the head block or an intermediate with `:>`, the bound name resolves before everything else: built-ins, note names, and your own words alike. The shadowing is local to one body and ends with it, so the dictionary itself is untouched.

```
: twice { dup }  dup dup + print ;
3 twice
```

Inside `twice`, `dup` is the argument bound to `3`, not the built-in, so `dup dup +` is `3 3 +` and prints `6`. Naming an argument after a

built-in is rarely a good idea, but it shows the order: locals first, built-ins second, your words third.

2.6.5 Keeping your words

The words you define live for as long as the session. To keep them between sessions, save them in your prelude or personal dictionary, described under *User dictionary* (4.3). There is no word that lists the dictionary from inside the language. Browsing is done in the Dictionary view.

2.7 Quotations & combinators

A quotation is a piece of code kept as a value. You write it between parentheses, and Cagire does not run it when it reads it. It puts the block on the stack, where it waits like any other value until something runs it. Keeping code as a value is what makes it useful: a word can take the quotation off the stack and decide what happens to it, whether to run it, when, or how many times. A word that acts on a quotation this way is a combinator. Quotations are the held code and combinators are the words that run it, two halves of how Cagire expresses control flow. This page covers both, the quotation first and then the combinators.

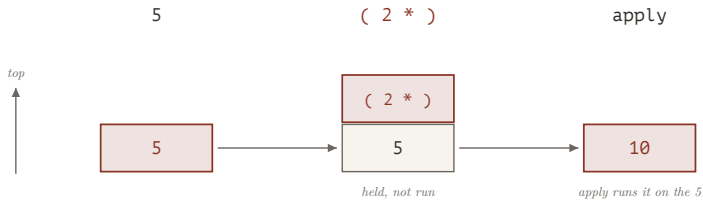


*Figure 27: Reading (2 *) holds the code as one value on the stack above the 5; apply runs it on the 5, leaving 10.*

```
( 2 * )
```

This pushes one value: the code `2 *`, held for later. Nothing is multiplied yet.

2.7.1 Running a quotation

`apply` takes a quotation off the stack and runs it on whatever is below.

```
5 ( 2 * ) apply print
```

The `5` is on the stack, `( 2 * )` waits above it, and `apply` runs the block on the `5`, doubling it. This prints `10`. Because a quotation is just a value, you can store it in a variable and run it later.

```
( 2 * ) !double 5 @double apply print
```

`!double` stores the block under `double`, `@double` puts it back on the stack, and `apply` runs it on the `5`; this prints `10` as well. Variables are covered in *Scope and variables* (2.9).

A stored quotation is frozen code, but it does not carry a local with it. If a quotation reads a local, an argument named in a head block or a value named with `:>`, and you run it after the binding word has returned, the name is gone and Cagire reports it out of scope. Run such a quotation on the spot, with `apply`, `dip`, or `at`, while the local is still in scope. Locals are covered in *Defining words* (3.11.1).

2.7.2 Combinators

Combinators come in three families. Some run a quotation only under a condition. `?` runs it when the value before it is true.

```
( 0.4 verb ) 1 ? saw snd .
```

The `1` is true, so the reverb is applied and the saw plays with `0.4` of it. Change the `1` to `0` and the block is skipped. Some shape the stack around a quotation. `dip` runs it while holding the top value aside.

```
c4 e4 ( 12 + ) dip note saw snd .
```

`e4` is held aside, the octave is added to `c4`, then `e4` comes back, so both notes play. Some run a quotation many times. `map` runs it over every item of a list.

```
[ 60 64 67 ] ( 12 + ) map note sine snd .
```

Each note is raised an octave, so the chord becomes `72`, `76`, `79`.

Those are one taste from each family. The full set, with `apply`, `dip`, `keep`, `?`, `!?`, `ifelse`, `select`, `times`, `map`, `filter`, `each`, and `gen`, is documented under the *Combinators* (2.7.2) reference: *Application & dataflow* (3.10.1), *Conditional* (3.10.2), and *Iteration* (3.10.3). Every one of them expects a quotation on the stack, and that is the shape of control flow in Cagire: put the code in a quotation, then hand it to a word that decides its fate.

2.7.3 Quotations nest

A quotation can hold other quotations. They are values like any other, so they sit inside without being run.

```
3 ( ( 2 * ) apply ) apply print
```

The outer `apply` runs `( ( 2 * ) apply )`, which itself runs `( 2 * )` on the `3`; this prints `6`. You meet this when a combinator chooses between blocks that themselves contain blocks.

2.7.4 The mute trick

A quotation that nothing runs simply sits on the stack and is dropped at the end of the line. Wrapping a line in parentheses is a quick way to silence it while you edit, without deleting it.

```
( kick snd . )
```

The kick is built and described but never fired, so you hear nothing. Remove the parentheses and it plays again. This is handy for muting one voice in a busy step. Parentheses defer code. Square brackets are different. They run their contents at once and collect whatever they leave into one list. That bracket and the words that take a list are covered in *Lists* (3.6).

2.8 The live model

Everything so far has treated a script as something you type and run once. In a playing pattern, things are different. The playhead comes back around, and the same script fires again on every pass. The script will repeat over time. A thirty-two-step pattern looping

for a minute can run its scripts hundreds of times. How steps and patterns are arranged is the subject of the interface part. Here we care about what re-running means for the language. Running the same script over and over would be dull if it always did the same thing. It does not have to, because a script can ask where it is in time and answer differently each run. That is the live model: the code is fixed, the moment moves, and the sound evolves.

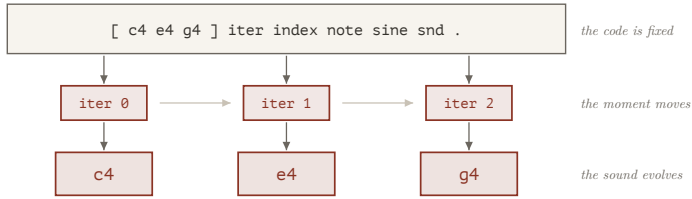


Figure 28: One unchanging script fired on three passes; iter advances 0, 1, 2, so index picks c4, then e4, then g4.

2.8.1 Reading the moment

Cagire exposes the state of the moment as a set of context words. You name them and they push their current value, so a line can react to where it sits on the timeline. Almost all of them are read-only. A few of the most used:

Word	Pushes
`step`	current step index
`phase`	position in the bar, 0 to 1
`tempo`	current tempo in BPM
`runs`	times this step has run
`iter`	times the pattern has looped

```
tempo print
```

Prints the current tempo, `120` by default. The rest of the context words, including `nsteps`, `beat`, `pattern`, `pbank`, `slot`, `stepdur`, `fill`, and on the desktop build the mouse position, are listed in *Context* (3.8). Because these are values like any other, you feed

them straight into the words you already know. Pick a note by where you are in the pattern:

```
[ c4 e4 g4 ] iter index note sine snd .
```

`index` reads a value from the group by number, and here the number is `iter`, the loop count. On the first loop it plays `c4`, on the next `e4`, then `g4`, then it wraps. The script never changes; `iter` does.

2.8.2 Two counters: runs and iter

Two of the context words count repeats. The difference between them matters once you have more than one step or more than one pattern:

- `runs` counts how many times one particular step has played. Each step keeps its own count.
- `iter` counts how many times the whole pattern has looped. Every step in the pattern shares the same `iter`, and it ticks once per lap around the pattern.

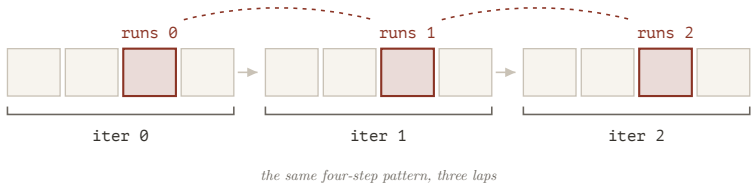


Figure 29: A four-step pattern looping three times: iter ticks once per lap and is shared, while runs counts one step's own plays.

`runs` is local to a step and `iter` is global to the pattern. This split runs through the rest of the language: the cycling words come in pairs, one driven by `runs` and one by `iter`, and the periodic words like `every` count laps with `iter`. You meet them in *Selection & cycling* (3.12) and *Periodicity & timing* (3.14); this is the idea they are built on.

The counters are also the one part of the moment you can write. `runs!` sets a step's own count and `iter!` sets the pattern's shared lap count, so a phrase can start itself over instead of waiting for

the numbers to wrap. The run that writes keeps the count it was handed; the new value is what the next run reads.

```
sine snd runs 100 * 200 + freq . runs 3 = if 0 runs!
then
```

The pitch climbs with the count, 200, 300, 400, 500 hertz, and on the fourth run the write sends the count home, so the rise loops every four runs instead of climbing forever. Both words are covered in *Context* (3.8).

2.8.3 Why a line can change while it plays

Editing a step while it plays does not stop it. Cagire runs your new version on the next step, and because the moment keeps moving, a freshly edited line picks up the current `iter`, `phase`, and the rest at once. This is what makes Cagire feel live: you change the code, time carries on, and the next run reflects both. What happens when an edit is broken is the subject of *Error handling* (2.10).

2.9 Scope and variables

Scope is the part of a program where a name means something. In many languages a name is visible only inside the function or block that introduces it, and the same word can mean different things in different places. Cagire has one scope and no blocks: every name lives in a single table that every pattern, every bank, and the prelude share, and that table lasts the whole session. Set a name anywhere and it means the same thing everywhere, until you change it or quit. Most of the time you don't name values at all, you pass them on the stack. Variables are for when you want to set one aside and reach for it later by name. A variable is just a name with a value that you can fetch from anywhere in your scripts.

2.9.1 Setting and reading

Three special forms manage variables. To deal with variables, you have to prepend a symbol to a name, depending on what behavior you expect. The value goes on the stack first, then the word.

Form	Effect
`!name`	store the top value under `name`, taking it off the stack
`,name`	store the top value under `name`, leaving it on the stack
`@name`	push the value stored under `name`

```
5 !x
@x print
```

`5 !x` stores `5` as `x` and removes it from the stack. `@x` puts it back; `print` shows `5`. The difference between `!` and `,` is whether the value stays behind. Use `,name` when you want to keep working with the value after storing it.

```
5 ,x 1 + print
```

`5 ,x` stores `5` as `x` but leaves the `5` on the stack, so `1 +` makes `6`. This prints `6`, and `x` still holds `5`. Reading a name you never set gives `0`, not an error. Be careful with this, as it can lead to unexpected results if you make a typo in your program:

```
@undefined print
```

A related form, `'name`, reads back a parameter you have set in the sound register, not a variable. It is described in *The sound register* (2.5). It looks like a variable and feels like a variable, which is why it should be mentioned here.

2.9.2 Local bindings

Variables are global and lasting. Locals are the exception: a name that belongs to one word and is gone when it returns. There are two forms. A brace block right after a definition name, `{ x }`, names the word's arguments; a `:>` names an intermediate partway through a body. Both bind a value to a bareword you read back without a sigil, and both are covered in *Defining words* (3.11.1).

```
: half { x }  x 2 / print ;
8 half
```

`8 half` binds `x` to `8` inside `half` and prints `4`. Outside `half`, `x` means nothing again. Reach for a local when a value belongs to one

word, and for a variable when it should outlive the line that set it and be seen elsewhere.

2.9.3 Carrying state across the session

Because every name is shared and outlives each step, variables are how you carry state through a whole session. A variable set on one line is there on the next line, in another pattern, or on a later step: a shared root note, a counter, a value you want several patterns to agree on. Variables live in memory only. They are not saved with the project, and loading a project clears them. To have one ready every time, set it in your prelude, described under *User dictionary* (4.3).

2.9.4 The loop counter

The one name Cagire sets for you is `i`, the counter inside a `times` loop. It counts from zero.

```
( @i print ) 3 times
```

`times` runs the quotation three times and `@i` reads the counter each time, so this prints `0`, `1`, and `2`. Because `i` is an ordinary variable, a `times` loop overwrites any `i` you set yourself, and its last value remains after the loop ends. The looping words are covered in *Quotations & combinators* (2.7).

2.10 Error handling

Live coding means typing code that is sometimes wrong, often in the middle of a performance. Cagire is built so that a mistake is cheap. A line that does not make sense is reported and skipped, and whatever was already playing keeps playing. You are never one typo away from silence, and that is what lets you edit boldly while a set is running.

2.10.1 When are errors caught?

An error is caught either before a script runs or while it runs, at runtime. A line that cannot be read at all fails to compile. In that

case, nothing runs at all. An unclosed quotation, a definition with no `;`, a stray bracket are all caught before runtime:

```
( 1 2 +
```

The quotation is never closed, so this fails with `missing )`. A line that reads cleanly but asks for something impossible will fail during runtime, a second type of error. Dividing by zero is a runtime error:

```
3 0 /
```

2.10.2 A failed line is almost a no-op

When a line fails, Cagire restores the stack, leaves your variables as they were, and emits no sound. It is as close to never having run as it can be, so you fix the line and try again from a clean state. The one exception is the dictionary: a word definition or a `forget` takes effect as the line is read, before it runs, so those stay even if a later word on the same line fails.

2.10.3 Errors do not halt the performance

The previous version of your script continues to run if you ever encounter an error. This is what makes live editing safe. While a pattern is playing and you edit its code, Cagire runs the new version on the next step. If the new version errors, the last version that worked keeps sounding instead of cutting out. You hear no gap. You see the error and fix it. A step that has never had a working version stays silent until it does.

2.10.4 Where errors show

The error message appears in the footer. While the pattern is playing, the word that caused the error is highlighted in the editor. You can see exactly where the script broke. Code that evaluates cleanly produces no message.

Message	Means
`missing )` / `missing ]`	an unclosed quotation or list
`missing ';' in word definition`	a `:` definition with no `;`

Message	Means
`unknown word 'X'; did you mean 'Y'?`	a name one edit away from a real word, caught as a typo (names further off become strings)
`stack underflow`	a word wanted more values than were on the stack
`expected number/string`	a word got the wrong kind of value
`division by zero`	dividing or taking a remainder by zero
`nothing to emit`	`.` with no sound built in the register
`call depth exceeded`	a recursion went past the depth limit

Chapter 3

Language reference

3.1 Overview

You are entering the Language reference. This section documents Cagire's words family by family, word after word, for looking one up once you know your way around. Words that deal with the audio engine itself (oscillators, filters, and effects) live in the next chapter: *The engine* (IV). Only language-level words will be documented in this section. This section cannot be read linearly, read pages that you find interesting and explore slowly.

Family	Look under
Tokens, literals, comments, brackets	*Lexical structure* (3.2)
Stack shuffling, arithmetic, comparison, logic, `print`	*Stack & math words* (3.3)
Starting and firing a sound	*The sound command* (3.4)
Playing several voices at once	*Polyphony* (3.5)
The `[ ]` list value	*Lists* (3.6)
Storing and reading values	*Variables* (3.7)
Reading the playing moment	*Context* (3.8)
`if` and `case` branching	*Control flow* (3.9)
Running quotations	*Combinators* (3.10)
Defining and removing words	*Definitions* (3.11)
Cycling and choosing from a group	*Selection & cycling* (3.12)
Random values and chance	*Randomness* (3.13)
Firing on a cycle or a fraction of a step	*Periodicity & timing* (3.14)
Building runs of values and rhythms	*Generators* (3.15)
Rhythm mini-notation	*Patterns* (3.16)
Notes, scales, chords, tuning	*Pitch & harmony* (3.17)
LFOs, slides, envelopes	*Motion* (3.18)

Family	Look under
MIDI, OSC, and recording	*MIDI, OSC & recording* (3.19)

Every entry gives the word and its stack effect written as `(before -- after)`. You will also learn about existing word aliases, potential errors or edge cases. To find one word fast without browsing the documentation, it is preferable to use the in-app *Dictionary* (2.1). The dictionary is much faster to browse.

3.2 Lexical structure

Cagire reads your source as a flat stream of tokens separated by whitespace. A whole script is one program, newlines are just whitespace, and the stack clears once per evaluation. Every token is one of five things: an integer, a float, a string, a `{ }` patch capture, or a word. Whitespace is the only separator, so how you lay code out is up to you. The two lines below belong to the same program:

```
1 2
+ print
```

Cagire reads this as `1 2 + print` and prints `3`.

3.2.1 Numbers

A token that parses as a whole number becomes an integer. One with a decimal point becomes a float. Cagire tries integer first, then float:

```
2.5 print
```

You can drop the leading zero on a fraction. A token that starts with `.` followed by a digit, or `-.` followed by a digit, is read as a shorthand float:

```
.25 print
```

Prints `0.25`. Likewise `-.5` is `-0.5`. A token written as `num/den` is a ratio literal: Cagire divides the two integers and pushes the result as a float. The denominator cannot be zero, which is a compile error:

```
3/4 print
```

Prints `0.75`.

3.2.2 Strings

Text between double quotes is a string literal. A word Cagire does not recognise also becomes a string, which is how bare source names like `kick` reach `snd` (see *Words & comments* (2.3)):

```
"hello" print
```

Prints `hello`. An unknown word does the same.

```
wobble print
```

`wobble` is not a word, so it becomes the string `wobble` and prints it. A name that lands within a single edit of a real word is caught as a typo rather than becoming a string, and the line stops with a suggestion (see *Words & comments* (2.3)).

3.2.3 Comments and definition end

Cagire has two comment forms, `;` to the end of a line and `;* ... *;` across several lines, both taught in *Words & comments* (2.3). Two tokenizer details matter here. A single `;` is a different token from `;;`: it ends a word definition (see *The dictionary* (2.6)). And inside a `{ }` patch capture only `;;` and that lone `;` are recognised, so a `;* ... *;` block comment written inside braces is not stripped and breaks the patch.

3.2.4 Inline patches

Curly braces capture everything up to the matching `}` as one raw patch token, a kind of its own rather than a quoted string, which is what lets the same braces mean something else right after a definition name. Nothing inside is tokenized, so a whole synth definition written in arf with its own `;`, `/`, and `:` travels as a single string value. This is how inline patches reach `synth` and `defsynth`:

```
{ 110 saw 800 lpf out } synth .
```

The braces push the patch source as a string, `synth` installs it and sets it as the sound, and `.` plays it. Braces nest, so a patch can

contain its own `{ a b }` locals blocks. A stray `}` outside a block is a tokenizer error, and an unclosed `{` is too. The patch language has its own part of the manual: *A second language* (1).

The one place braces mean something else is right after a definition name. `: name { a b } ... ;` reads the block as the word's argument list and binds the stack to those names (see *Defining words* (3.11.1)). Position is the only signal: a `{ ... }` that opens a definition body is an argument list, a `{ ... }` anywhere else is a patch, and Cagire never inspects the contents to decide. The names inside a head block must be plain words, so a number or a nested patch there is a compile error rather than a patch.

3.2.5 Bracket forms

There are many bracket types in Cagire. They are often used as syntax sugar for manipulating lists and groups of values on the stack. They are all documented in their proper section:

Form	What it does	See
`( ... )`	quotation, deferred code pushed as one value	*Quotations & combinators* (2.7)
`[ ... ]`	runs immediately, collects what it leaves into one list	*Lists* (3.6)
`< ... >`	single alternation, advances per pattern iteration	*Alternation brackets* (3.12.2)
`<< ... >>`	double alternation, advances per step run	*Alternation brackets* (3.12.2)
`{ ... }`	raw capture, patch source pushed as one string	*Inline patches* (3.2.4) above

Square brackets are a very common occurrence. They run their contents and push everything that landed as one list.

```
[ 60 64 67 ] print
```

This leaves a single value on the stack, the list of three notes, so `print` shows `[ 60 64 67 ]`. Alternation brackets pick one value per iteration. A dash makes them an arrow that sets direction, covered in *Selection & cycling* (3.12).

```
< c4 e4 g4 > note sine snd .
```

Plays one note from the list and steps to the next each time the pattern loops.

3.2.6 Compile-time word forms

A few words are handled while Cagire compiles, not at run time. They shape the program rather than the stack. `:` opens a definition and `;` closes it. The body in between becomes a new word. A brace block right after the name, `: name { a b } ... ;`, names the word's arguments and is the one place braces are not a patch (see *Defining words* (3.11.1)).

```
: double 2 * ;
5 double print
```

Defines `double`, then `5 double` prints `10`. Definitions are covered in *The dictionary* (2.6). `:>` binds the top of the stack to a local named by the next word, for the rest of the current word; naming intermediates this way is covered in *Defining words* (3.11.1). `if` / `else` / `then` and `case` choose between branches at compile time. They are covered in *Control flow* (3.9).

```
1 if 42 then print
```

The condition `1` is truthy, so the branch runs and prints `42`.

3.3 Stack & math words

These words rearrange the stack and crunch numbers on it. They never touch the sound register. They only take values and leave values. They are the plumbing under every line you write. For how the stack works at all, see *The stack* (2.2); for what counts as a value, see *Values & types* (2.4). In the tables below, examples use `=>` to show what the stack holds afterwards.

3.3.1 Copying and reordering

Word	Stack effect	What it does	Example
`dup` (`_`)	`(a -- a a)`	Duplicate the top	`5 dup => 5 5`
`dupn` (`!`)	`(a n -- a a ... a)`	Duplicate the value `n` times	`7 3 dupn => 7 7 7`
`drop`	`(a --)`	Remove the top	`1 2 drop => 1`
`swap`	`(a b -- b a)`	Exchange the top two	`1 2 swap => 2 1`
`over`	`(a b -- a b a)`	Copy the second item to the top	`1 2 over => 1 2 1`
`rot`	`(a b c -- b c a)`	Rotate the top three	`1 2 3 rot => 2 3 1`
`nip`	`(a b -- b)`	Remove the second item	`1 2 nip => 2`
`tuck`	`(a b -- b a b)`	Copy the top under the second	`1 2 tuck => 2 1 2`
`2dup`	`(a b -- a b a b)`	Duplicate the top pair	`1 2 2dup => 1 2 1 2`
`2drop`	`(a b --)`	Drop the top pair	`1 2 2drop =>`
`2swap`	`(a b c d -- c d a b)`	Exchange the top two pairs	`1 2 3 4 2swap => 3 4 1 2`
`2over`	`(a b c d -- a b c d a b)`	Copy the second pair to the top	`1 2 3 4 2over => 1 2 3 4 1 2`

`dup` copies the top item, so you can use one value twice. `dup` turns `5` into `5 5`, then `+` adds them, printing `10`:

```
5 dup + print
```

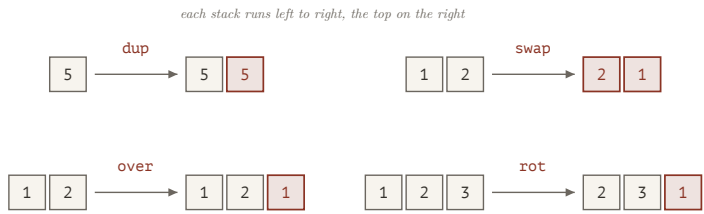


Figure 30: Four reordering words as before and after, top on the right; the brick tile marks what dup, swap, over, and rot each left on top.

3.3.2 Inspecting

`depth` counts the stack and `print` shows a value, neither of which changes what you are building.

Word	Stack effect	What it does	Example
`depth`	`(-- n)`	Push how many items are on the stack	`1 2 3 depth => 1 2 3 3`
`print`	`(x --)`	Show the top value in the footer bar	`42 print`

`depth` reports the current size of the stack without disturbing it. It tells you how much is sitting there, which is what you want while chasing a line that leaves more behind than you meant it to.

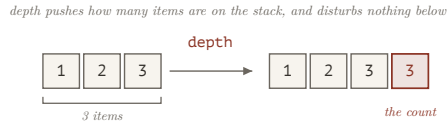


Figure 31: depth before and after, top on the right: on 1 2 3 it pushes the count 3 (the brick) and leaves the items untouched below.

```
1 2 3 depth print
```

There are three items, so this prints `3` and leaves `1 2 3` underneath. `print` is the inspection tool used throughout these docs: it consumes the top value and writes it to the footer, where numbers and strings show plainly and other kinds show in a rough internal form. Don't try to print too much; this is not what you have downloaded Cagire for!

3.3.3 Folding and reordering many items

These words act on a whole list at once. Each one takes the list off the stack and works on every item inside it, so `[ ]` is how you gather the values first (see *Lists* (3.6)).

Word	Stack effect	What it does	Example
`sum`	`(list -- total)`	Add up a list	`[ 1 2 3 ] sum => 6`

Word	Stack effect	What it does	Example
`prod`	`(list -- product)`	Multiply a list together	`[ 2 3 4 ] prod => 24`
`sort`	`(list -- list)`	Sort a list ascending	`[ 5 1 3 ] sort => [ 1 3 5 ]`
`rsort`	`(list -- list)`	Sort a list descending	`[ 5 1 3 ] rsort => [ 5 3 1 ]`
`rev`	`(list -- list)`	Reverse a list	`[ 1 2 3 ] rev => [ 3 2 1 ]`
`shuffle`	`(list -- list)`	Randomly reorder a list	`[ 1 2 3 ] shuffle`

`sum` collapses a list into its total.

```
[ 1 2 3 ] sum print
```

The brackets gather `1 2 3` into one list, `sum` adds them, and this prints `6`. Swap `sum` for `prod` to multiply them instead. `sort`, `rsort`, and `rev` hand back a list holding the same values in a new order, while `shuffle` draws on the random generator, so it is reseedable with `seed` (see *Random values* (3.13.1)).

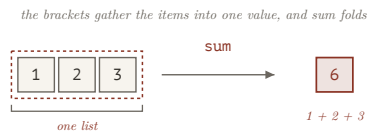


Figure 32: sum before and after, top on the right: the list [1 2 3] collapses into its total, the brick 6.

3.3.4 Arithmetic

The binary operators take two values, the unary ones take one, and `clamp` takes three. Division is true division, so `5 2 /` is `2.5`. Cagire only keeps the float when it has to; land exactly on a whole number and you get an integer back.

Word	Stack effect	What it does	Example
`+`	`(a b -- a+b)`	Add	`2 3 + => 5`

Word	Stack effect	What it does	Example
`-`	`(a b -- a-b)`	Subtract	`5 3 - => 2`
`*`	`(a b -- a*b)`	Multiply	`4 3 * => 12`
`/`	`(a b -- a/b)`	Divide	`7 2 / => 3.5`
`mod`	`(a b -- a%b)`	Remainder of `a / b`	`7 3 mod => 1`
`neg`	`(a -- -a)`	Negate	`5 neg => -5`
`abs`	`(a -- |a|)`	Absolute value	`-5 abs => 5`
`sign`	`(a -- -1|0|1)`	Sign as `-1`, `0`, or `1`	`-5 sign => -1`
`floor`	`(f -- n)`	Round down	`2.7 floor => 2`
`ceil`	`(f -- n)`	Round up	`2.1 ceil => 3`
`round`	`(f -- n)`	Round to nearest	`2.6 round => 3`
`trunc`	`(f -- n)`	Round toward zero	`-2.7 trunc => -2`
`min`	`(a b -- min)`	Smaller of two	`3 7 min => 3`
`max`	`(a b -- max)`	Larger of two	`3 7 max => 7`
`clamp`	`(val lo hi -- clamped)`	Hold a value between a low and high bound	`5 0 3 clamp => 3`
`pow`	`(a b -- a^b)`	Raise `a` to the power `b`	`2 8 pow => 256`
`sqrt`	`(a -- sqrt a)`	Square root	`16 sqrt => 4`

`mod` gives the remainder, which is handy for wrapping a counter back into a range.

```
7 3 mod print
```

Seven divided by three leaves a remainder of one, so this prints `1`.

3.3.5 Trigonometry and logarithms

These take an angle in radians, a value to take the natural logarithm of, or a power to raise `e` to. They are most useful for shaping modulation curves by hand.

Word	Stack effect	What it does	Example
`sin`	`(a -- sin a)`	Sine, in radians	`0 sin => 0`
`cos`	`(a -- cos a)`	Cosine, in radians	`0 cos => 1`

Word	Stack effect	What it does	Example
`tan`	`(a -- tan a)`	Tangent, in radians	`0 tan => 0`
`log`	`(a -- ln a)`	Natural logarithm	`1 log => 0`
`exp`	`(a -- e^a)`	Raise `e` to the power	`0 exp => 1`

```
0 cos print
```

The cosine of zero is one, so this prints `1`.

3.3.6 Mapping ranges

`linmap` rescales a value from one range to another in a straight line; `expmap` does it on an exponential curve, which suits frequencies and other things the ear hears logarithmically. `expmap` always reads its input as a `0` to `1` position.

Word	Stack effect	What it does	Example
`linmap`	`(val inlo inhi outlo outhi -- mapped)`	Linear rescale between two ranges	`64 0 127 200 2000 linmap`
`expmap`	`(val lo hi -- mapped)`	Exponential rescale of a `0-1` value into `lo-hi`	`0.5 200 8000 expmap`
`range`	`(val min max -- scaled)`	Scale a `0-1` value onto `min-max`	`0.5 200 800 range`

`linmap` is the everyday choice for turning a MIDI value into a parameter range.

```
64 0 127 200 2000 linmap print
```

A `64` sitting just past the middle of `0-127` maps to about `1107` in the `200-2000` range.

```
0.5 200 8000 expmap print
```

Halfway up the curve between `200` and `8000` lands at about `1265`, much lower than the linear midpoint because the curve is exponential. `range` is the simplest of the three: it assumes its input is already a `0` to `1` position and scales it straight onto `min-max`. That

is why it pairs so naturally with a sampled shape, which already hands you a `0` to `1` value (see *Sampling motions* (3.18.4)).

```
0.5 200 800 range print
```

Half of the way from `200` to `800` is `500`.

3.3.7 Joining strings

`++` glues two values into one string, turning numbers into their digits along the way. It is how you build a source name out of a stem and a number, such as picking one of several samples by index.

Word	Stack effect	What it does	Example
`++`	`(a b -- s)`	Concatenate as a string	`"osc" 2 ++ => osc2`

```
"osc" 2 ++ print
```

This stringifies the `2` and joins it on, printing `osc2`.

3.3.8 Comparison

Every comparison leaves `1` for true and `0` for false, ready to feed a conditional combinator. There is no `<` or `>` word, since the angle brackets build alternations (see *Alternation brackets* (3.12.2)); use `lt` and `gt` instead.

Word	Stack effect	What it does	Example
`=`	`(a b -- bool)`	Equal	`2 2 = => 1`
`!=` (`<>`)	`(a b -- bool)`	Not equal	`2 3 != => 1`
`lt`	`(a b -- bool)`	Less than	`2 5 lt => 1`
`gt`	`(a b -- bool)`	Greater than	`5 2 gt => 1`
`<=`	`(a b -- bool)`	Less than or equal	`2 2 <= => 1`
`>=`	`(a b -- bool)`	Greater than or equal	`3 2 >= => 1`

```
2 5 lt print
```

Two is less than five, so this prints `1`. Equality compares numbers by value across integers and floats, and any number against a non-number is simply unequal rather than an error.

3.3.9 Logic

These treat any non-zero value as true and `0` as false, and they too leave `1` or `0`. Combine them with the comparisons to build the test for a conditional.

Word	Stack effect	What it does	Example
`and`	`(a b -- bool)`	True if both are true	`1 0 and => 0`
`or`	`(a b -- bool)`	True if either is true	`1 0 or => 1`
`not`	`(a -- bool)`	Invert a truth value	`0 not => 1`
`xor`	`(a b -- bool)`	True if exactly one is true	`1 1 xor => 0`
`nand`	`(a b -- bool)`	Not `and`	`1 1 nand => 0`
`nor`	`(a b -- bool)`	Not `or`	`0 0 nor => 1`
`xnor`	`(a b -- bool)`	Not `xor`, true when both match	`1 1 xnor => 1`

```
1 1 nand print
```

Both inputs are true, so `and` would be true and `nand` inverts it to `0`. For the words that consume these truth values to decide what to run, such as `?`, `ifelse`, and `select`, see *Conditional* (3.10.2).

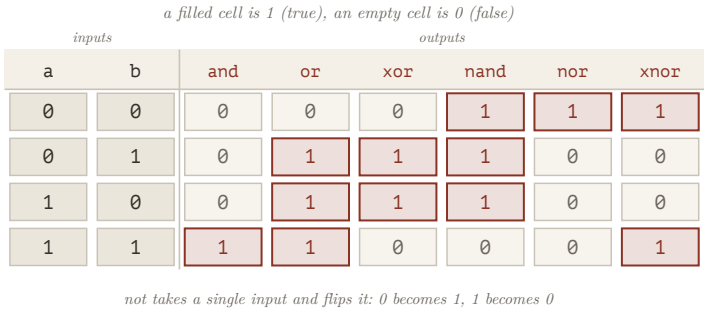


Figure 33: Truth table for the six binary logic words: a and b down the left, one operator per column, filled cells for 1.

3.3.10 Errors and edges

There are some must-know errors when dealing with numbers and logic:

- Taking from an empty or too-shallow stack stops the line with `stack underflow`, and the line changes nothing.
- `/` and `mod` by zero are errors.
- Integers and floats mix freely, and results collapse to an integer whenever nothing remains after the decimal point: `4 2 /` is `2` but `5 2 /` is `2.5`.
- `expmap` expects a positive low and high, since it works on a ratio between them.
- `++` joins numbers and strings, but not quotations, tunings, or scales. Those raise a type error.

3.4 The sound command

A sound is built in a small workspace called the register: you name a source, set parameters, then fire. These are the words that open, fire, and reset that register. The parameter words you set in between, like `note`, `gain`, and `lpf`, belong to *The engine* (IV); the concept is walked through in *The sound register* (2.5).

Word	Stack effect	What it does
`sound` (`snd`)	`(name --)`	open the register on a source
`wt`	`(name --)`	open the register on a sample, read as a wavetable
`.`	`(--)`	fire the register
`rest`	`(-- rest)`	push silence, a skipped voice
`clear`	`(--)`	empty the register
`all`	`(--)`	apply the current params to every voice
`noall`	`(--)`	stop applying them

3.4.1 Opening and firing

`sound (name --)` opens the register on a source named by the string below it. Until you fire, every parameter word you write adds to this one sound. It aliases to `snd`:

```
saw snd c4 note .
```

`saw` is the source, `snd` opens the register, `c4 note` sets the pitch, and `.` fires it. `saw` is a bareword, so it arrives as the string `"saw"` (see *Words & comments* (2.3)).

`wt (name --)` opens the register the same way, on a sample you want read as a wavetable rather than played back as a recording. It is the only other word that opens the register, and everything after it works exactly as it does after `snd` (see *Wavetable oscillators* (2.2)).

```
pad wt c4 note .
```

`. (--)` fires the register, sending the sound you have described to the engine. Nothing is heard until you write it, and the register keeps its contents afterwards, so you can fire the same sound again.

```
sine snd .
```

Plays one sine voice. If you set no `gate`, the voice is given a default length and a release tail so it still sounds and fades. Set `gate` yourself for an exact length, where `0` holds the voice open until a later positive `gate` ends it.

3.4.2 Resting and clearing

`rest (-- rest)` pushes silence. Among a group of notes it masks one voice, so that slot plays nothing, and in a test it counts as false.

```
60 rest 67 note sine snd .
```

Three pitches go in, but the middle is `rest`, so only `60` and `67` sound.

`clear (--)` empties the register, both the source and every parameter, so you can start a fresh sound on the same line.

```
"kick" snd 0.5 gain . clear "hat" snd .
```

The kick fires at half gain, `clear` wipes the register, and the hat starts fresh without the kick's gain.

3.4.3 Applying to every voice

`all (--)` takes the parameters currently in the register and applies them to every voice. It patches sounds already fired in this evaluation and stays sticky for the sounds you emit afterwards,

not just on this step but on every step that follows, until `noall` or playback stops.

```
"bd" snd . "sn" snd . 0.3 verb all
```

Both drums fire dry, then `0.3 verb all` adds the same reverb to both and to anything that follows.

`noall (--)` stops the global application started by `all`, so later sounds no longer inherit those parameters.

```
noall
```

Clears the global parameters on its own line.

3.5 Polyphony

One sound command can play several voices at once. You do not open a new register for each voice. You stack several values before a parameter, and Cagire fans the sound out into one voice per value. Chords, stacked detunes, and spread panning all work this way. The register these voices come from is described in *The sound command* (3.4).

3.5.1 Values become voices

Give a parameter more than one value and the sound plays them together.

```
60 64 67 note sine snd .
```

Three notes go onto the stack, `note` takes all three, and the sine sounds a C major chord. This works for any parameter, not only `note`. A list holds the same three values as one, so `[ 60 64 67 ] note` fans out into the same chord and keeps the group together while you build it (see *Lists* (3.6)).

The source is a group of its own. Name several before `snd` and each voice takes one, so a single command can fan out across different sounds.

```
kick snare snd 60 64 note .
```

Two voices, a kick at note 60 and a snare at 64. One source is shared by every voice, exactly the way one parameter value is.

When a parameter holds just one value, every voice uses it. So you set the pitches as a group and the rest of the sound once.

```
60 64 67 note 0.4 gain saw snd .
```

Three voices at notes 60, 64, and 67, each at gain 0.4. The single `0.4` reaches all of them.

Set a parameter with several values and each voice takes one, in order.

```
60 64 67 note -0.5 0 0.5 pan sine snd .
```

Three voices, panned hard left, centre, and hard right. A list shorter than the number of voices wraps around by position.

```
60 64 67 note 0.4 0.8 gain sine snd .
```

Three voices but only two gains, so the values cycle: `0.4`, then `0.8`, then `0.4` again for the third voice.

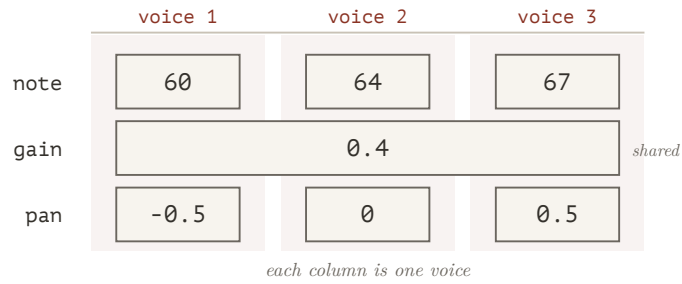


Figure 34: One sound command fans out into three voices; note and pan are lists (one value each), while the lone gain is shared by all three.

3.5.2 How many voices play

The voice count is the length of the longest group among the source, the pitch, the parameters, and any chord. Every shorter group lines up from the first voice and wraps to fill the rest. A lone value counts as a group of one, so it reaches every voice.

```
60 note 0.2 0.5 0.8 gain sine snd .
```

One pitch but three gains, so three voices play, all at note 60, at gains 0.2, 0.5, and 0.8.

Loose values do not wait their turn: whatever sits on the stack when a parameter word runs belongs to it, including things you meant for later. So build one parameter's values, name it, then move to the next. In `60 64 67 note 0.4 0.8 gain`, `note` clears the three pitches first, then `0.4 0.8` land on `gain`. Brackets are the way out when a line gets crowded: `[ 60 64 67 ] note` hands `note` one value that no stray can join. This is covered in *Lists* (3.6).

3.5.3 Silencing a voice

`rest` stands in for a value and masks that one voice, leaving a gap in the chord.

```
60 rest 67 note sine snd .
```

The middle slot is a `rest`, so the chord plays with a hole in it: `60` and `67` sound, nothing in between.

3.5.4 Chords are polyphony

Setting a `chord` quality is another way to reach several voices. It expands the root from `note` into the chord's tones at emit, so one `note` and one `chord` give a full voicing.

```
c4 note
maj7 chord
sine snd
.
```

Four voices, the C major seventh at 60, 64, 67, and 71. Building chords from a quality, an anchor, or single tones is covered in *Chords* (3.17.3).

3.5.5 Building the values

To stack several values for a parameter, write them out, make them with a generator such as `..` or `gen`, or transform a group with `map`.

```
[ 60 64 67 ] ( 12 + ) map note sine snd .
```

`map` raises each note an octave before `note` takes them, so the chord plays at 72, 76, and 79. The selection words such as `cycle` and `choose` do the opposite: they collapse a group down to a single value (see *Cycling words* (3.12.1) and *Choice* (3.12.3)).

3.6 Lists

Square brackets `[ ]` run their contents right away and collect everything they leave into one value: a list. `[ 60 64 67 ]` is a single item on the stack, not three, and it stays one item until a word opens it. This is the opposite of `( )`, which holds its code back instead of running it (see *Quotations & combinators* (2.7)). A list is an ordinary value, so you can store one in a variable, hand it to a word you defined, compare it, and print it. Most of the words that work on several values at once ask for a list, and there is never a count to tally.

3.6.1 What `[ ]` leaves

The brackets run the code between them exactly as if you had written it outside them, then sweep up whatever it left and push that as one list.

```
[ 60 64 67 ] print
```

The stack holds one value and `print` shows it as `[ 60 64 67 ]`, written the way you would type it, so a printed list can be pasted straight back into a script. The body is ordinary code, which means the list holds what the code produced rather than what you typed.

```
[ 2 dup dup ] print
```

One number and two `dup`s leave three values, so this prints `[ 2 2 2 ]`. Brackets with nothing between them make an empty list.

```
[ ] print
```

Prints `[ ]`. An empty list is false in a test, so you can ask whether a generator gave you anything before you use it (see *Values & types* (2.4)).

3.6.2 A list is a value

Anything that carries a value carries a list. A variable holds one and gives it back whole.

```
[ 60 64 67 ] !notes
@notes print
```

`!notes` stores the three notes under one name and `@notes` puts that list back on the stack, so this prints `[ 60 64 67 ]`. Variables are covered in *Scope and variables* (2.9).

A word you define takes a list the same way, so a head block can name one as its argument.

```
: octave { xs } xs 12 + ;
[ 60 64 67 ] octave print
```

`octave` names its argument `xs`, lifts it by twelve, and this prints `[ 72 76 79 ]`. Definitions and head blocks are covered in *Defining words* (3.11.1).

Two lists are equal when they hold equal values in the same order.

```
[ 60 64 67 ] [ 60 64 67 ] = print
```

The two lists match item for item, so this prints `1`.

3.6.3 Nesting

A list holds values, and a list is a value, so lists nest.

```
[ [ 1 2 ] [ 3 4 ] ] print
```

This prints `[ [ 1 2 ] [ 3 4 ] ]`, one outer list of two inner ones, `print` recursing into each. An inner pair of brackets closes before the outer one does, so a word written between them works on the inner list alone and its result becomes an item of the outer list.

```
[ [ 1 2 ] sum [ 3 4 ] sum ] sum print
```

The two inner sums leave `3` and `7`, the outer brackets collect those two numbers, and the last `sum` adds them to `10`.

3.6.4 Arithmetic reaches inside

Give an arithmetic word a list and a number and it works on every item, handing you a list back. This is how you transpose, scale, or offset a whole group in one move.

```
[ 60 64 67 ] 12 + note sine snd .
```

The `12` reaches each note, so the chord sounds an octave up at `72`, `76`, and `79`. Every binary math word behaves this way and so does every unary one, so `[ -1 2 -3 ] abs` is `[ 1 2 3 ]` and `[ 1 2 3 ] 2 *` is `[ 2 4 6 ]`.

3.6.5 A list on a parameter is a chord

A parameter word swallows every loose value on the stack, and it takes a list the same way: one item per voice.

```
[ c4 e4 g4 ] note sine snd .
```

Three voices, a C major triad, exactly the sound `c4 e4 g4 note` gives you. The list travels as one value up to the parameter, where it becomes the fan-out described in *Polyphony* (3.5). The source is a group in the same sense, so `[ kick snare ] snd` names two sources, and `[ 1 3 ] speakers` lists output pairs the way `1 3 speakers` does.

3.6.6 The words that take a list

A word that works on several values at once asks for the list itself. They fall into a few families:

- Folding and reordering: `sum` and `prod` hand back a number, while `rev`, `sort`, `rsort`, and `shuffle` hand back a list. See *Stack & math words* (3.3).
- Cycling and choosing: `cycle`, `pcycle`, `rcycle`, `prcycle`, `bounce`, `pbounce`, `walk`, `pwalk`, `urn`, `purn`, `choose`, `wchoose`, and `index` collapse a list to one of its items. See *Cycling words* (3.12.1) and *Choice* (3.12.3).
- Running a quotation over the items: `map` and `filter` hand back a list, `each` leaves whatever the quotation leaves, and `gen` builds a list from nothing. See *Iteration* (3.10.3).

- Building a temperament: `tuning` and `scale` read their steps from a list. See *Tuning & microtonality* (3.17.4).
- Placing hits inside a step: `at` takes a list of offsets as readily as loose ones. See *Fractional timing* (3.14.2).

```
[ c4 e4 g4 ] cycle note sine snd .
```

`cycle` takes the list and leaves one of its three notes, advancing each time the step runs, so `note` plays a single voice that walks the list. At the start of playback it sounds `c4`. The alternation brackets `<` `>` are shorthand for this pairing of a list with a cycling word (see *Alternation brackets* (3.12.2)).

Hand one of these words something that is not a list and it stops the line with `expected list`. Hand a selection word an empty list and it stops with `choose: empty list`, naming the word that asked, since there is no item to pick. `sum` and `prod` are the exception: an empty list adds up to `0` and multiplies to `1`.

3.6.7 `select` is not one of them

`select` looks like these words and is not one. Its number is an index into loose quotations sitting on the stack, so bracketing them hands it a list where it wants blocks, and it finds none to run.

```
( c4 ) ( e4 ) ( g4 ) 1 select note sine snd .
```

Index `1` runs the second block, so this plays `e4`. Leave the quotations loose. `select` is covered in *Conditional* (3.10.2).

3.7 Variables

A variable is a named box for one value. You store with `!`, read with `@`, and the name is whatever you write right after the sigil, with no space. Variables persist across steps and across the whole session, so one step can leave a value for another to read. The performer's view is in *Scope and variables* (2.9).

Word	Stack effect	What it does
`!name`	`(val --)`	store into name

Word	Stack effect	What it does
`@name`	`(-- val)`	read name
`,name`	`(val -- val)`	store, keep a copy
`'param`	`(-- val)`	read back a set parameter
`:> name`	`(v --)`	name an intermediate (a body-local binding)

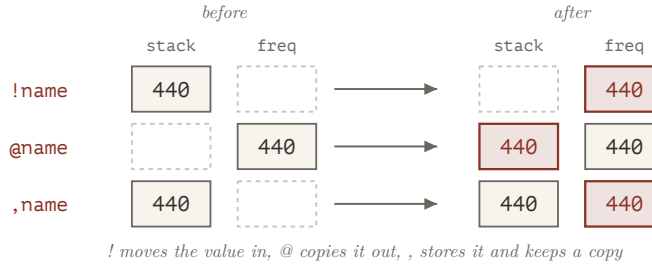


Figure 35: The three variable words compared by what they leave: !name empties the stack, @name keeps the box freq, ,name keeps a copy.

3.7.1 !name

`!name (val --)` stores the value below it into the variable `name`, consuming it.

```
440 !freq
```

Puts `440` into `freq`. The write is held until the line finishes without error, so a line that fails leaves the old value untouched.

3.7.2 @name

`@name (-- val)` reads the variable `name` onto the stack. A name that was never set reads as `0`.

```
440 !freq @freq print
```

Stores `440`, reads it back, and prints `440`.

3.7.3 ,name

`,name (val -- val)` stores into `name` like `!`, but leaves a copy on the stack, so you can use a value and save it at once.

```
440 ,freq 2 * print
```

Saves `440` into `freq` and prints `880`, the kept copy doubled.

3.7.4 ‘param

`'param (-- val)` reads back the value of a parameter you set in the current register. The name must be a real parameter word: an unknown name is a compile error, and reading a parameter you have not set is a runtime error. The value comes from the register, so a value you set locally wins over anything `all` has applied.

```
500 freq 'freq print
```

Sets `freq` to `500`, reads it back, and prints `500`. A parameter you set with several values reads back as the whole group, one value per voice, rather than a single number (see *Polyphony* (3.5)).

3.7.5 :> name

`:> name (v --)` pops the top value and binds it to a local called `name` for the rest of the word or line, read back as a bareword with no sigil. Unlike a variable it does not outlive that word and is not shared. Naming intermediates with `:>`, and why the bind must come before `snd`, is covered in *Defining words* (3.11.1).

3.7.6 The loop counter i

`times` sets the variable `i` to the loop index, counting from zero, and clobbers any `i` of your own. The counter is covered in *Scope and variables* (2.9), and `times` itself in *Iteration* (3.10.3).

3.8 Context

Cagire hands the playing moment to your scripts as a set of context words. Naming one pushes what it holds right now, so the same script can do different things depending on where the playhead sits. Most of them are read-only; the few that write back are gathered at the end of this page. This is the machinery behind the live model, described in *The live model* (2.8).

3.8.1 Reading the moment

Word	Stack effect	What it does	Example
`step`	`(-- n)`	Current step index	`step print`
`nsteps`	`(-- n)`	Number of steps in the pattern	`nsteps print`
`beat`	`(-- f)`	Current beat position	`beat print`
`phase`	`(-- f)`	Position in the bar, 0 to 1	`phase print`
`pattern`	`(-- n)`	Current pattern index	`pattern print`
`pbank`	`(-- n)`	Current pattern's bank index	`pbank print`
`slot`	`(-- n)`	Current slot number	`slot print`
`tempo`	`(-- f)`	Current tempo in BPM	`tempo print`
`runs`	`(-- n)`	Times this step has run	`runs print`
`iter`	`(-- n)`	Times the pattern has looped	`iter print`
`stepdur`	`(-- f)`	Step duration in seconds	`stepdur print`
`fill`	`(-- bool)`	1 while Fill is on, else 0	`fill print`

```
tempo print
```

The tempo lands on the stack and `print` shows it, `120` unless you changed it. Nothing distinguishes these values from ones you typed yourself; any word that takes a number will accept them. Here the loop count picks a note from a list:

```
[ c4 e4 g4 ] iter index note
sine snd
.
```

`index` picks from the group by position, and the position comes from `iter`. Loop one sounds `c4`, loop two `e4`, loop three `g4`, and the fourth wraps back around. The script never changes; `iter` does.

3.8.2 The fill switch

`fill` is the odd one out: it does not read the clock, it reads you. Press `f` and Cagire turns Fill on; press it again and Fill goes off. While it is on, a lit F badge appears next to the tempo in the header. A step that names `fill` sees 1 while it is on and 0 while it is off, so you can change what a step plays the very moment you reach for the key, the way you would on the fill button of a drum machine.

Key	What it does
`f`	Toggle Fill on or off, from any view except *Patterns* (3.16), where `f` opens the follow editor

```
fill if "snare" else "hat" then snd .
```

The step plays a hat as it loops, and a snare for as long as Fill is on. The script never changes; you do, live, with one key.

3.8.3 runs versus iter

`runs` is local to a step, the number of times that one step has played, and `iter` is global to the pattern, the number of times the whole pattern has looped. This pairing drives the cycling and periodic words (see *Cycling words* (3.12.1) and *Periodic execution* (3.14.1)). The distinction is explained in *The live model* (2.8).

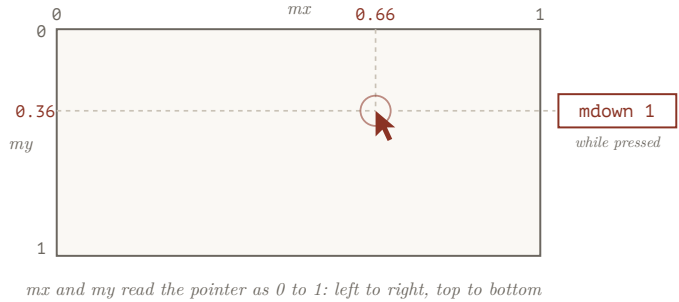
3.8.4 Desktop: the mouse

On the desktop build, three more context words read the mouse, which turns it into a live controller.

Word	Stack effect	What it does	Example
`mx`	`(-- f)`	Mouse X, 0 to 1	`mx print`
`my`	`(-- f)`	Mouse Y, 0 to 1	`my print`
`mdown`	`(-- bool)`	1 while the mouse button is held, else 0	`mdown print`

```
mx 440 880 range freq sine snd .
```

`mx` reads the horizontal mouse position and `range` maps it from 0 to 1 onto 440 to 880 hertz, so sliding the mouse sweeps the pitch.



mx and my read the pointer as 0 to 1: left to right, top to bottom

Figure 36: mx runs left to right and my top to bottom over the window, matching the screen; here the cursor sits at mx 0.66, my 0.36.

3.8.5 tempo!

The moment is mostly read-only, but two words write back to the transport. `tempo! (bpm --)` sets the global tempo in beats per minute, clamped to 20-300.

```
140 tempo!
```

Sets the tempo to 140 BPM. The read-only `tempo` above always reports whatever is currently set.

3.8.6 speed!

`speed! (multiplier --)` scales how fast the pattern advances against the tempo, clamped to 0.125-8.

```
2.0 speed!
```

Runs the pattern at double rate without changing the tempo.

3.8.7 theme!

`theme! (n --)` switches the color theme by number. The index wraps around the theme list, so any integer lands on a theme, and negative numbers count back from the end.

```
2 theme!
```

Because it is just a number going in, `theme!` can ride `every` or `chance` like anything else, so the whole interface changes color along with the music.

3.8.8 runs! and iter!

The two counters can be written as well as read. `runs! (n --)` sets the run count of the step that executes it, and `iter! (n --)` sets the loop count of the whole pattern. A write never touches the run that makes it; that run keeps the count it was handed, and the new value is what the next one reads. Negative values clamp to 0, and restarting a pattern still resets both counters as it always did.

```
0 runs!
```

The next run of this step sees `runs` at 0, so everything driven by step runs (`cycle`, `walk`, `bjork`, and their kin) starts over with it.

```
( 0 iter! ) 8 every
```

Sends the pattern's loop count back to 0 on every eighth lap. `iter` is shared by every step, so one step writing it re-steers the `iter`-driven words across the whole pattern, and the phrase they spell repeats as an eight-lap unit.

3.9 Control flow

3.9.1 if / else / then

`if` decides whether a stretch of code runs. You push a condition, write `if`, and everything up to `then` runs only when that condition is true. Cagire compiles these directly into branch instructions, so they are syntax rather than dictionary words and will not show up in the *Dictionary* (2.1). These words are a bit special and rarely encountered in a live performance.

if ... then

`cond if ... then` runs the body when `cond` is true and skips it when `cond` is false. `if` consumes the condition it tests, `(flag --)`. Truthiness follows *Values & types* (2.4): `0`, an empty string,

`rest`, and any other empty value count as false, and everything else counts as true.

```
3 2 gt if 100 then print
```

`3 2 gt` is true, so the body pushes `100` and this prints `100`. Had the condition been false the body would be skipped, and with nothing left on the stack the `print` would fail, so make sure the path you take leaves the stack the way the rest of the line expects.

if ... else ... then

Add `else` for a two-way split. The `if` branch runs when the condition is true, and the `else` branch runs when it is false. Exactly one of the two runs.

```
2 3 gt if
100
else
200
then print
```

`2 3 gt` is false, so the `else` branch runs and this prints `200`. The condition does not have to be a comparison. Any value works, and any non-zero number passes.

```
0 if 1 else 2 then print
```

`0` is false, so this prints `2`. In a sound, branch on something live, such as the loop counter, to change the voice as the pattern repeats.

```
iter 2 mod if saw else sine then snd c4 note .
```

`iter` is how many times the pattern has looped (see *Context* (3.8)). On odd loops `iter 2 mod` is true and you get a saw, and on even loops you get a sine.

For the quotation form of conditionals, where the branches sit on the stack as `( ... )` blocks and a word runs or skips them, see *Conditional* (3.10.2) in the combinators.

3.9.2 case

`case` matches one value against several options. It reads cleaner than a chain of `if`s once you have more than two branches. Like `if`, it is compiled syntax and does not appear in the *Dictionary* (2.1).

Matching clauses

Push the value to match, write `case`, then a clause for each option as `test of ... endof`, and close with `endcase`. Cagire compares the value to each `test` in turn, and the first clause whose `test` is equal runs its body. The construct consumes the matched value, `(value --)`, and the matched body replaces it with its own result.

```
1 case
0 of 10 endof
1 of 20 endof
2 of 30 endof
endcase print
```

The value `1` matches the `1 of` clause, so that body pushes `20` and this prints `20`. The tests are usually constants, but each one is ordinary code, so anything that leaves a value works. A clause body is ordinary code too, so one `case` can hold another. A common shape is a live counter folded into range with `mod`, so the match changes as the pattern repeats.

```
iter 2 mod case
0 of 0.9 gain endof
1 of 0.4 gain endof
endcase
sine snd
c4 note .
```

`iter 2 mod` is `0` or `1` on every loop, so one clause always matches: a louder sine on even loops and a quieter one on odd loops (see *Context* (3.8) for `iter`).

The default clause

The code between the last `endof` and `endcase` is the default. It runs when no clause matched. The value being matched is still on the stack when the default runs, and `endcase` clears it afterwards, so a default that just keeps the value uses `dup`.

```
9 case
0 of 60 endof
1 of 62 endof
dup endcase print
```

`9` matches neither clause, so the default runs, `dup` keeps the value past `endcase`, and this prints `9`. With a matching value such as `1` the `1 of` clause would replace the value and print `62` instead. Give every `case` either a default or a value sure to match. An unmatched `case` with no default drops the value and leaves nothing in its place, so the rest of the line then comes up a value short. For one-way and two-way branching, see *if / else / then* (3.9.1). Equality and what counts as a match follow *Values & types* (2.4).

3.10 Combinators

3.10.1 Application & dataflow

These combinators run a quotation and shape the stack around it. They do not decide whether to run it, they always do. What differs is which values the quotation gets to see. For deciding whether to run, see *Conditional* (3.10.2). For running over many values, see *Iteration* (3.10.3). The quotation form `( ... )` is the subject of *Quotations & combinators* (2.7).

Word	Stack effect	What it does
`apply`	`(quot --)`	Run a quotation on what is below
`dip`	`(a quot -- ..a)`	Run with the top set aside, then restore it
`keep`	`(a quot -- ..a)`	Run on the top, keeping the original too

apply

`apply (quot --)` runs a quotation on whatever is below it, no questions asked.

```
5 ( 2 * ) apply print
```

`apply` pops the quotation and runs it against what remains, so the `5` underneath doubles. This prints `10`.

dip

`dip (a quot -- ..a)` sets the top value aside, runs the quotation on what remains, then puts the top back.

```
c4 e4 ( 12 + ) dip note saw snd .
```

`e4` is held aside, `( 12 + )` raises `c4` an octave, then `e4` comes back, so the chord is the raised first note and the untouched second. `dip` is the relief valve when the value you want is buried just under the top.

keep

`keep (a quot -- ..a)` runs a quotation on the top value but gives you the original back as well, so you end up with both.

```
c4 ( 7 + ) keep note saw snd .
```

`( 7 + )` makes the fifth above `c4`, and `keep` hands back the original `c4`, so the chord is the root and its fifth.

3.10.2 Conditional

These combinators decide whether, or which, quotation to run, from a value on the stack. They are how a line chooses what to do instead of just doing it. For the quotation form `( ... )` itself, start at *Quotations & combinators* (2.7).

Word	Stack effect	What it does
`?`	`(quot bool --)`	Run the quotation if true
`!?`	`(quot bool --)`	Run the quotation if false

Word	Stack effect	What it does
`ifelse`	`(true-quot false-quot bool --)`	Run one of two by the condition
`select`	`(..quots n --)`	Run the nth quotation, from zero

? and !?

`? (quot bool --)` runs a quotation only when the value before it is true, and `!? (quot bool --)` runs it only when the value is false.

```
( 0.4 verb ) 1 ? saw snd .
```

The `1` is true, so the reverb is applied. With `0` it would be skipped. What counts as true follows the rules in *Values & types* (2.4); the short version is that `0`, an empty string, `rest`, and any other empty value fail the test and anything else passes.

ifelse

`ifelse (true-quot false-quot bool --)` chooses between two quotations. The true branch comes first, then the false one, then the condition.

```
( c5 note ) ( c3 note ) 1 ifelse saw snd .
```

The condition is `1`, so the first block runs and the note is `c5`. A `0` would pick `c3`.

select

`select (..quots n --)` picks one quotation from several by index, counting from zero.

```
( c4 ) ( e4 ) ( g4 ) 1 select note sine snd .
```

Index `1` runs the second block, so the note is `e4`. `select` is a truly powerful word when paired with moving data. Unlike `index` it does not wrap: reaching past the last quotation stops the line, so fold a live value into range with `mod` first. You never bracket the

quotations either, since `[ ]` would gather them into one list and leave `select` no loose blocks to choose from.

3.10.3 Iteration

These combinators run a quotation more than once: a fixed number of times, once per item of a list, or repeatedly to build a run of fresh values.

Word	Stack effect	What it does
`times`	`(quot n --)`	Run the quotation n times, `@i` holds the index
`map`	`(list quot -- list)`	Run over every item, replacing each
`filter`	`(list quot -- list)`	Keep each item the quotation likes
`each`	`(list quot --)`	Run over every item and leave what it leaves
`gen`	`(quot n -- list)`	Run n times and collect the results into a list

times

`times (quot n --)` runs a quotation a fixed number of times. Inside the loop, `@i` reads the counter, which counts from zero.

```
( @i print ) 3 times
```

Runs the block three times and prints `0`, `1`, `2`. The counter `i` is described in *Scope and variables* (2.9).

map

`map (list quot -- list)` runs a quotation over every item of a list and hands back a list of the results.

```
[ 60 64 67 ] ( 12 + ) map note sine snd .
```

The quotation adds `12` to each pitch in turn, so `note` receives `[ 72 76 79 ]` and the chord sounds an octave up. The quotation has to leave a value on every pass, since `map` takes one result per item.

filter

`filter (list quot -- list)` runs a quotation over every item and keeps the ones for which it comes back true.

```
[ 60 62 64 65 67 ] ( 63 gt ) filter note saw snd .
```

`63 gt` is true for values above `63`, so the list narrows to `[ 64 65 67 ]` and those three play. The quotation's result is only the test; what `filter` keeps is the item itself.

each

`each (list quot --)` runs a quotation over every item too, then gets out of the way: whatever the quotation leaves stays loose on the stack. Where `map` hands you a list back, `each` hands you the results themselves, which is what a parameter word wants.

```
[ 0 3 7 ] ( c4 + ) each note sine snd .
```

The list holds intervals, the quotation turns each one into a note above `c4`, and the three loose values feed `note` as a chord: `60`, `63`, `67`. Reach for `each` when the results go straight onto a parameter, and for `map` when you want to keep working on them as a list.

gen

`gen (quot n -- list)` runs a quotation `n` times and collects the results into a list. Where `times` repeats an effect, `gen` builds values.

```
( 0 8 rand ) 4 gen note sine snd .
```

`( 0 8 rand )` picks a whole number from 0 to 8, and `gen` runs it four times, so the list of four random notes reaches `note` and they sound together as a chord. The quotation has to leave a value behind on every pass; one that leaves nothing stops the line. Because `at` reads a list of offsets as readily as loose ones, a `gen` run also drops straight into it (see *Fractional timing* (3.14.2)).

3.11 Definitions

3.11.1 Defining words

A definition starts with `:`, then the name, then the body, then `;`. Afterwards the new word behaves like any built-in.

```
: double 2 * ;
5 double print
```

The body multiplies by two, so `5 double` leaves `10` behind for `print`.

Give your words descriptive names. Single letters are risky, because some are taken by built-in aliases; `a`, for example, is short for `attack`. You cannot shadow a built-in either: Cagire checks the built-ins before your words, so naming yours after one compiles fine but the built-in keeps winning every lookup (see *The dictionary* (2.6)).

Arguments in a head block

A definition can name the values it takes off the stack. Write a brace block right after the name, list the argument names inside it, and Cagire binds the top of the stack to them the moment the word is called. From then on writing a name pushes its value back. This turns a word into a small instrument with knobs, rather than one that bakes its numbers in.

```
: bell { bright }  sine snd  14 fmh  bright fmfb . ;
0.9 bell
```

`0.9 bell` binds `bright` to `0.9`, `14 fmh` sets a fixed ratio, and `bright fmfb` feeds the knob to the feedback. Call it with a different number for a different timbre, and `0 1 rand bell` gives a fresh one every fire.

The names bind in call order. List several and the deepest value on the stack takes the first name, the top takes the last, so they read left to right the way you pushed them:

```
: tone { carrier ratio }  saw snd  carrier fm  ratio
fmh . ;
200 3 tone
```

`200 3 tone` binds `carrier` to `200` and `ratio` to `3`, each name taking the value in the same position you wrote it. Arguments bind at the very top of the body, before `snd` or any parameter word runs, so a value you name is safe even though those words drain the rest of the stack (that draining is how you ask for several voices, see *The sound register* (2.5)):

```
: bamkick { level }  "kick" snd  level distort . ;
4 bamkick
```

`level` is bound before `snd`, so `snd` swallowing the rest of the stack cannot touch it, and `level distort` still finds its `4`.

The braces are part of the definition form, but only in this one place. A `{ ... }` right after the name is the argument list; a `{ ... }` anywhere else in a body opens an inline patch handed to Arf (see *A second language* (1)), so position alone tells the two apart, and the content is never guessed at. The names must be plain words: a number, a string, a comment, or a nested patch inside the block is a compile error, and so is an empty `{ }`. While the body runs, an argument name wins over everything with the same spelling, a built-in, a note name, one of your own words, and the shadowing ends when the word returns.

An argument lives only while the word runs. A quotation the word runs on the spot, with `dip`, `keep`, `apply`, `map`, or `at`, sees it. A quotation the word stores to run later does not, because by then the word has returned and the name is gone. Locals also appear on their own in *Scope and variables* (2.9).

Naming intermediates with :>

The same idea names a value partway through a body. `:>` takes the top of the stack and binds it to the name that follows, right where you write it, so you can compute something once and read it back by name instead of juggling `dup` and `swap`.

```
: swell  saw snd  0 0.8 rand :> amt  amt gain  amt
distort . ;
swell
```

`0 0.8 rand` picks one amount, `:> amt` names it, and the two reads feed the same value to `gain` and `distort`. The name is fresh on every call and gone when the body ends, exactly like an argument, but it is bound in the middle rather than declared up front.

The one order to respect is the drain again. `snd` and the parameter words take every value on the stack, so a `:>` reaching for a value that one of them already ate finds nothing and reports a stack underflow. Compute and name such a value before you emit, or let it come off the stack after, the way `0 0.8 rand` does above.

Redefining your own words

You can redefine a word you made, and the new version takes over from then on. But words that already used it keep the version they were built with: a definition captures the words it calls at the moment you write it.

```
: bass 1 ;
: lead bass 10 + ;
: bass 100 ;
lead print
```

`lead` was built while `bass` was `1`, so it stays `11` even after `bass` becomes `100`. A fresh use of `bass` would see `100`. This keeps older words steady while you experiment, and it is what lets you build a new version of a word on top of its old one.

Redefining numbers

A definition name does not have to be a word. It can be a number. Redefining a number is unusual, but it is real.

```
: 2 4 ;
2 print
```

After this, `2` pushes `4`, so it prints `4`. Use it sparingly, since it is easy to confuse yourself, and `"2" forget` puts it back (see *Removing words* (3.11.2)).

Recursion

A word cannot call itself by its own name, because that name still points at the previous definition. To recurse, use `recurse (--)`, which calls the word being defined.

```
: fact
dup 2 lt if
drop 1 else
dup 1 - recurse *
then ;
5 fact print
```

This is factorial, and `5 fact` prints `120`. Recursion is capped at a fixed depth, so a runaway definition stops with an error instead of locking up.

3.11.2 Removing words

`forget (name --)` removes one of your words. Put the name as a string in front of it.

```
: temp 42 ;
"temp" forget
temp print
```

After `forget`, `temp` is no longer a word, so it falls back to a plain string and this prints `temp`. Built-in words cannot be forgotten. At the top level of a step, `forget` is also applied as the line compiles. The name is already gone for the rest of that same script. The opposite of forgetting is keeping: words meant to survive a session belong in a prelude or personal dictionary, described under *User dictionary* (4.3).

3.12 Selection & cycling

3.12.1 Cycling words

Cagire's cycling words pick one value from a list and walk through that list as the pattern repeats. You lay the values down between `[ ]` brackets and the word hands back the one for this moment. If the value it lands on is a quotation, Cagire runs it. Otherwise it leaves it on the stack for the next word. Gathering is the job of the brackets: they run their contents immediately and collect everything they leave into one value, so `[ 60 64 67 ]` is a single list holding the three notes (see *Lists* (3.6)).

The cycling words

The cycling words come in pairs that differ only in what drives them. The plain forms count by `runs`: how many times this one step has played. The `p`-prefixed forms count by `iter` instead, which follows the pattern as a whole. That split is the subject of *The live model* (2.8). Here it is enough to remember which clock each pair follows, the step's own or the pattern's:

Word	Stack effect	What it does
`cycle`	`(list -- selected)`	walk the group forward, advancing by `runs`
`pcycle`	`(list -- selected)`	walk the group forward, advancing by `iter`
`rcycle`	`(list -- selected)`	walk the group backward, advancing by `runs`
`prcycle`	`(list -- selected)`	walk the group backward, advancing by `iter`
`bounce`	`(list -- selected)`	ping-pong up then down the group, by `runs`
`pbounce`	`(list -- selected)`	ping-pong up then down the group, by `iter`
`walk`	`(list -- selected)`	wander the group one place either way, by `runs`
`pwalk`	`(list -- selected)`	wander the group one place either way, by `iter`

`cycle` steps through the group in order and wraps at the end.

```
[ 60 64 67 ] cycle note
sine snd
.
```

Each time this step plays it advances one place, so it sounds `60`, then `64`, then `67`, then back to `60`. On the step's first run, with `runs` at zero, the first note is the one you hear. `bounce` turns at the ends instead of wrapping, so a four-value group reads `0`, `1`, `2`, `3`, `2`, `1`, `0`, `1` and on.

```
[ 60 64 67 72 ] bounce note
sine snd
.
```

The melody climbs `60 64 67 72` and walks back down `67 64 60` before climbing again: a pendulum rather than a loop.

`walk` takes a random step of one place each time, turning back at the ends the way `bounce` does. Where `bounce` always swings the full width, `walk` meanders, sometimes lingering in one corner of the group, sometimes crossing the whole of it.

```
[ 60 64 67 72 ] walk note
sine snd
.
```

It starts on the first value and from there drifts up or down at random, never jumping more than one place and never falling off either end. Because `walk` draws on the random generator, the same seed gives the same wander every time, so it is reseedable with `seed` (see *Random values* (3.13.1)). Its sibling `pwalk` takes its step once per pattern loop rather than once per run.

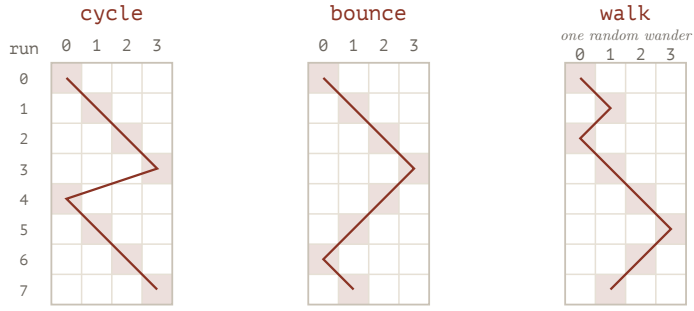


Figure 37: cycle, bounce, and walk stepping through a four-slot group as one step repeats; the filled slot on each row is the one selected.

index

`index` is `(list idx -- selected)`. Where the cycling words advance on their own, `index` lets you pick the slot yourself with an explicit number. The index wraps with modulo, so it can never fall off the group, and a negative index counts from the end.

```
[ c4 e4 g4 ] 1 index note
sine snd
.
```

Index `1` is the second value, so this plays `e4`.

```
[ c4 e4 g4 ] -1 index note
sine snd
.
```

Index `-1` is the last value, so this plays `g4`. Feed `index` a context word such as `step` or `iter` and the moment chooses the slot for you. See *The live model* (2.8).

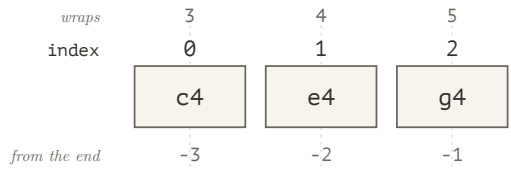


Figure 38: index picks a slot of [c4 e4 g4] by number; the index wraps by modulo and a negative index counts from the end.

Selecting a quotation

When a cycling word lands on a quotation, it runs it rather than pushing it. This lets each slot do its own work instead of carrying a single value.

```
[
   ( c4 note )
   ( e4 note )
   ( g4 note )
] cycle
sine snd
.
```

Each slot sets the note when it is reached, so `cycle` walks `c4 e4 g4` across the runs of the step. A quotation is the `( )` form, deferred code. See *Quotations & combinators* (2.7). For the random siblings of these words see *Choice* (3.12.3), and for the angle-bracket shorthand that expands to the forward, reverse, and ping-pong cycling words see *Alternation brackets* (3.12.2).

3.12.2 Alternation brackets

Alternation brackets are shorthand for the cycling words. Instead of laying out a group and naming a cycle word, you wrap the values in angle brackets and Cagire cycles them for you. A single pair advances once per pattern loop, like `pcycle`; a doubled pair advances once per step run, like `cycle`. See *Cycling words* (3.12.1) for the

words they stand in for and *The live model* (2.8) for the `runs` and `iter` counters behind them.

Single and double

`< ... >` is the single form, leaving one value from the group on the stack, `(-- value)`. It picks a fresh value each time the pattern loops, the same motion as `pcycle`.

```
< c4 e4 g4 > note sine snd .
```

The note changes once per lap around the pattern: `c4` on the first loop, `e4` on the next, then `g4`, then it wraps.

`<< ... >>` is the double form. More arrows, faster motion: it picks a fresh value each time this step fires within a loop, the same motion as `cycle`.

```
<< kick snare >> snd .
```

The source alternates every time the step plays, `kick` then `snare` then `kick`.

Direction

A dash on the opener turns the bracket into an arrow, and the arrow tells you which way the group is read. Forward is the default; `<-` reverses the group; `<-` together with a dashed closer `->` makes it ping-pong. A dash on the closer alone is only decoration.

Direction	Single (per loop)	Double (per run)
forward	`< a b c >`	`<< a b c >>`
reverse	`<- a b c >`	`<<- a b c >>`
ping-pong	`<- a b c ->`	`<<- a b c ->>`

Reverse walks the group from the end.

```
<- c4 e4 g4 > note sine snd .
```

It reads `g4 e4 c4` across the pattern loops, the descending form of `< c4 e4 g4 >`.

Ping-pong climbs and then falls back, the bracket form of `bounce`.

```
<<- kick snare hat ->> snd .
```

It walks `kick snare hat snare kick` as the step fires, turning at each end instead of wrapping.

Composing

Alternations are values like any other, so several in a row build a chord, each voice cycling on its own.

```
< 60 64 67 > < 72 76 79 > note sine snd .
```

Two voices move in lockstep: on each pattern loop the first sounds `60 64 67` while the second sounds `72 76 79`. An empty alternation, `< >` with nothing inside, is a compile error.

3.12.3 Choice

The cycling words are predictable. The choice words are not. They pick from a group at random every time the script runs. A script can keep surprising you. For the ordered siblings of these words see *Cycling words* (3.12.1), and for the `runs` and `iter` counters the ordered ones lean on see *The live model* (2.8).

Word	Stack effect	What it does
`choose`	`(list -- val)`	Pick one item at uniform random
`wchoose`	`(list -- selected)`	Pick one item, weighted by odds
`urn`	`(list -- selected)`	Shuffled pick without replacement, by step runs
`purn`	`(list -- selected)`	The same, advancing by pattern iteration

choose

`choose` is `(list -- val)`. It picks one value from the group with equal odds. This is a flat coin across however many you give it. As with the cycling words, the `[ ]` brackets are how you gather the items.

```
[ kick snare hat ] choose
snd
.
```

Each time the step plays, one of the three drums sounds, chosen uniformly. Run it again and you may well hear a different one.

wchoose

`wchoose` weights the odds: you pair each value with a weight, and Cagire favours the heavier values. Wrap the pairs in `[ ]` like every other choice word, alternating value then weight inside the brackets.

```
[ 60 0.6  64 0.3  67 0.1 ] wchoose note
sine snd
.
```

Here `60` is the likeliest note, `64` less so, and `67` rare. The weights are relative, so they need not add up to one; `6 3 1` would split the odds the same way. A negative weight is an error, and the weights cannot all be zero. Random picks reseed on every evaluation. To make a run repeatable, set `n seed` before the choice so the sequence comes out the same each time.

urn

`choose` is allowed to repeat itself; `urn` is not. It deals the group like a shuffled deck: each item comes out exactly once per pass, then Cagire reshuffles and deals again, and no item ever lands twice in a row, not even across the reshuffle. It takes its list in `[ ]` like every other choice word.

```
[ kick snare hat clap ] urn
snd
.
```

Across four runs of the step you hear all four drums, in an order that changes with every pass. `urn` advances with the step's `runs` counter; `purn` is the same deck driven by the pattern's `iter` counter,

matching the `p` twins of the cycling words. With two items the no-repeat rule leaves only one possibility, strict alternation. Unlike `choose`, the order is not fresh dice on every evaluation: it is derived from the counter and the current seed, so a restarted pattern replays its bags exactly, and `n seed` shuffles a different set of orders.

3.13 Randomness

3.13.1 Random values

Cagire draws fresh randomness on every evaluation, so a word that rolls a number gives a new result each time the step fires. These words leave a random value on the stack, where it can feed a parameter, a note, or anything else. To pick from a fixed set of values rather than a range see *Choice* (3.12.3); to gate whole quotations by chance see *Probabilistic execution* (3.13.2).

Word	Stack effect	What it does
`rand`	`(min max -- n|f)`	Uniform random number in range
`exprand`	`(lo hi -- f)`	Random float, biased toward `lo`
`logrand`	`(lo hi -- f)`	Random float, biased toward `hi`
`seed`	`(n --)`	Fix the random generator

rand

`rand (min max -- n|f)` draws a uniform random number between the two bounds. If both bounds are integers the result is an integer in the inclusive range. If either bound is a float the result is a float.

```
1 6 rand print
```

Prints a whole number from 1 to 6, different on each run. Write the bounds as floats for a continuous draw.

```
0.0 1.0 rand print
```

Prints a float somewhere between 0 and 1.

exprand and logrand

Both draw a float between two positive bounds, but they skew the result instead of spreading it evenly. `exprand` favours the low end of the range, `logrand` favours the high end. Both require positive bounds and raise an error on zero or a negative value.

Word	Stack effect	What it does
`exprand`	`(lo hi -- f)`	exponential draw biased toward `lo`
`logrand`	`(lo hi -- f)`	exponential draw biased toward `hi`

```
1.0 100.0 exprand print
```

Prints a float between 1 and 100, leaning toward the low bound.

```
1.0 100.0 logrand print
```

Prints a float between 1 and 100, leaning toward the high bound.

seed

`seed (n --)` fixes the random generator to a known starting point. Every draw made after a `seed` follows the same sequence, so a passage built on randomness repeats exactly. Reach for it when a happy accident is worth keeping.

```
12345 seed 1 6 rand print
```

Prints the same number every time you run it. Change the seed and you get a different, but equally repeatable, result.

3.13.2 Probabilistic execution

These words run, or skip, a piece of code by chance. Each one takes a quotation (deferred code written in `( )`, see *Quotations & combinators* (2.7)) and a probability, rolls a fresh random number, and runs the quotation only when the roll succeeds. The draw happens every evaluation, so the same line plays differently each time the step fires. For the raw numbers behind these decisions see *Random values* (3.13.1); for the deterministic counterpart that fires on a fixed cycle see *Periodic execution* (3.14.1).

Word	Stack effect	What it does
`coin`	`(-- bool)`	A 50/50 toss, `1` or `0`
`chance`	`(quot prob --)`	Run `quot` with probability 0.0 to 1.0
`prob`	`(quot pct --)`	Run `quot` with probability 0 to 100
adverbs	`(quot --)`	Named fixed probabilities, `always` down to `never`

coin

`coin (-- bool)` is the plainest source of chance, a 50/50 toss that pushes `1` or `0`. Feed it to `if` or any word that reads a boolean.

```
coin print
```

Prints `1` about half the time and `0` the other half.

chance and prob

`chance` and `prob` run a quotation with a probability you supply. They are one idea in two scales: `chance` takes a probability from 0.0 to 1.0, `prob` takes a percentage from 0 to 100.

Word	Stack effect	What it does
`chance`	`(quot prob --)`	run `quot` with probability `prob` (0.0 to 1.0)
`prob`	`(quot pct --)`	run `quot` with probability `pct` (0 to 100)

```
( sine snd . ) 0.75 chance
```

Plays the sine roughly three evaluations out of four. The same intent written in percent:

```
( sine snd . ) 75 prob
```

This plays the sine about 75% of the time.

Probability adverbs

The adverbs are named shortcuts for common probabilities. Each one is `(quot --)`: it takes a quotation and gates it like `chance` with

a fixed value, so `( ... ) often` is exactly `( ... ) 0.75 chance`. They read well in a line and spare you naming a number.

Word	Probability	Runs the quotation
`always`	1.0	every time
`almostAlways`	0.9	90% of the time
`often`	0.75	75% of the time
`sometimes`	0.5	half the time
`rarely`	0.25	25% of the time
`almostNever`	0.1	10% of the time
`never`	0.0	not at all

```
( sine snd . ) always
```

Plays the sine on every evaluation, because `always` is probability 1.0. Its opposite never fires:

```
( sine snd . ) never
```

This emits nothing at all.

3.14 Periodicity & timing

3.14.1 Periodic execution

These words gate a quotation by where the pattern is. Each one takes a quotation and a spec, then runs the quotation only at certain places in the loop, reading a live value to decide. They are how a part changes from loop to loop without you touching the code. See *The live model* (2.8) for the values they read. The quotation is deferred code written in `( )`; it runs only when the gate opens, and when it stays shut nothing happens and the stack is left clean. Most of these read `iter`, the number of times the whole pattern has looped. `bjork` reads `runs`, the number of times this one step has fired. `on` reads `step`, the position in the pattern rather than a loop count.

Word	Stack effect	What it does
`every`	`(quot n --)`	run the quotation on every `nth` pass: iterations `0`, `n`, `2n`, ...
`except`	`(quot n --)`	run it on every pass except those
`every+`	`(quot n offset --)`	the every-`nth` gate slid along by offset
`except+`	`(quot n offset --)`	every pass except the offset-shifted `nth`
`first`	`(quot n --)`	run it only on the first `n` passes
`after`	`(quot n --)`	run it from pass `n` onward
`once`	`(quot --)`	run it only on the very first pass
`on`	`(quot ..steps --)`	run it only on the named step positions, counting from 1
`bjork`	`(quot k n --)`	run it on a Euclidean gate of `k` hits over `n`, advanced by step runs
`pbjork`	`(quot k n --)`	the same Euclidean gate, advanced by pattern passes
`bjorkrot`	`(quot k n r --)`	the Euclidean gate rotated by `r` slots, advanced by step runs
`pbjorkrot`	`(quot k n r --)`	the rotated Euclidean gate, advanced by pattern passes

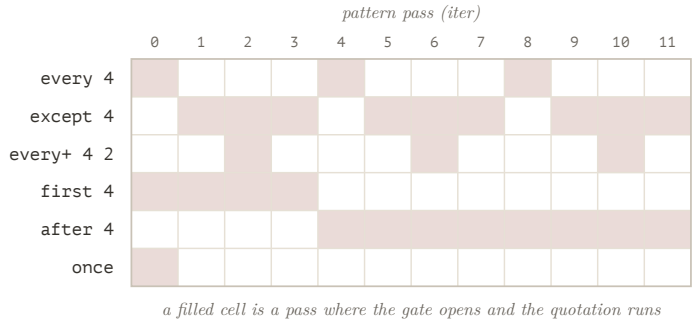


Figure 39: Which passes each gate opens across twelve loops read against iter; a filled cell is a pass where the quotation runs.

every and except

`every (quot n --)` runs the quotation when the iteration is a multiple of `n`, so the gate opens on passes `0`, `n`, `2n`, and so on.

```
( "kick" snd . ) 4 every
```

Plays the kick on every fourth pass of the pattern and stays silent in between.

`except (quot n --)` is the complement: it runs on every pass where `every` would not.

```
( "hat" snd . ) 4 except
```

Plays the hat on every pass except every fourth one, dropping out exactly where the `every` above fires.

Phase offset

`every+ (quot n offset --)` and `except+ (quot n offset --)` are `every` and `except` with the cycle slid along by offset, so you can stagger two parts that share a length.

```
( "snare" snd . ) 4 2 every+
```

Fires the snare on a four-cycle shifted two passes along, so it lands on iterations `2`, `6`, `10`, and so on.

Range and bounds

`first (quot n --)` runs the quotation only while the pattern is in its first n passes, and `after (quot n --)` runs it from pass n onward. Together they carve an intro and a later section out of one loop. `first` wants a count of at least `1`, while `after 0` is allowed and simply runs from the very first pass. `once (quot --)` is the narrowest of these: it fires on pass `0` and never again.

```
( "crash" snd . ) once
```

Plays the crash a single time, on the very first pass of the pattern.

By step position

`on (quot ..steps --)` is the one gate here that reads position rather than a loop counter. It takes a quotation and then a run of step numbers, and opens the gate whenever the playhead sits on one of them. The numbers count from 1, matching the step labels on the grid, so you name the steps you see. The quotation acts as the floor: `on` reads numbers off the stack until it reaches the quotation, so there is no count to keep in sync and nothing below the quotation is touched.

```
( "kick" snd . ) 1 4 9 on
```

Plays the kick only when the pattern is on step 1, 4, or 9, and stays silent on the rest. Reach for it when one script runs across many steps, a mirrored row or a single generative step, and you want it to pick its moments. For a fixed pattern it is usually plainer to place the sound on those steps directly and skip the gate.

Euclidean gates

`bjork (quot k n --)` opens the gate on a Euclidean rhythm of k hits spread as evenly as possible across `n` slots, advancing one slot per step run. `pbjork (quot k n --)` reads the same rhythm but advances one slot per pattern pass instead, so the gate moves at the pattern's pace rather than the step's.

```
( "hat" snd . ) 3 8 bjork
```

Plays the hat on a three-over-eight Euclidean rhythm. Which slot is live depends on the running step count, so the rhythm emerges as the step repeats. Ask for as many hits as there are slots, or more, and the gate opens on every one of them.

`bjorkrot (quot k n r --)` and `pbjorkrot (quot k n r --)` take a third argument that rotates the pattern by `r` slots, opening the gate on the same slots that an `r`-rotated `euclidrot` reports (see *Euclidean rhythms* (3.15.2)). A plain `bjork` is just `bjorkrot` with `r` at `0`, so reach for the rotated form when you want to slide the accents off the downbeat. The rotation has to be zero or positive.

```
( "hat" snd . ) 3 8 2 bjorkrot
```

Plays the same three-over-eight rhythm shifted two slots along, so the accents fall where a two-step rotation of the pattern would place them.

Looping a sample

`loop (n --)` is the odd one in this family. Instead of gating a quotation it fits the sample in the sound register across n steps, so one pass of the sample lines up with n steps of the pattern. It writes the playback length for you. See *The sound register* (2.5) for the register it fills.

```
"break" snd 16 loop .
```

Fits the break sample to span sixteen steps by changing its playback speed, then plays it, so the loop stays locked to the grid. To place a sound at a fractional position inside a single step rather than gating whole passes, see *Fractional timing* (3.14.2).

3.14.2 Fractional timing

`at` runs a quotation several times from a single step, placing each run at a precise point in time rather than firing once when the step does. Where the words in *Periodic execution* (3.14.1) gate whole passes of the pattern, `at` works below the step, so one step can spray a burst of hits. Each run starts from the sound register as it stood before `at`, so any parameters you set beforehand apply to every hit alike. The `runs` counter moves with the hits as well: every run inside `at` sees its own value, counting through this step's hits and carrying on as the step repeats, so a `cycle` or a `walk` written in the quotation advances from hit to hit instead of standing still.

Fractional offsets

`at (quot v..n --)` in its numeric form takes a quotation and one or more positions, each from `0` at the start of the step to `1` at its end, and runs the quotation once at each position within the step.

```
( "kick" snd . ) 0 0.5 at
```

Plays the kick at the start of the step and again halfway through, two hits in the span of one step. Add as many positions as you want and `at` places a run at each:

```
( "click" snd . ) 0 0.25 0.5 0.75 at
```

Plays four evenly spaced clicks across the one step. A list of positions works just as well as loose ones, so `[ 0 0.25 0.5 0.75 ] at` places the same four hits and anything that hands you a list, `gen` above all, feeds `at` directly (see *Lists* (3.6)).

From a pattern

`at` also takes a Pattern value in place of the numbers, and then it spreads the quotation across the hits of that pattern over the pattern's own span of steps. Build the value with `pat`, which turns a pattern string into hits. See *Pattern strings* (3.16.1) for the string mini-language.

```
( "hat" snd . ) "x.x.x.x." 8 pat at
```

Plays the hat on each of the pattern's four hits, spread across eight steps. A bare string will not do. `at` needs either numbers or a value built with `pat`, and it hands you an error if it finds a loose string instead.

3.15 Generators

3.15.1 Ranges

These words leave a whole run of numbers on the stack in one go, the raw material for melodies, offsets, and sweeps. Four of them build a fixed run from endpoints; `gen` builds one from code. For the siblings that turn into rhythms see *Euclidean rhythms* (3.15.2) and *Subdivision* (3.15.4).

Word	Stack effect	What it does
`..`	`(start end -- run)`	Integer run from start to end, inclusive
`.,`	`(start end step -- run)`	Run with a chosen step size
`geom..`	`(start ratio count -- run)`	Geometric run, multiplying by a ratio
`lin..`	`(start end count -- run)`	Linear run of `count` evenly spaced values, endpoints included
`gen`	`(quot n -- list)`	Run a quotation n times, collecting a list (see *Iteration* (3.10.3))

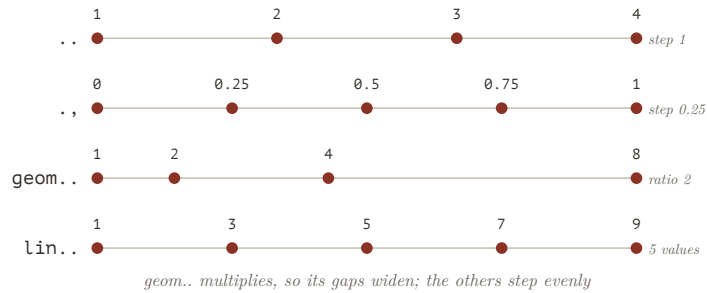


Figure 40: The four fixed range builders as number lines; the even ladders of .., ., and lin.. against the widening gaps of geom..

..

`..` `(start end -- run)` pushes every integer from start to end, inclusive, and counts down when start is above end.

```
1 4 ..
```

Leaves `1 2 3 4` on the stack. Written the other way round, `4 1 ..` leaves `4 3 2 1`.

.,

`., (start end step -- run)` is the same idea with a step size you choose, so the run can be fractional. A negative step counts down, and the end is included only when the step lands on it exactly.

```
0 1 0.25 .,
```

Leaves `0 0.25 0.5 0.75 1`. Whole values come back as integers, the rest as floats.

geom..

`geom.. (start ratio count -- run)` multiplies by a ratio instead of adding a step, pushing count values. Reach for it when you want octaves, harmonics, or an exponential sweep rather than an even ladder.

```
1 2 4 geom..
```

Leaves `1 2 4 8`, four values doubling each time.

lin..

`lin.. (start end count -- run)` is the linear cousin of `geom..`: instead of a ratio it spreads `count` values evenly from start to end, with both endpoints landing exactly. Where `.,` asks you for the step and reaches the end only when that step divides the span exactly, `lin..` asks how many values you want and works out the spacing for you. It counts down when end is below start.

```
1 9 5 lin..
```

Leaves `1 3 5 7 9`, five values from `1` to `9`. `0 1 5 lin..` leaves `0 0.25 0.5 0.75 1`, splitting a `0` to `1` span into even quarters.

gen

`gen (quot n -- list)` belongs to this family too: it runs a quotation n times and collects the results, so the run is computed rather than fixed. It is the one builder here that hands its run back

as a list instead of as loose values; wrap any of the others in `[ ]` to get the same shape (see *Lists* (3.6)). It is documented with the other quotation-runners in *Iteration* (3.10.3).

3.15.2 Euclidean rhythms

A Euclidean rhythm spreads `k` hits as evenly as it can across `n` steps, the spacing behind a great deal of today's electronic music. These two words hand you the hit positions as a run of step indices, ready to test against the current `step` or to pick from with `index`. For the related range and slicing words see *Ranges* (3.15.1) and *Subdivision* (3.15.4). For hits chosen by where the bar's weight falls rather than by even spacing, see *Barlow rhythms* (3.15.3).

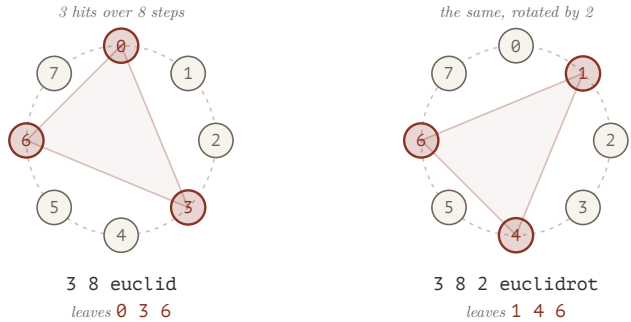


Figure 41: Three hits spread as evenly as the maths allows over eight steps land on 0, 3, 6; euclidrot shifts them to 1, 4, 6.

Word	Stack effect	What it does
`euclid`	`(k n -- indices)`	k hits spread evenly over `n` steps
`euclidrot`	`(k n r -- indices)`	The same pattern rotated by `r`

euclid

`euclid (k n -- i1 i2 ... ik)` pushes the indices of `k` hits spread over n steps, counting from `0`.

```
4 8 euclid
```

Leaves `0 2 4 6`, four hits on the even steps of eight. Ask for an uneven split and Cagire spaces it as evenly as the maths allows: `3 8 euclid` leaves `0 3 6`.

euclidrot

`euclidrot` `(k n r -- i1 i2 ... ik)` is the same pattern rotated `r` places, so the run starts somewhere other than the first hit.

```
3 8 2 euclidrot
```

Leaves `1 4 6`, the `3 8` pattern shifted along by two. The indices are plain step numbers, so a common move is to compare them against the current `step`, or to pick a value at one of them with `index` (see *Cycling words* (3.12.1)). To run a quotation on a Euclidean schedule directly rather than collecting indices, reach for `bjork` in *Periodic execution* (3.14.1).

3.15.3 Barlow rhythms

Where a Euclidean rhythm spreads its hits as evenly as it can and forgets what bar it is in, a Barlow rhythm asks a different question: if you may only play `k` of these `n` steps, which ones does the metre least want you to leave out? Clarence Barlow called the answer a pulse's *indispensability*. These words rank every step in a bar and hand you the strongest ones. For the evenly-spread alternative see *Euclidean rhythms* (3.15.2).

Word	Stack effect	What it does
`barlow`	`(k n -- indices)`	The `k` most indispensable of `n` steps
`barlowg`	`(k n g -- indices)`	The same, with `g` steps to a beat
`barlowv`	`(n -- weights)`	The metric weight of every step, 1 down to 0

Three rules build the ranking. The downbeat is always first. Within any level the later beats outrank the earlier ones, because a beat's job is to be the run-up to the next strong one. And every offbeat inherits the rank of the beat it leads into.

barlow

`barlow (k n -- i1 i2 ... ik)` pushes the indices of the `k` strongest of `n` steps, counting from `0`.

```
3 8 barlow
```

Leaves `0 4 6`, beats one, three and four of a bar of eight, which is the order a drummer thins a pattern down to. `3 8 euclid` leaves `0 3 6` instead, three hits at equal distance and no particular relationship to the bar.

Two things follow from ranking rather than spacing. Each density contains the one below it, so raising `k` thickens a pattern instead of rearranging it:

```
2 16 barlow    ;; 0 8
3 16 barlow    ;; 0 8 12
4 16 barlow    ;; 0 4 8 12
```

That makes `k` a busyness control you can sweep, which `euclid` cannot offer. Its hits move every time you change the count.

barlowg

`barlowg (k n g -- i1 i2 ... ik)` takes `g`, the number of steps in one beat, and uses it to choose the metre. Twelve steps is genuinely two different bars, and only this word can tell them apart:

```
5 12 4 barlowg    ;; 0 4 6 8 10   three beats of four
5 12 3 barlowg    ;; 0 3 6 9 11   four beats of three
```

`barlow` is the same word with `g` left at `0`, which lets the factorisation decide and puts the coarsest division outermost. A `g` that does not divide `n` falls back to the same automatic choice rather than failing.

barlowv

`barlowv (n -- w1 w2 ... wn)` pushes one weight per step instead of a selection: `1.0` on the downbeat, falling to `0.0` on the weakest pulse in the bar.

```
4 barlowv
```

Leaves `1 0 0.667 0.333`. This is the metric hierarchy as plain numbers, so it drives anything that takes one. Accent a line by velocity, bias a probability, open a filter further on the strong steps. Pick the weight for the current step with `index` (see *Cycling words* (3.12.1)) rather than consuming the whole run.

Where it stops

The ranking is exact for bars that divide into twos and threes, which covers 4, 6, 8, 9, 12, 16, 24 and 32 steps. Bars whose length has a prime factor of five or more (5, 7, 10, 14, 15) fall back to plain descending order within that level and give noticeably worse results, clumping hits at the end of the bar. Reach for `euclid` there instead.

Rotation is deliberately absent. Barlow's ranking is a statement about where the downbeat is, and rotating it away removes the reason to prefer it over a Euclidean pattern. If you want the pattern moved anyway, build it into a pattern string and use `prot` (see *Pattern transforms* (3.16.2)).

3.15.4 Subdivision

These words slice the current step into even fractions and leave those fractions on the stack, so a single step can fire several times. The fractions run from `0` (the start of the step) up to but not including `1`, which is exactly what `at` wants. Feeding one of these to `at` is the usual way to ratchet or roll a hit. See *Fractional timing* (3.14.2). For the related run-builders see *Ranges* (3.15.1).

Word	Stack effect	What it does
`div`	`(n -- offsets)`	n evenly spaced offsets across the step
`swing`	`(n ratio -- offsets)`	n offsets with the off-beats pushed late

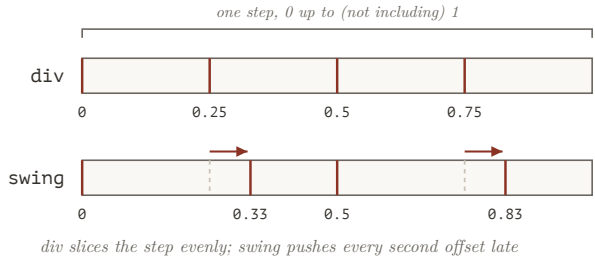


Figure 42: div slices the step into evenly spaced offsets; swing keeps the on-beats and pushes every second offset late, ghost ticks marking the straight positions.

div

`div (n -- f1 f2 ...)` pushes `n` evenly spaced offsets across the step.

```
4 div
```

Leaves `0 0.25 0.5 0.75`, the start, quarter, half, and three-quarter points of the step. Hand them to `at` and the step plays a sound at each one.

```
( hat snd . ) 4 div at
```

Plays the hat four times across the step, evenly spaced.

swing

`swing (n ratio -- f1 f2 ...)` slices the step like `div`, but pushes every second offset late by the ratio, so the run swings. A ratio of `0.5` is straight. Higher values drag the off-beats later.

```
4 0.66 swing
```

Leaves `0 0.33 0.5 0.83`, the off-beats nudged late. With `4 0.5 swing` you get `0 0.25 0.5 0.75`, exactly what `div` gives.

3.16 Patterns

3.16.1 Pattern strings

A pattern string is a rhythm written as text. `pat` reads one and builds a pattern value, which you then play by firing a quotation at every hit. The string says where the hits fall. The step count you give `pat` decides how wide they spread.

Building a pattern

`pat ("pattern" n -- pat)` takes a pattern string and a step count and leaves a pattern value. The string is read left to right and stretched to fill `n` steps, so the same string plays wider or tighter depending on `n`.

```
"x.x." 4 pat
```

This leaves a pattern of two hits, at the first and third of four slots, spread over four steps. Nothing sounds yet: a pattern value is inert until you run it.

Running a pattern

Put a quotation before the pattern and hand both to `at`. Cagire fires the quotation once at each hit, offset to the hit's position in the bar. The same `at` also takes plain fractional offsets, covered in *Fractional timing* (3.14.2).

```
( "kick" snd . ) "x.x." 4 pat at
```

Two kicks, one on the downbeat and one halfway through (this uses `snd` and `.`, see *The sound register* (2.5)).

The mini-language

A pattern string is built from these symbols. Whitespace is ignored, so you can space slots out for readability.

Symbol	What it does
`x`	a hit

Symbol	What it does
`.`	a rest
`-`	extend the previous hit over one more slot
`*N`	repeat the preceding token `N` times
`[ ]`	subdivide one slot into a packed group
`<a|b>`	alternate between segments, one per run

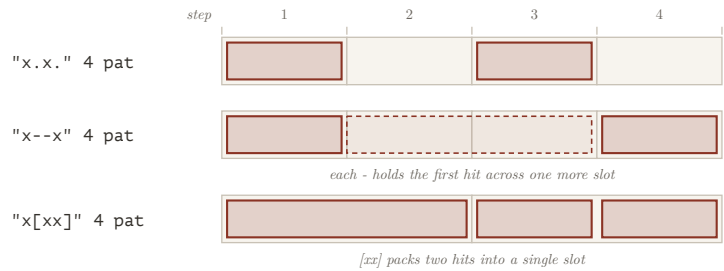


Figure 43: Three pattern strings laid onto the same four steps, showing how hits, rests, held notes, and bracketed groups map to slots.

`-` lengthens the gate of the hit before it rather than adding a new hit, so a held note rings across the slots it covers.

```
( "kick" snd . ) "x--x" 4 pat at
```

Two kicks: the first holds across three slots, the second lands on the fourth.

`[ ]` packs a group of hits into the space of a single slot, which is how you get faster subdivisions inside a slower pulse.

```
( "hat" snd . ) "x[xx]" 2 pat at
```

Three hats: one on the beat, then a pair squeezed into the second slot.

`*N` repeats the token in front of it, so `x*4` is shorthand for four hits in a row.

```
( "click" snd . ) "x*4" 4 pat at
```

Four evenly spaced clicks.

`<a|b>` alternates between its segments across runs: the first time the step fires it plays the first segment, the next time the second,

then it wraps back. At the start, before the step has repeated, you hear the first segment.

```
( "kick" snd . ) "<x.|.x>" 1 pat at
```

On the first run this plays the `x.` segment, a kick on the downbeat. On the next run it would play `.x` instead.

A string Cagire cannot read stops the line rather than playing something approximate. An unrecognised character, a `-` with no hit in front of it, a `|` outside a `< >` group, and a bracket left open are all errors.

3.16.2 Pattern transforms

These three words reshape a pattern string before you build it. Each takes a string and leaves a string, so you slot them between the literal and `pat`. The symbols they rearrange are the ones from *Pattern strings* (3.16.1).

Word	Stack effect	What it does
`prot`	`("pattern" n -- "pattern")`	rotate the slots right by `n`, negative rotates left
`prev`	`("pattern" -- "pattern")`	reverse the slot order
`pinv`	`("pattern" -- "pattern")`	invert: hits become rests, rests become hits

`prot` and `prev` work on top-level slots, so a `[ ]` or `< >` group moves as one unit and stays intact. `prot` shifts the whole string along by `n` slots.

```
"x..x" 1 prot print
```

This prints `xx..`: rotating `x..x` right by one carries the trailing hit around to the front.

`prev` flips the slot order end to end.

```
"xx.." prev print
```

This prints `..xx`.

`pinv` swaps hits and rests character by character, and a hold `-` becomes a hit too.

```
"x..x" pinv print
```

This prints `.xx.`: the two hits turn into rests and the two rests into hits.

Because each transform returns a string, you chain it straight into `pat` and play the result.

```
( "snare" snd . ) "x..." 1 prot 4 pat at
```

`x...` rotated right by one becomes `.x..`, so the snare lands on the second slot.

3.17 Pitch & harmony

3.17.1 Notes & intervals

Cagire can read note names and intervals as ordinary words. These words will translate immediately to numbers, either MIDI integers or floats. A note name is immediately replaced by a number. An interval name adds a second number above or below the note already on the stack, allowing you to build chords by stacking intervals. Because notes are just numbers, you can do arithmetic on them and feed them to `note` (see *The sound register* (2.5)).

A note name is a letter, an optional accidental, and an octave. The letters follow the English naming standard: `c d e f g a b`. The octave runs from `0` to `9`, and `c4` is middle C at MIDI `60`. The accidental sits between the letter and the octave: `#` or `s` sharpens, `b` flattens. Octave `-1`, the bottom of the MIDI range, only reads as a note when the name carries an accidental, so `cs-1` gives `1` while `c-1` stays a plain string. Case does not matter, so `C4` and `c4` are the same note.

```
c4 print
```

This program prints `60`: middle C as a MIDI number. `cs4` and `c#4` both give `61`, and `db4` gives `61` too.

The French solfège names work the same way: `do re mi fa sol la si` (with `ti` for `si` and `ut` as a second name for `do`). They take an octave and an accidental, with `#` for sharp and `b` for flat, and an

accented `ré` reads the same as `re`. The following program prints `67`, the G above middle C:

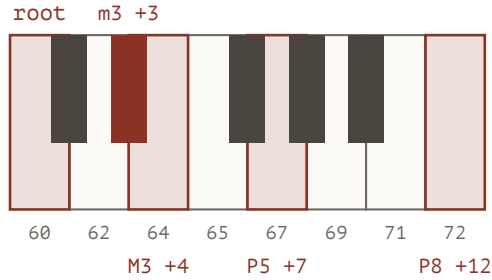
```
sol4 print
```

Interval names

An interval name compiles to `dup <semitones> +`, so it duplicates the note on top of the stack and raises the copy by that many semitones. The result is two notes, the original and the one an interval above it, ready to feed `note` as a two-voice chord.

```
a3 m3
```

This leaves `57` and `60` on the stack, an A and the minor third above it. Stacking another interval acts on whichever note is now on top, not on the original root. To spell a full chord from a root in one move, reach for `chord` in *Chords* (3.17.3).



an interval keeps the note and adds one that many semitones above

Figure 44: Intervals as semitone distances from a root over one octave of keys, from $C_4 = 60$ to $C_5 = 72$.

Word	Stack effect	Interval
`P1` (`unison`)	`(n -- n n)`	perfect unison
`m2`	`(n -- n n+1)`	minor second
`M2`	`(n -- n n+2)`	major second
`m3`	`(n -- n n+3)`	minor third

Word	Stack effect	Interval
`M3`	`(n -- n n+4)`	major third
`P4`	`(n -- n n+5)`	perfect fourth
`aug4` (`dim5`)	`(n -- n n+6)`	tritone
`P5`	`(n -- n n+7)`	perfect fifth
`m6`	`(n -- n n+8)`	minor sixth
`M6`	`(n -- n n+9)`	major sixth
`m7`	`(n -- n n+10)`	minor seventh
`M7`	`(n -- n n+11)`	major seventh
`P8`	`(n -- n n+12)`	octave
`m9`	`(n -- n n+13)`	minor ninth
`M9`	`(n -- n n+14)`	major ninth
`m10`	`(n -- n n+15)`	minor tenth
`M10`	`(n -- n n+16)`	major tenth
`P11`	`(n -- n n+17)`	eleventh
`aug11`	`(n -- n n+18)`	augmented eleventh
`P12`	`(n -- n n+19)`	twelfth
`m13`	`(n -- n n+20)`	minor thirteenth
`M13`	`(n -- n n+21)`	major thirteenth
`m14`	`(n -- n n+22)`	minor fourteenth
`M14`	`(n -- n n+23)`	major fourteenth
`P15`	`(n -- n n+24)`	double octave

The six-semitone interval is `aug4` or `dim5`; `tritone` on its own is the scale of that name, covered in *Scales & modes* (3.17.2), not the interval. Unlike note names, interval names are read exactly as written, so the case carries the meaning: `m3` is the minor third and `M3` the major one.

Shifting by octaves

`oct` is `(note shift -- note)`. It raises the note on top of the stack by `shift` whole octaves, a readable alternative to adding multiples of `12` by hand.

```
c4 1 oct print
```

This prints `72`, middle C moved up one octave.

3.17.2 Scales & modes

A scale word pushes a scale value built on the standard twelve-tone tuning. You turn a scale into pitch with `deg`, which reads a root, a scale, and a degree and leaves a note, and you pull outside notes onto the scale with `snap`. `mode` rotates a scale into one of its modes, and you can also build your own scale from a tuning (see *Tuning & microtonality* (3.17.4)).

Built-in scales

Each of these is a bareword with the stack effect `(-- scale)`; it pushes a scale value whose degrees are the semitone offsets listed below.

Word	Degrees
`major`	0 2 4 5 7 9 11
`minor`	0 2 3 5 7 8 10
`dorian`	0 2 3 5 7 9 10
`phrygian`	0 1 3 5 7 8 10
`lydian`	0 2 4 6 7 9 11
`mixolydian`	0 2 4 5 7 9 10
`locrian`	0 1 3 5 6 8 10
`pentatonic`	0 2 4 7 9
`minpent`	0 3 5 7 10
`blues`	0 3 5 6 7 10
`chromatic`	0 1 2 3 4 5 6 7 8 9 10 11
`wholetone`	0 2 4 6 8 10
`harmonicminor`	0 2 3 5 7 8 11
`melodicminor`	0 2 3 5 7 9 11
`bebop`	0 2 4 5 7 9 10 11
`bebopmaj`	0 2 4 5 7 8 9 11
`bebopmin`	0 2 3 5 7 8 9 10
`altered`	0 1 3 4 6 8 10
`lyddom`	0 2 4 6 7 9 10

Word	Degrees
`halfwhole`	0 1 3 4 6 7 9 10
`wholehalf`	0 2 3 5 6 8 9 11
`augmented`	0 3 4 7 8 11
`tritone`	0 1 4 6 7 10
`prometheus`	0 2 4 6 9 10
`dorianb2`	0 1 3 5 7 9 10
`lydianaug`	0 2 4 6 8 9 11
`mixb6`	0 2 4 5 7 8 10
`locrian2`	0 2 3 5 6 8 10

`minor` also answers to `aeolian`: the natural minor scale and the aeolian mode are the same degrees, so Cagire keeps one scale under two names.

Resolving a degree

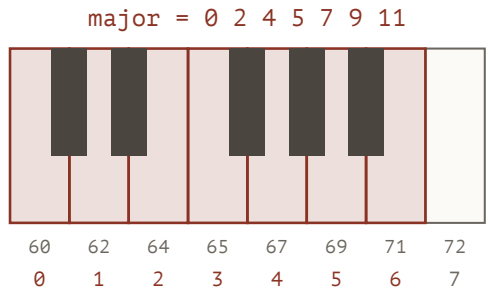
`deg` is `(root scale degree -- note)`. It reads a root MIDI note, a scale, and a degree index, and leaves the note of that degree. Degrees wrap past the top of the scale and climb by octaves, so degree `7` of a seven-note scale is the root an octave up.

```
60 major 0 deg print
```

This prints `60`, middle C itself, since degree `0` of C major is the root. A degree is a note like any other: send it to `note`, shift it with `oct` or plain arithmetic, stack intervals on it, or make it the root of a `chord`.

```
60 major 2 deg note sine snd .
```

This plays `64`, the E that is the third degree of C major.



deg picks a degree by index; mode rotates which degree begins the scale

Figure 45: C major laid out as degrees across the octave, where deg picks a degree by index and mode rotates which degree begins the scale.

Snapping to a scale

`snap` is `(root scale note -- note)`. It reads the same root and scale as `deg`, then a note from anywhere, and moves that note to the nearest scale tone. Notes already in the scale pass through untouched, a note exactly halfway between two tones resolves upward, and distance is measured across octaves, so any register snaps against the same scale.

```
60 major 63 snap print
```

This prints `64`: the D sharp halfway between D and E resolves up to E.

Snapping is what keeps generated material in key. Any word that leaves a number can feed it.

```
60 minpent c3 c5 rand snap note sine snd .
```

This picks a random note across two octaves and pulls it onto C minor pentatonic before it plays.

Modes

`mode` is `(n scale -- scale)`. It rotates a scale's degree ordering by `n`, turning C major into its modes without spelling them out by hand. The rotated scale behaves like any other under `deg`.

```
60 1 major mode 0 deg print
```

This prints `62`, a D: rotating C major by one starts the scale on its second degree.

Building a scale

Build a custom scale by selecting tuning steps with `[ degrees ] N edo scale`, then resolve it with `deg`. The full machinery of `edo`, `tuning`, and `scale` lives in *Tuning & microtonality* (3.17.4).

```
60 [ 0 2 4 6 8 10 ] 12 edo scale 1 deg print
```

This builds a whole-tone scale from twelve-tone tuning and prints `62`, the D a whole tone above the C root.

3.17.3 Chords

`chord` names a chord quality and stores it in the sound register. The chord is not spelled out until you play: at emit time Cagire takes the root from the `note` parameter and stacks the quality's intervals above it, so one `note` and one `chord` give you a full voicing. `anchor` and `cn` shape how that voicing comes out.

Setting a quality

`chord` is `(quality --)`. It takes a quality name and sets it as the active chord. The root comes from `note`, so set the note first and leave nothing loose on the stack. At emit, the register fans out into one voice per chord tone.

```
c4 note
maj7 chord
sine snd
.
```

This plays a C major seventh: four sines at `60`, `64`, `67`, and `71`.

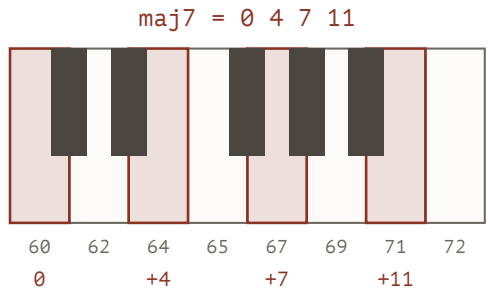


Figure 46: A chord quality stacks its intervals on the root: maj7 (0 4 7 11) on c4 fans into keys 60 64 67 71.

Quality	Aliases	Intervals
maj	M	0 4 7
m	min	0 3 7
dim	o	0 3 6
aug		0 4 8
sus2		0 2 7
sus4	sus	0 5 7
maj7	Δ7	0 4 7 11
min7		0 3 7 10
dom7	7	0 4 7 10
dim7	o7	0 3 6 9
m7b5	min7b5 h7 hdim7 halfdim7 ø7	0 3 6 10
minmaj7	mM7 mmaj7	0 3 7 11
aug7		0 4 8 10
maj6	6	0 4 7 9
min6		0 3 7 9
dom9	9	0 4 7 10 14
maj9		0 4 7 11 14
min9		0 3 7 10 14
dom11	11	0 4 7 10 14 17

Quality	Aliases	Intervals
maj11	M11	0 4 7 11 14 17
min11	m11	0 3 7 10 14 17
dom13	13	0 4 7 10 14 21
maj13		0 4 7 11 14 21
min13		0 3 7 10 14 21
add9	add2	0 4 7 14
add11	add4	0 4 7 17
madd9	madd2	0 3 7 14
pwr	5 power	0 7
7sus4	sus47	0 5 7 10
9sus4	sus49	0 5 7 10 14
augmaj7	+M7 +maj7	0 4 8 11
maj69	M69 6/9	0 4 7 9 14
min69	m69 m6/9	0 3 7 9 14
dom7b9	7b9	0 4 7 10 13
dom7s9	7s9 7#9	0 4 7 10 15
dom7b5	7b5	0 4 6 10
dom7s5	7s5 7#5	0 4 8 10
dom7s11	7s11 7#11	0 4 7 10 18

Watch the single-letter shorthand: names like `m7` and `M9` are the interval words of *Notes & intervals* (3.17.1), not chord qualities, which is why they are absent from the alias column. Spell those qualities out as `min7`, `maj9`, and so on.

Anchoring a voicing

`anchor` is `(note --)`. It sets a target note, and Cagire picks the inversion and octave that sits nearest the anchor instead of stacking the chord straight up from the root.

```
g4 anchor c4 note maj7 chord sine snd .
```

This voices the same C major seventh near G_4, coming out as `64`, `67`, `71`, `72`. Don't sleep on `anchor`, it is the key to easily write

nice arpeggios or invert chords extremely easily. It can add a lot of movement to your harmonic lines.

Picking one tone

`cn` is `(index --)`. It selects a single tone of the chord by index, counting from `0`. The index wraps around the chord and shifts by an octave each time it passes the top, so you can arpeggiate a chord by feeding `cn` a changing index.

```
c4 note maj7 chord 1 cn sine snd .
```

This plays only the second tone of Cmaj7, the E at `64`.

Numeric qualities

The number aliases `5 6 7 9 11 13` stand for power, sixth, dominant seventh, ninth, eleventh, and thirteenth chords. Because the number sits on the stack, the root must be in the `note` parameter, not loose alongside it.

```
c4 note 7 chord sine snd .
```

This plays a C dominant seventh: `60`, `64`, `67`, `70`. For stacking individual intervals by hand instead of a named quality, see *Notes & intervals* (3.17.1); for how the register fans several notes into voices, see *The sound register* (2.5).

3.17.4 Tuning & microtonality

There are specialized words for moving between pitch units and for building tunings of your own. `mtof` and `ftom` convert between MIDI notes and frequency, `edo` divides the octave into equal steps, `tuning` builds a temperament from cents, and `scale` picks degrees out of a tuning. Together they take you past twelve-tone equal temperament into any tuning you can describe. It is recommended to register user words with your favorite scales and tunings. That way, you will never need to define them again and you will be able to use them in all your sessions.

Converting pitch units

`mtof` is `(midi -- hz)` and `ftom` is `(hz -- midi)`; they are inverses of each other. Reach for them when a value is in one unit and a parameter or calculation wants the other.

```
69 mtof print
```

This prints `440`, the frequency of MIDI note `69`, concert A. Running `440 ftom print` takes you back to `69`.

Dividing the octave

`edo` is `(n -- tuning)`. It builds an equal-division tuning over the `2/1` octave, leaving a tuning value of `n` equally spaced steps. `12 edo` is the familiar chromatic tuning of twelve `100`-cent steps. Raise `n` for finer microtonal grids, so `24 edo` gives quarter-tones.

```
60 [ 0 1 2 ] 24 edo scale 1 deg note sine snd .
```

This builds a three-note scale out of 24-tone tuning and plays degree `1`, the note `60.5`, a quarter-tone above middle C. When a tuning lands between semitones, `deg` leaves a fractional note, and the engine plays it as readily as a whole one.

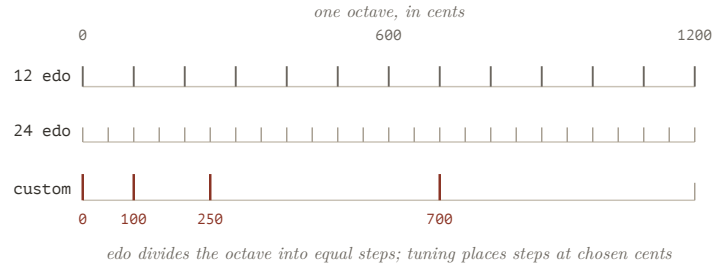


Figure 47: edo cuts the 1200-cent octave into equal steps; tuning instead places steps at cents you choose.

Building a tuning from cents

`tuning` is `(list period -- tuning)`. It builds a tuning from explicit cents offsets within a period, also in cents. The opening `0`

is implied, the values must rise strictly and stay below the period, and `1200` is one octave.

```
[ 100 250 700 ] 1200 tuning
```

This leaves a tuning whose steps are `0`, `100`, `250`, and `700` cents inside a `1200`-cent octave.

Selecting degrees

`scale` is `(list tuning -- scale)`. It chooses which steps of a tuning belong to a scale by their index, leaving a scale value. Resolve that scale to pitch with `deg`, covered in *Scales & modes* (3.17.2).

```
[ 0 1 2 ] [ 100 250 700 ] 1200 tuning scale
```

This selects the first three steps of the custom tuning above, leaving a three-note microtonal scale ready for `deg`.

3.18 Motion

3.18.1 Shape words

A parameter given a number holds it. Give it a motion and it moves for the whole event. The words here each build one shape with its arguments already arranged, so a filter sweep is `200 4000 2 lfo lpf` and nothing more.

Every time on this page is in steps and follows the tempo. *Motions* (3.18.2) holds the vocabulary underneath these words, for shapes they do not cover. *Motion notation* (3.18.3) writes the same shapes as text. *The engine* (IV) covers what each one does to the sound.

LFOs

`lfo (min max period -- m)` oscillates as a sine between `min` and `max`, one cycle per period.

```
saw snd 200 4000 2 lfo lpf .
```

The cutoff rides the sine from 200 up to 4000 Hz and back, two steps per round trip.

The sine is the only shape with a word of its own. Build the others with `osc`, which takes a shape name and a cycle length in seconds. `stepsecs` converts steps into seconds:

```
saw snd tri 2 stepsecs osc 200 4000 range lpf .
```

The same two-step sweep, drawn as a triangle. Write `saw` or `square` in place of `tri`. *Motions* (3.18.2) lists all six shapes.

Slides

`slide (start end dur -- m)` travels once from `start` to `end` over `dur`, then holds. The travel is linear.

```
sine snd 0 1 0.5 slide gain .
```

The gain ramps from silence to full over half a step.

No word covers the other curves. Build them by hand or write them as a notation suffix. `1 stepsecs rise 200 4000 expmap` is the ratio glide a filter sweep usually wants, `1 curve` bends a rise into a swell, `8 levels` cuts it into a staircase. *Motions* (3.18.2) covers all three. A travel can take six shapes in all: linear, plus the five the notation spells with a suffix, so `"200>4000:2e"` writes the exponential one in a single token. See *Motion notation* (3.18.3).

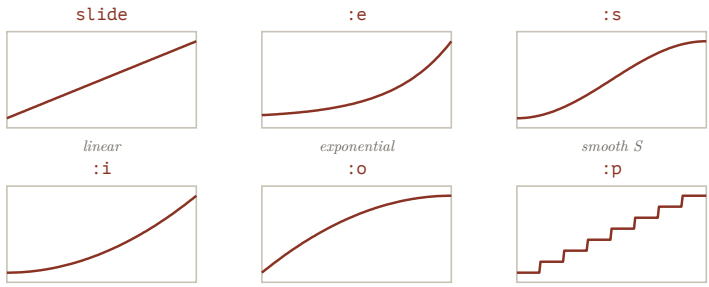


Figure 48: The six curves a travel can take from start to end, from linear to the eight-step stair; slews use the same shapes.

Slews

`slew (target dur -- m)` starts from wherever the parameter already sits and travels to `target` over `dur`, linearly. The curves work as they do for a slide, with `current` standing in for the start. `1 stepsecs rise current 4000 expmap` is the exponential slew, `">4000:1e"` the notation for it.

```
sine snd
0.5 1 slew gain
.
```

The gain slews from its current value toward `0.5` over one step.

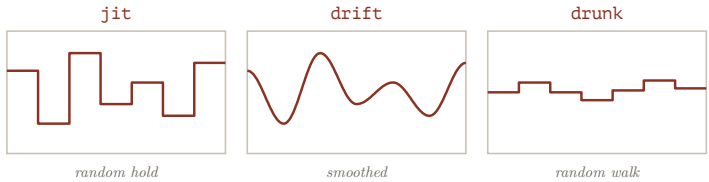
Random motion

`jit (min max period -- m)` jumps to a fresh random value between `min` and `max` every period. `drunk` walks to a neighbour each period. Neither smooths its jumps, but the `drift` shape does: `drift 0.5 stepsecs osc 200 4000 range` glides between the levels `jit` lands on.

Word	Stack effect	What it does	Example
`jit`	`(min max period -- m)`	Random hold	`200 4000 0.5 jit lpf`
`drunk`	`(min max period -- m)`	Drunk walk	`200 4000 0.5 drunk lpf`

```
saw snd 200 4000 0.5 jit lpf .
```

The cutoff lands on a new random value in 200 to 4000 Hz every half step.



t jumps to a fresh value each period, drunk steps only to a neighbour, a drift oscillation glides between jum

Figure 49: The three random motions: jit jumps to a fresh value each period, drunk wanders to a neighbour, a drift oscillation smooths the jumps into a glide.

Envelopes

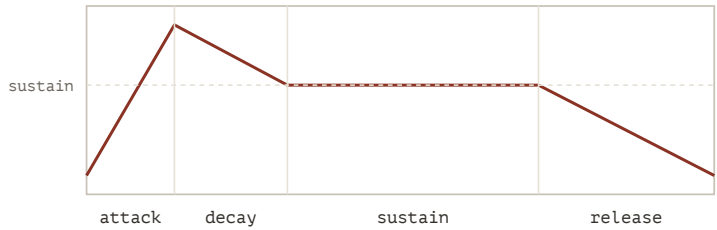
`ead (min max a d -- m)` sweeps from `min` to `max` and back on a percussive attack and decay. `eadr` adds a release stage, `eadsr` is the full ADSR. `env` is another name for `eadsr`.

Word	Stack effect	What it does	Example
`ead`	`(min max a d -- m)`	Attack, decay	`200 8000 0.01 0.1 ead lpf`
`eadr`	`(min max a d r -- m)`	Attack, decay, release	`200 8000 0.01 0.1 0.2 eadr lpf`
`eadsr` (`env`)	`(min max a d s r -- m)`	Full ADSR	`200 8000 0.01 0.1 0.5 0.2 eadsr lpf`

```
saw snd
200 8000 0.01 0.1 ead lpf
.
```

The cutoff snaps from 200 to 8000 Hz over a brief attack and falls back over the decay.

These four stages are all a modulation envelope has. The voice's own amplitude envelope runs six, adding a delay and a hold in front (see *Voice envelope* (6.1)).



ead traces attack then decay; eadr adds the release; eadsr (env) is the full four-stage shape

Figure 50: The ADSR shape the ead family sweeps a parameter through: rise over attack, fall to sustain over decay, hold, then return over release.

Low-pass gate

`lpg` leaves nothing on the stack. It writes straight into the register, pairing the amplitude envelope you have set with a matching lowpass sweep. *Lowpass / bandpass / highpass* (7.2.1) covers it in full.

3.18.2 Motions

A motion is a parameter value that keeps moving after the event starts. You build one on the stack and hand it to a parameter where a number would go, and Cagire plays that curve for the whole event.

Every source here gives a value between `0` and `1`, so `range` ends almost every line below. A plain number is a motion that never moves, and the two mix freely in the same arithmetic. Durations here are in seconds, not steps.

Shape words (3.18.1) packages these into ready-made shapes and *Motion notation* (3.18.3) writes them as text. This page is the vocabulary underneath both.

Word	Stack effect	What it does
`osc`	`(shape len -- m)`	LFO on any shape, one cycle per `len` seconds
`rise`	`(len -- m)`	One-shot ramp `0` to `1` over `len` seconds, then hold
`current`	`(-- m)`	The parameter's value when the event arrived

Word	Stack effect	What it does
`range`	`(m min max -- m)`	Map a `0-1` motion onto `min-max`
`expmap`	`(m lo hi -- m)`	Exponential map of a `0-1` motion onto `lo-hi`
`smooth`	`(m len -- m)`	Lag a motion by `len` seconds
`levels`	`(m n -- m)`	Quantise a motion to `n` evenly spaced values
`curve`	`(m k -- m)`	Bend a motion: `k` above `0` starts slow, below `0` starts fast
`above`	`(m t -- m)`	`1` above the threshold `t`, else `0`
`later`	`(m secs -- m)`	Shift a motion by `secs`, negative for earlier
`retrig`	`(m -- m)`	Count from the note instead of the transport
`repeat`	`(m -- m)`	Loop a one-shot motion instead of holding
`+ - * min max`	`(m m -- m)`	Arithmetic, on motions or numbers
`sample`	`(m -- n)`	Read a motion as a plain number
`stepsecs` / `ss`	`(steps -- secs)`	Convert steps to seconds at the current tempo
`beatsecs` / `bs`	`(beats -- secs)`	Convert beats to seconds at the current tempo

```
saw snd sine 0.5 osc 200 4000 range lpf .
```

`sine 0.5 osc` builds a sine of half a second per cycle, `200 4000 range` rescales it for the filter, `lpf` stores it. The cutoff sweeps between 200 and 4000 Hz for as long as the note lasts. `sample` is the one word in the table that hands back a number: it reads a motion at the instant the step fires. *Sampling motions* (3.18.4) has it.

Starting a motion

Three words start a motion.

`osc` is the LFO. It takes a shape and the length of one cycle in seconds. The shape is a bare word, like a source name: `sine`, `tri`, `saw`, `square`, `random` or `drift`. The first four are the waveforms. `random` holds one level for a whole cycle then jumps to another, and `drift` glides between the same levels. Any other word stops the step and names the six.

`rise` is a one-shot ramp from `0` to `1` that holds at the top. You give it the time it takes. It counts from the note, so it builds fades and swells.

`current` reads the value the parameter held when the event arrived. It earns its keep as the low bound of a `range`, so a travel starts wherever the parameter sits:

```
saw snd 0.5 rise current 4000 range lpf .
```

The cutoff leaves whatever value it had and reaches 4000 Hz half a second later. `slew` in *Shape words* (3.18.1) does this for you.

Two shapes have no word here: the random walk and the envelope. `drunk` and the `ead` family cover them in *Shape words* (3.18.1). The rest of that page is these three rearranged. `lfo` is an `osc` scaled with `range`, `slide` a `rise` scaled the same way, `slew` that travel starting from `current`.

Scaling into a range

`range` rescales a source's `0` to `1` onto the bounds you name. `expmap` spreads the same motion over a ratio, which is how we hear pitch and cutoff:

```
saw snd sine 2 stepsecs osc 200 4000 expmap lpf .
```

The sweep now takes as long to cross 200 to 400 Hz as to cross 2000 to 4000 Hz, so the wobble sounds even. Both bounds must be positive, and the step stops with `expmap requires positive bounds`.

`range` also takes motions as its bounds. Two of them make a crossfade, and the motion you feed in first sets where that crossfade sits:

```
saw snd 2 rise sine 0.5 osc random 0.25 osc range 200
4000 range lpf .
```

Over the first two seconds the cutoff hands off from the smooth sine to the chain of random holds. The `rise` slides the balance.

Shaping the curve

`smooth` lags a motion, turning its steps into glides. You give it the time it takes to catch up. A slow lag over a random source makes a filter wander:

```
saw snd random 0.25 osc 0.1 smooth 200 4000 range
lpf .
```

The cutoff picks a new target four times a second and eases towards each one. The bare source would jump.

`levels` quantises a motion onto a fixed number of values, so a smooth sweep becomes a staircase:

```
square snd saw 2 osc 8 levels 200 4000 range lpf .
```

The cutoff climbs in eight audible steps and drops back every two seconds.

`curve` bends a motion towards one end of its travel. A positive amount holds it low then rushes, a negative amount rushes then eases, zero leaves it alone:

```
sine snd saw 1 osc 2 curve 0 1 range gain .
```

The swell stays close to silence for most of the second, then rushes up at the end.

`above` turns a motion into a gate: `1` over the threshold you name, `0` under it. Move the threshold and the width of the gate moves with it. That is pulse width modulation without a pulse oscillator:

```
sine snd saw 0.25 osc sine 4 osc 0.1 0.9 range above
0 1 range gain .
```

The gain chops four times a second, and the width of each chop breathes over four seconds.

Combining motions

Motions take the ordinary math words, so you can build one from several. Multiplication controls depth. A motion times a `rise` starts at nothing and grows into itself, so the modulation fades in:

```
sine snd c3 note sine 0.15 osc -1 1 range 1 rise * 25
* detune .
```

The note starts in tune, and a vibrato of a quarter of a semitone swells in over the first second.

Name a motion you write more than once. A `:` definition holds one like anything else:

```
: wobble sine 0.5 osc 200 4000 range ;
```

```
saw snd wobble lpf .
```

Every line that uses `wobble` moves in step with every other, because two copies of one motion are the same program on the same clock.

Delaying and repeating

`later` shifts a motion by the seconds you give it. A shifted `rise` waits before it climbs, because a motion counted from the note holds its first value until its start arrives:

```
sine snd c3 note sine 0.15 osc -1 1 range 1 rise 0.5
later * 25 * detune .
```

The vibrato holds off half a second before it swells.

The same shift on an oscillation puts two copies out of phase. Rotating pan effects are built this way:

```
saw snd sine 1 osc 0 1 range pan sine 1 osc 0.25
later 200 4000 range lpf .
```

The filter follows the same one-second sine as the pan, a quarter of a cycle behind it, so the brightest moment never lands on the widest one.

The shift must be a plain number of seconds. A negative shift over a motion with memory is an error, and `later` refuses a drunk walk outright: `later cannot shift a drunk walk: it has no clock to shift`.

`repeat` sends a one-shot round again as soon as it ends, so a travel loops where it would hold. It works on a `rise` and on a whole envelope:

```
saw snd 200 4000 0.05 0.4 ead repeat lpf .
```

The filter envelope fires, falls, and fires again for as long as the note lasts. `repeat` needs exactly one one-shot to work on. With none you get `repeat needs a rise or an env to loop`, with two `repeat found two one-shots: wrap only the one you mean`.

Seconds and steps

Every duration on this page is in seconds, so `sine 0.5 osc` takes half a second at any tempo. The shape words work the other way. `lfo`, `slide`, `slew`, `jit` and `env` take steps and convert them at the tempo of the moment, so what they build stretches with the music. `sine 2 osc` is two seconds. `200 4000 2 lfo` is two steps, a quarter of a second at 120 BPM. `stepsecs` bridges the two, running the same conversion the shape words run inside themselves:

```
saw snd sine 2 stepsecs osc 200 4000 range lpf .
```

The round trip now takes two steps at any tempo, which is the motion `200 4000 2 lfo` builds. `stepsecs` returns a number, so it fits anywhere a length is expected.

`beatsecs (beats -- secs)` converts beats, for a modulation measured against the pulse. One cycle every two beats is `2 beatsecs`, four cycles per beat is `0.25 beatsecs`. Both carry a short spelling, `ss` and `bs`, so `sine 2 ss osc` is the example above.

Moving lengths

The length you give a source is a motion like anything else, so an oscillation whose length keeps shrinking is a chirp:

```
saw snd sine 1 rise 0.5 0.05 range osc 200 4000 range
lpf .
```

The sweep starts at one cycle every half second and ends up ten times faster by the end of the first second. An oscillation like this has to integrate its own phase, so it counts from the note and not from the transport.

Which clock a motion follows

Two clocks run underneath every motion, and which one it uses decides how it behaves across a pattern. An `osc` with a fixed length follows the transport clock in all six shapes. That clock starts with the transport, not with your note, so two voices carrying the same motion agree and a phrase played twice sounds the same both times.

`rise`, `current`, the envelopes and an `osc` with a moving length count from the note. They start when the event arrives, and they hold their first value until that start comes, so `later` gives a delay and not a jump. `retrig` moves a motion from one clock to the other, counting everything underneath it from the note:

```
saw snd sine 1 osc retrig 200 4000 range lpf .
```

Every note starts its own sweep from the bottom rather than joining the one already running. Chord notes draw their own random values, so a chord shimmers, while one motion written onto two parameters of the same voice agrees exactly.

One catch on the random shapes. A `random` or a `drift` under `retrig` replays the same sequence on every note, because the draw is seeded by the motion and only a chord index moves it. Leave the shape on the transport clock when you want each note to land somewhere new.

Replacing a motion

A parameter holds one motion. Send it a plain number later and the number wins, clearing the motion: `200 4000 2 lfo lpf` sweeps until an `800 lpf` on the same voice pins the cutoff.

Re-evaluating a step does not restart its motions. Cagire compares the new motion against the running one, and identical code leaves that one untouched, still on its own phase. This is what lets you edit a line while an LFO underneath keeps its place. Change the code and the surviving parts carry over: `smooth` keeps its memory, an envelope keeps its stage. `current` is always read afresh, so a re-aimed slew starts from where the parameter stands now.

Limits

A motion is a small program, built before it plays. One holds 24 steps. Ask for more and the step stops with `motion is too long (24 steps at most)`. Two more errors catch expressions that are deep rather than long: `motion holds too many values at once (8 at most)` and `motion has too many moving parts (6 at most)`. All three land on the step that built them, the moment you evaluate it, with the offending word underlined.

A voice carries its own budget on top of that: 15 motions across all its parameters, and 96 steps between them. An orbit takes 16 motions and sets no step budget. Past either ceiling Cagire refuses the motion and counts the refusal.

Voice motions move every sample. Orbit motions move once per block, so the same sweep is drawn in coarser stages on an orbit effect than on a filter inside a voice. Comb frequency and feedback time are the exceptions and move per sample.

Shape words (3.18.1) has the ready-made shapes, *Motion notation* (3.18.3) the written form, *Sampling motions* (3.18.4) how to read one as a number, *Streaming motions* (3.18.5) how to send one down a MIDI or OSC port.

3.18.3 Motion notation

A parameter takes a motion written as text as readily as it takes a number. A string in one of the forms below is read as a motion anywhere a motion is accepted: straight into a parameter, into one of the operators in *Motions* (3.18.2), or out of a port as a controller value.

You rarely write one by hand. The shape words build them for you. This page is the grammar, so you can read a form and know what it does. *The engine* (IV) covers what each one sounds like.

Motion shapes

A mark between the two endpoints picks the form. Hand one to a parameter where you would put a number:

Form	Shape	What it does
`min~max:period`	LFO	oscillate between `min` and `max`, one cycle per period
`start>end:dur`	slide	travel once from `start` to `end` over `dur`, then hold
`>target:dur`	slew	travel from the current value to `target` over `dur`
`min?max:period`	random	jump to a fresh random value in range every period
`min^max:a:d:s:r`	envelope	trace an envelope from `min` to `max` through attack, decay, sustain, release

```
saw snd "200~4000:0.25" lpf .
```

The lowpass cutoff oscillates as a sine between 200 and 4000 Hz, one cycle every quarter second. The envelope form spells out its four stages after the colon, in order: attack, decay, sustain, release.

```
saw snd "200^4000:0.01:0.1:0.5:0.3" lpf .
```

The cutoff opens from 200 toward 4000 Hz on a short attack, falls to its sustain level, and releases back down.

Once read, a written form is an ordinary motion, and the operators work on it like anything else.

```
saw snd "200~4000:0.5" 0.25 later lpf .
```

The same half-second sweep, delayed a quarter of a second from the note that starts it.

Suffixes

A trailing letter picks the variant. On the `~` LFO form, nothing is a sine, `t` a triangle, `w` a saw, `q` a square. On the `>` slide and slew forms the letter names the curve: `e` exponential, `s` smooth, `i` swell, `o` pluck, `p` stair. No letter is linear, which makes six curves in all. On the `?` random form, nothing is a plain jump, `s` smooths the jumps, `d` walks drunkenly. The `^` envelope form takes no letter.

A final `~`, after the curve letter if there is one, loops a travel. The slide or slew restarts at its own length rather than holding at the end.

```
saw snd "200>4000:0.5o~" lpf .
```

The cutoff plucks up from 200 to 4000 Hz every half second, over and over, because the trailing `~` sends the travel round again. It and the `repeat` word do the same thing, so `"200>4000:0.5o" repeat` builds this exactly.

Time and tempo

Every time field is in seconds: the period, the duration, and each envelope stage. The shape words take those lengths in steps and convert them at the tempo of the moment, so what they build follows the music. See *Shape words* (3.18.1).

Strings that are not motions

Cagire reads a string as a motion only when it looks like one. It must carry a colon and one of the four marks `~ ? > ^`. Ordinary text has neither, so it travels on as text and a sample named `what?` is never taken for a curve. A string that looks like a motion but fails to parse stops the step, and the error names the parameter: `lpf: '200~garbage:2' is not a shape I can read`. The word is underlined in the editor, and the last good script keeps playing while you fix it.

What the notation cannot write

The notation covers one shape at a time. A motion you composed has no written form, and Cagire never prints one for it. That covers an oscillation multiplied by a rise, a range with moving bounds, and anything wrapped in `retrig` or `later`. These travel as themselves, straight to a parameter or a port, so the sound loses nothing. There is only no text to look at.

The OSC mirror is where that shows. It reports every emit as text, so a shape the notation can spell is mirrored as itself and anything composed is mirrored as `<motion>`. The engine still plays the real curve. To see a value rather than a shape, sample the motion into a number and print that (see *Sampling motions* (3.18.4)).

3.18.4 Sampling motions

A motion is a curve, so it only has a value at a given instant. `sample` reads one at the moment the step fires and leaves a plain number on the stack. Feed that number to a parameter, print it, compare it, do arithmetic on it. The parameter then holds it steady for the whole event and changes only when the step fires again.

Word	Stack effect	What it does	Example
`sample`	`(m -- n)`	Read a motion as a plain number	`random 1 osc sample`
`beatsecs` / `bs`	`(beats -- secs)`	Convert beats to seconds at the current tempo	`2 beatsecs`
`stepsecs` / `ss`	`(steps -- secs)`	Convert steps to seconds at the current tempo	`2 stepsecs`

```
saw snd random 1 osc sample 200 800 range lpf .
```

`random 1 osc` is a sample and hold with a one-second cycle, `sample` reads it once, `range` scales that reading into the filter's span. The cutoff takes one value and holds it for the whole event.

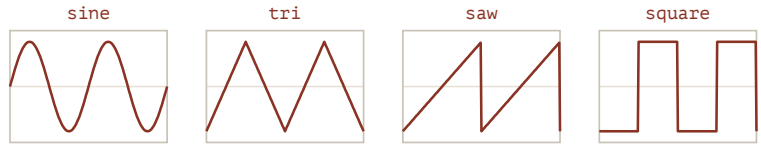
One motion you cannot sample: anything built with `current`. Only the engine knows where a parameter stands, so there is nothing to read when the step fires. The step stops with `sample cannot read a parameter's current value` and never quietly reports zero. Everything else reads fine, the shape words included. What comes back is what the same motion would be playing at that instant, because both are one program on one clock. A filter following a motion and a step that sampled it land on the same value.

Choosing the cycle length

A sampled shape usually wants measuring against the music. `beatsecs` and `stepsecs` convert at the tempo of the moment, as *Motions* (3.18.2) describes. Feed either to `osc` as its cycle length.

```
saw snd sine 2 beatsecs osc sample 200 800 range
lpf .
```

The sine takes two beats per cycle, and each step reads it wherever it has got to. The clock runs whether or not a step fires, so the value walks forward as the music plays. At the start of a cycle the sine reads its midpoint, the saw and the ramps read `0`, and the square reads its low level.



each shape sampled once per fire, read from the clock; the value walks forward as the music plays

Figure 51: The four osc shapes over two cycles: sine, triangle, saw, and square.

Motions (3.18.2) lists the six shapes. `curve` bends any of them, so a saw becomes a ramp that starts slow or fast.

```
saw snd saw 1 beatsecs osc 1 curve sample 200 800
range lpf .
```

The ramp rises once per beat, holding low for most of it before it rushes up.

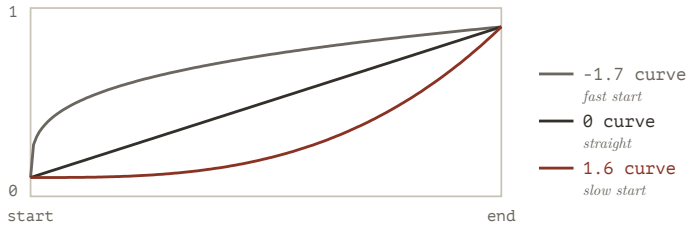


Figure 52: One rising ramp under three curve amounts: slow start above 0, straight at 0, fast start below 0.

Scaling a sampled value into a range

Every shape gives a value from `0` to `1`. `range` rescales that fraction onto the span a parameter wants. `linmap` and `expmap` rescale from any input range. *Stack & math words* (3.3) has all three in full.

Word	Stack effect	What it does	Example
`range`	`(val min max -- scaled)`	Map a `0-1` value onto `min-max`	`0.5 200 800 range`
`linmap`	`(val inlo inhi outlo outhi -- mapped)`	Linear rescale between two ranges	`0.5 0 1 200 2000 linmap`
`expmap`	`(val lo hi -- mapped)`	Exponential rescale of a `0-1` value	`0.5 200 8000 expmap`

```
0.5 200 800 range print
```

Halfway lands at the middle of the range, so this prints `500`. `range` on a sampled number gives a number, `range` on a motion gives a motion.

Drifting noise

`drift` is the smooth one of the two random shapes. It draws a fresh value every cycle as `random` does, then eases from one to the

next, so a slow `drift` wanders where a slow `random` jumps. Use it when a filter or a pan should move and not lurch. Cagire draws its values from the current `seed`, so reseeding to the same number replays the same wander and a different number gives an unrelated one. See *Random values* (3.13.1).

```
saw snd drift 10 beatsecs osc sample 200 800 range
lpf .
```

One cycle every ten beats, so Cagire picks a new target every ten beats and eases the cutoff toward it. Shorten the cycle for nervier movement, lengthen it for a drift you feel and do not hear.

Reading centred on zero

A shape gives a value from `0` to `1`. For one centred on zero, a detune or a pan offset, rescale it with `range`. The bounds work on a plain number as they do on a motion, so `-1 1 range` gives the bipolar reading of any shape.

```
sine snd
sine 0.25 beatsecs osc sample -1 1 range
0.02 * detune
.
```

Wobbles the detune around zero with a sine, four cycles per beat, scaled small. To leave that shape as a curve and let Cagire play it across the event, see *Motions* (3.18.2).

3.18.5 Streaming motions

A motion has no single value, so Cagire sends the whole curve. Write a motion where a MIDI or OSC value would go and the port carries a run of messages tracing that curve across the step. Cagire draws it with the same code the audio engine runs, on the same clock, so a filter sweep and a controller written the same way move together. Everything in *Motions* (3.18.2) works here: the shape words, the written notation, and a motion you composed all reach a port as themselves.

```
1 chan 1 ccnum 0 127 1 lfo ccout m.
```

CC 1 sweeps the full controller range and comes back over one step.

Where a motion goes

Any controller value takes one. On MIDI that is `ccout`, `bend` and `pressure`. The other parameters of `m.`, the channel and the CC number, are addresses and want plain numbers:

```
1 chan -1 1 1 lfo bend m.
```

The pitch bend swings the whole wheel, drawn across the fourteen bits a bend carries and not the seven of a CC.

```
1 chan 0 127 2 lfo pressure m.
```

Aftertouch breathes once across two steps. A cheap way to make a receiving synth sound played and not triggered.

On OSC every argument of `o.` takes a motion too, and each moving argument gets its own stream inside the same messages:

```
"/synth" address 100 200 1 lfo freq o.
```

The path receives a run of messages. `freq` climbs from 100 to 200 and comes back over the step while every other argument repeats unchanged. Motions you composed travel as well, written form or not:

```
1 chan 1 ccnum sine 0.5 osc sine 3 osc * 0 127 range
ccout m.
```

A half-second sine, with a slow three-second one controlling its depth, drawn onto CC 1 as a single stream.

What Cagire sends

A plain number is still one message. Only a motion becomes a stream. Cagire draws the curve at up to 200 points a second, and never more than 512 points in one event, so a long step or a slow tempo cannot flood a port.

The rate comes from what the cable carries. MIDI takes about a thousand three-byte messages a second, and a 7-bit controller has

128 values to reach, so drawing finer would mostly produce repeats. Cagire drops them anyway. A CC stream is rounded to the seven bits it will occupy and never sends the same value twice in a row, so a motion that barely moves sends a handful of messages and not hundreds. OSC arguments carry floats, with no such resolution to round to, and nothing is dropped there.

Whatever the cap and the rounding do in between, the last point of a stream is always the value the motion ends on. A controller is never left one step short of where the curve was going.

Envelopes and the gate

An envelope on a controller runs its attack, decay and sustain across the note and releases when the gate ends, not when the step ends. The voice envelope follows the same rule, which is what makes a controller envelope usable as an expression curve:

```
1 chan 1 ccnum 0.25 gate 0 127 0.05 0.05 1 0.2 env
ccout m.
```

CC 1 rises, then holds at its sustain level for the quarter step the gate lasts. It falls back to zero over the release, drawn past the end of the gate and not cut off with it.

Slews and the last value sent

A slew travels from wherever the parameter already stands. Cagire has no way of knowing where a knob on the receiving machine is set, so it remembers the last value it sent to each destination and slews from there:

```
1 chan 1 ccnum 100 0.5 slew ccout m.
```

On a destination already streaming, the CC travels from its last value to 100 over half a step. On one with no history the first message is 100 and it holds, so you land on the value you asked for and never sweep from an invented start. A destination is one CC number on one channel, or a channel's bend, or a channel's pressure,

or one argument of one OSC address, each on its own device. None share memory, so a slew on CC 1 never inherits from CC 74.

Random motion

`drunk` keeps no memory across events on a port. Each event draws a fresh walk from the middle. A wandering controller still wanders, but it will not drift somewhere and settle there over a phrase.

What matches the audio engine

Everything that depends only on the clock matches exactly on both sides. An `osc` with a fixed length in any of the six shapes, arithmetic on it, and any plain number all read the same on a controller, in a filter and through `sample`, at one instant. One is drawn `200` times a second and the other at the audio rate, which makes no difference to the value.

Motions that carry memory track each other without matching exactly. `smooth` lags its input, an `osc` with a moving length integrates its own phase, and a drunk walk draws its steps one at a time. Two copies of these at different rates stay close in shape and land on different numbers. Motions counted from the note agree as long as the note and the gate are the same, which is what an event hands both sides at once. That covers `rise`, `current` and the envelopes.

Motions (3.18.2) has the vocabulary. *MIDI* (3.19.1) and *OSC* (3.19.2) have the ports these streams travel out of.

3.18.6 Control buses

A control bus is a number that lives in the audio engine, outside any voice. The engine holds 128 of them, called lanes and numbered `0` to `127`, and each lane carries a value from `0` to `1`. A pattern, a MIDI controller or an OSC message writes a lane. A motion reads it, so a parameter follows the lane while the sound plays and no new event is needed. This is how a fader reaches a filter that is already sounding.

Word	Stack effect	What it does
`bus`	`(lane -- m)`	A motion that reads lane `lane`, `0` to `1`
`bus!`	`(value lane --)`	Write a plain number to lane `lane` on this step's beat
`cc>bus`	`(cc chan lane --)`	Route an incoming MIDI control change to a lane
`osc>bus`	`(addr idx lane --)`	Route one argument of an incoming OSC message to a lane
`busmap-clear`	`(--)`	Drop every route

```
saw snd c3 note 3 bus 200 4000 range lpf .
```

`3 bus` is a motion that reads lane 3. `200 4000 range` scales it like any other source in *Motions* (3.18.2), and `lpf` stores it. The cutoff sits at 200 Hz while the lane is `0`, at 4000 Hz while it is `1`, and moves with the lane in between for as long as the note lasts.

Reading a lane

`bus` takes a lane number and leaves a motion. Every source in *Motions* (3.18.2) gives a value from `0` to `1`, and a lane is no exception, so `range` or `expmap` ends the line. The motion takes the rest of the vocabulary too. `smooth` lags it, `levels` quantises it, and arithmetic combines it with another motion:

```
saw snd c3 note 3 bus sine 0.5 osc * 200 4000 range
lpf .
```

The sine wobbles the cutoff, and lane 3 sets how deep the wobble goes.

The engine reads the lanes once per DSP block, thirty-two samples by default, so a lane moves in small steps rather than continuously. A fast fader move on a sensitive parameter can be heard as those steps. `smooth` hides them:

```
saw snd c3 note 3 bus 0.02 smooth 200 4000 range
lpf .
```

`sample` reads a lane as a plain number at the moment the step fires (*Sampling motions* (3.18.4)). `3 bus sample` leaves the current value of lane 3 on the stack, where it can be printed, compared or fed to a word that takes no motion:

```
3 bus sample print
```

Writing a lane

`bus!` writes a plain number to a lane. The write is scheduled on the step's beat exactly as a sound is, so a pattern can move a lane in time and the change lands where the step does:

```
< 0 0.5 1 0.5 > 3 bus!
```

Every step writes a new level to lane 3, and any voice reading `3 bus` follows. `bus!` emits the write itself, so no `.` follows it, and it leaves the sound register untouched. `bus!` takes numbers only. Hand it a motion and the step stops with `bus! takes a plain number`. A lane keeps its last value until something writes it again. Hush and panic clear the voices and leave the lanes alone.

Routing a controller to a lane

`cc>bus` routes a MIDI control change to a lane. It takes the CC number, the channel from `1` to `16` and the lane, and it reads the input device from `dev` like `ccval` does. Once routed, every control change that arrives is stored in the lane as its value divided by 127, without waiting for a step:

```
1 1 3 cc>bus
```

CC 1 on channel 1 of the device set by `dev` now drives lane 3.

`osc>bus` does the same for OSC. It takes the address as a string, the zero-based index of the argument to read, and the lane. Integer, float and boolean arguments are stored; a string argument is ignored:

```
"/touchosc/fader1" 0 3 osc>bus
```

A route is installed the moment the word runs, and running it again with the same values changes nothing, so the routing words

belong in a bank or global prelude. Routing a controller that is already routed replaces the old lane. `busmap-clear` drops every route at once:

```
busmap-clear
```

Buses on the wire

A motion that reads a lane has no closed form, because the lane can change at any moment. When such a motion is sent down a MIDI or OSC port with `m.` or `o.`, Cagire reads the lane once when the event fires and holds that value for the span of the event (*Streaming motions* (3.18.5)). The audio engine reads the lane live, so a filter and a controller written with the same `bus` motion agree at the start of the event and can diverge after it.

Buses in a patch

Inside the braces `bus` is an Arf word, `bus ( lane -- sig )`, and the lane must be a constant. It reads the lane as a signal, latched once per block like the sequencing side:

```
{ 110 saw  3 bus 0 1 200 4000 linlin lpf  0.2 * out }
synth .
```

An effect patch may read a lane. Unlike `gate`, `notefreq` and `vel`, a lane belongs to the engine and not to a note, so it is allowed alongside `in` as `bps` is (*Limits*). This is how a fader reaches an orbit effect built with `dsp`.

Motions (3.18.2) has the vocabulary a `bus` motion joins, *Sampling motions* (3.18.4) the fire-time read, *Streaming motions* (3.18.5) what happens on a port. *MIDI* (3.19.1) and *OSC* (3.19.2) have the inputs a route reads from, and *Core words* (4.1) has the patch side.

3.19 MIDI, OSC & recording

3.19.1 MIDI

Cagire speaks MIDI from the same register you use for sound. You fill the register with parameter words, then fire it with `m.` instead of `.`, and the engine sends a MIDI message to the chosen device. Transport words drive an external clock, and `ccval` reads control changes back from an input. The register and its parameters are described in *The sound register* (2.5).

Emitting: `m.`

`m.` is `( -- )`. It takes no stack values. It reads the parameters currently in the register and sends one MIDI message. Which kind of message it sends depends on which parameters you have set. With nothing special set it sends a note, reusing the register's `note`, `velocity`, and `gate`.

```
60 note 1 chan m.
```

Cagire sends a note-on for MIDI note 60 on channel 1. The velocity defaults to `0.8` (sent as 0 to 127) and the duration follows `gate`, scaled to the tempo.

Choosing the message

`m.` picks the message type by the parameters present, in a fixed order: a CC if both `ccnum` and `ccout` are set, otherwise a pitch bend if `bend` is set, otherwise channel pressure if `pressure` is set, otherwise a program change if `program` is set, otherwise a note. Set the parameters for the one message you want.

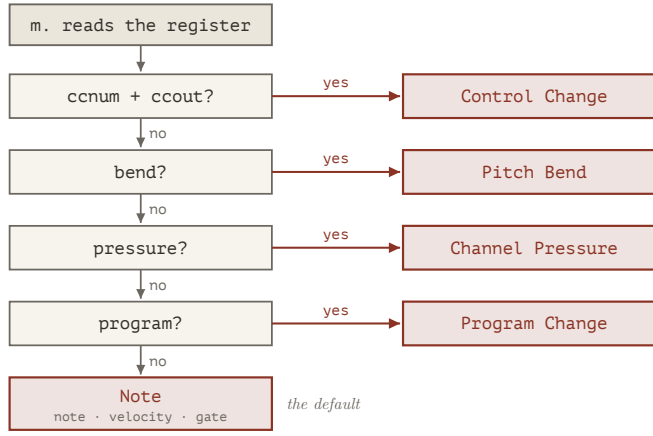


Figure 53: m. tests the flags in a fixed order and sends the first that matches, from control change down to a plain note.

```
1 ccnum 64 ccout 1 chan m.
```

Sends control change number 1 with value 64 on channel 1.

```
0.5 bend 1 chan m.
```

Sends a pitch bend halfway up. `bend` runs from `-1.0` to `1.0`, with `0` at center; Cagire converts it to the 14-bit MIDI range.

```
64 pressure 1 chan m.
```

Sends channel pressure (aftertouch) of 64 on channel 1.

```
0 program 1 chan m.
```

Sends program change 0 on channel 1.

Parameters

These parameter words feed `m.`. Each takes its value from the stack, the same postfix way as the sound parameters. `ccout`, `bend`, and `pressure` take a motion as readily as a number, in which case Cagire streams the curve across the step instead of sending one message (see *Streaming motions* (3.18.5)).

Word	Stack effect	What it does
`chan`	`(v.. --)`	set MIDI channel, 1 to 16
`ccnum`	`(v.. --)`	set the CC number, 0 to 127
`ccout`	`(v.. --)`	set the CC value to send, 0 to 127
`bend`	`(v.. --)`	set pitch bend, `-1.0` to `1.0` (`0` is center)
`pressure`	`(v.. --)`	set channel pressure (aftertouch), 0 to 127
`program`	`(v.. --)`	set program change number, 0 to 127
`dev`	`(v.. --)`	set the output device slot, 0 to 3

Device routing

`dev` chooses which of Cagire's four output slots receives the message. The same word routes OSC when you emit with `o.`, so a slot is a destination, not a protocol.

```
1 dev 60 note m.
```

Sends the note to device slot `1` instead of the default slot `0`.

Transport

These words send MIDI real-time messages to drive an external sequencer or drum machine. They take nothing from the stack and emit on their own.

Word	Stack effect	What it does
`mclock`	`(--)`	send one clock pulse (24 per quarter note)
`mstart`	`(--)`	send a start message
`mstop`	`(--)`	send a stop message
`mcont`	`(--)`	send a continue message

```
mstart
```

Tells the connected device to start. Pair it with `mstop` to halt and `mcont` to resume. Emit `mclock` repeatedly to supply the tempo.

Reading input: `ccval`

`ccval` is `(cc chan -- val)`. It reads the last value received for a control change on a MIDI input and leaves it on the stack, where you can `print` it or feed it into a parameter. It returns `0` when nothing has arrived, and it reads from the device set by `dev`.

```
1 1 ccval print
```

Reads CC number `1` on channel `1` and prints its value (`0` until that controller sends something).

`ccval` reads a controller once, when the step fires. To follow a controller while a sound plays, route it to a control bus with `cc>bus` and read the bus as a motion. A motion that reads a bus and travels back out through `m.` is read once at fire time and held for the event (*Control buses* (3.18.6)).

3.19.2 OSC

Cagire sends and receives OSC from the same register it uses for sound. You name a path with `address`, fill the register with parameters, then fire it with `o.`. Every parameter in the register travels as a named OSC argument. To read a value back, `oscval` returns one argument of the last message that arrived at an address. The register is described in *The sound register* (2.5).

Setting the path: `address`

`address` is `(s --)`. It takes a string off the stack and stores it as the path for the next emit. The path must start with `/`.

```
"/synth" address 440 freq o.
```

This sets the path `/synth`, attaches one argument named `freq` with value 440, and sends the message.

Emitting: `o.`

`o.` is `(--)`. It takes nothing from the stack. It reads the address you set and every other parameter in the register, and sends them as flat key/value pairs. Each parameter becomes its name followed

by its typed value, so the parameter words you already know (`freq`, `pan`, `note`, and the rest) double as OSC argument names.

```
"/synth" address 200 freq 0.5 pan o.
```

Sends a message to `/synth` carrying `freq` as the integer `200` and `pan` as the float `0.5`. Cagire tags each value by its type (integers, floats, and strings travel as themselves) and routes it through the device set by `dev`. An emit with no `address` set is an error. An argument written as a motion is streamed as a run of messages tracing that curve rather than sent once (see *Streaming motions* (3.18.5)).

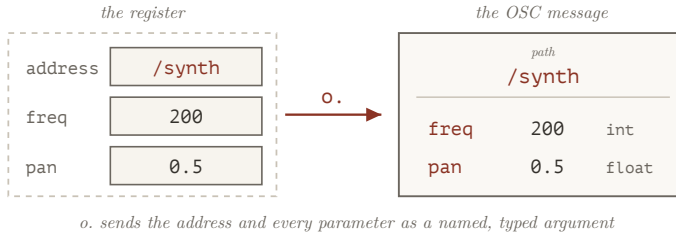


Figure 54: o. turns the register into one OSC message: the address becomes the path, and every other parameter becomes a named argument carrying its typed value.

Reading input: `oscval`

`oscval` is `(addr idx -- v)`. It reads one argument of the most recent OSC message received at an address: take the address as a string and the zero-based argument index, and it leaves that argument's value on the stack. A path that has received nothing returns 0, and an index past the end of the message leaves silence rather than a value.

```
"/touchosc/fader1" 0 oscval print
```

Reads the first argument of the last message at `/touchosc/fader1` and prints it (0 until that path receives something). Feed the value into a parameter to map an external controller onto a sound.

`oscval` reads the argument once, when the step fires. To follow it while a sound plays, route it to a control bus with `osc>bus` and read the bus as a motion. A motion that reads a bus and travels back out through `o.` is read once at fire time and held for the event (*Control buses* (3.18.6)).

3.19.3 Recording

These words drive Cagire's looper. A start verb opens a recording into a named buffer. A stop verb closes it and registers the result as a sample you can play back like any other source. The `o`-prefixed verbs record a single orbit (one output bus) instead of the full master output, and the `dub` verbs layer onto a buffer that already exists.

Starting a recording

`rec` is `(name --)`. It takes a name string and starts capturing the master output into that buffer.

```
"loop1" rec
```

Cagire emits the looper command `/doux/rec/loop1` and begins recording everything you play into a buffer called `loop1`.

`dub` (alias `overdub`) is `(name --)`. It starts an overdub onto a buffer that already holds a recording, so new sound layers on top of the old.

```
"loop1" dub
```

Emits `/doux/rec/loop1/overdub/1`, overdubbing onto `loop1`.

Recording one orbit

An orbit is a single one of Cagire's parallel output buses. `orec` and `odub` record just that bus rather than the whole mix, so you can capture one voice in isolation. Both are `(name orbit --)`: the name first, then the orbit number.

```
"drums" 0 orec
```

Emits `/doux/rec/drums/orbit/0`, recording only orbit 0 into `drums`.

```
"drums" 0 odub
```

Emits `/doux/rec/drums/overdub/1/orbit/0`, overdubbing onto orbit 0 of `drums`.

Stopping

Three stop verbs close whatever recording is active. They are all `(--)` and they emit the same command, so reach for the one that reads best next to the start verb you used.

Word	Stack effect	What it does
`endrec`	`(--)`	stop the active recording and register the sample
`enddub`	`(--)`	stop the active overdub
`endorec`	`(--)`	stop the active orbit recording

```
endrec
```

Emits `/doux/rec/endrec/1`, stopping the recording and turning the buffer into a playable sample. Once registered, the name plays like any source: write it bare and sound it with `snd`, as in *The sound register* (2.5).

Chapter 4

Your own language

4.1 Global prelude

The global prelude is a script shared by your whole project. It runs once before anything plays, so it is where you put the definitions, variables, and settings that every bank and every step should see. Press `P` to open it. Closing the editor saves and evaluates it. Cagire runs the global prelude first, before any bank prelude (see *Bank prelude* (4.2)), so a name defined here is available everywhere unless a bank redefines it later. Use it to store project-wide state and vocabulary:

```
c2 !root
42 seed
```

This sets a session-wide `root` that any step can read with `@root`, and `42 seed` fixes the random seed so a generative pattern repeats the same way each run. Variables are covered in *Scope and variables* (2.9), randomness in *Randomness* (3.13).

The prelude defines and configures. You cannot play anything musical from it. No step is playing while it runs, so the context words (`step`, `iter`, `phase`, and the rest from *The live model* (2.8)) have no meaning here. A bare `.` is evaluated and thrown away: no event a prelude emits reaches the engine, so you get no sound and no error either. Keep emitting code in your steps and the prelude for setup. Definitions are the exception, and that is the point of the place: a `defsynth` here installs its instrument for real, and reports its error here if it fails to compile (see *Named patches* (2.2)). Preludes are re-evaluated when playback starts, when a project loads, and whenever you press `d`. If you edit the prelude while playing, press `d` to push the changes into the running session.

4.1.1 Keys

Key	What it does
`P`	Open the global prelude editor
`d`	Re-evaluate all preludes

4.2 Bank prelude

Each bank can carry its own prelude. It works just like the global prelude (see *Global prelude* (4.1)) but it belongs to one bank and travels with it: save or share a bank and its definitions come along, so the bank stays self-contained. Press `p` to open the current bank's prelude. The most common use is to name your instruments. Without a prelude, every step that plays a bass has to spell out the whole sound:

```
saw snd
c2 note
0.8 gain
2000 lpf
.
```

Define it once in the bank prelude instead.

```
: bass saw snd 0.8 gain 2000 lpf . ;
```

Now any step in the bank plays a bass at a chosen pitch by writing `c2 note bass`. Changing the sound in the prelude changes every step at once. A prelude usually grows into a small vocabulary.

```
: bass saw snd 0.8 gain 2000 lpf . ;
: stab sine snd 0.4 verb . ;
: octup 12 + ;
```

Steps become short and expressive: the prelude holds the sound design, the steps hold the music. Defining a word never makes sound, even when the body ends in `.`; the `.` only fires when a step calls the word. Cagire evaluates the global prelude first, then each non-empty bank prelude in bank order, so a bank prelude can build on the global definitions or override one for its own bank. Like every

prelude, it runs at playback start, on load, and when you press `d`, and it should define rather than emit (see *Global prelude* (4.1)).

4.2.1 Keys

Key	What it does
`p`	Open the current bank's prelude editor
`d`	Re-evaluate all preludes

4.3 User dictionary

Words you define in a step or a prelude live inside that project. For vocabulary you want in every project, Cagire keeps a user dictionary: words saved with the application itself and loaded into every session. From the Dictionary view, press `u` to open it. You write only the body of each word, and Cagire wraps it under the entry's name, exactly as if you had typed `: name body ;`. So an entry named `octup` with the body `12 +` becomes the word `octup`, which works like any other. Cagire evaluates the user dictionary before the project and bank preludes, so your preludes can build on these words. Any project can redefine one locally without touching the saved version. Entries are stored as you edit them. There is nothing to save by hand. The user dictionary has its own editor inside the Dictionary view, covered under the interface *Dictionary* (2.1) topic.

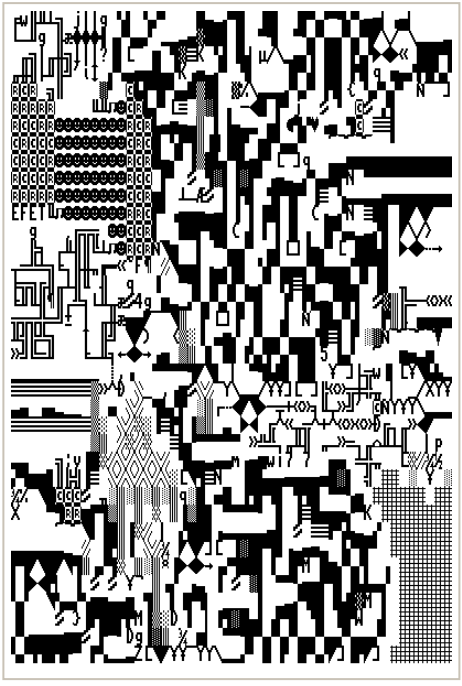
Each word is a plain text `.forth` file in a `words` folder next to Cagire's configuration file. The filename supplies the word's name and the file contains only its body. Cagire reads the folder at startup and whenever you open the User Dictionary, so changes made with another editor appear when you return to the view. Characters that are not portable in filenames are percent-encoded. For example, the word `ready?` is stored in `ready%3F.forth`; creating and renaming words inside Cagire handles this automatically.

4.3.1 Keys

Key	What it does
`u`	Open the user dictionary (from the Dictionary view)
`Enter` / →	Edit the selected word's body
`a`	Add a word
`r`	Rename the selected word
`d`	Delete the selected word
`Ctrl+E`	Save and re-evaluate the whole dictionary

Part IV

The engine



Chapter 1

Introduction

Cagire embeds its own audio engine named Doux. Doux is a synthesizer, a sampler and a recorder in one. It has been built and fine-tuned for live coding. Doux is built on the legacy of Dirt, SuperDirt and SuperDough, and belongs in the same family of engines. These engines are very popular within the live coding scene and have been used in many different environments throughout the years. Doux extends the classic model it inherits from and takes it as far as possible. The reason for that is simple: Cagire is built to be standalone and should be able to do as much as possible without the need for other resources or software. When you use Doux, there are no cables to patch and no plugins to load. You describe a sound in words and fire that sound you've just described. The engine builds a voice to match. Doux is simple in appearance but covers a very broad sonic territory, and this chapter walks that territory stage by stage.

In this chapter, we assume you now know how to use the Cagire language, and we dive deeper into sound synthesis, sampling and creative audio patching. Let's start by playing a simple voice, like the ones you have already met throughout the previous chapters:

```
saw snd
c4 note
0.8 gain
0.3 verb
.
```

That one voice is the whole engine concept in miniature. There is a source (`saw`) with parameters (pitch, gain). There is also an effect: a little reverb. `.` turns these parameters into a synthesis voice, a self-contained instrument the engine plays until its envelope ends (see *What is a doux voice?* (3.1)). The next voice could be totally

different, use a different set of parameters or add some new ones. Each voice is totally independent and, unless you deliberately hold it open, lives for a finite period of time after which it is killed. Crafting Doux voices is your job as a live coder. This chapter will teach you everything there is to know about that arcane art.

1.1 Inside a voice

Every voice runs a single fixed path from raw tone to finished sound, source first, then the chain of filters and effects, and you cannot reorder it live. Most of those stages stay dormant until you set a parameter that wakes them, so a bare source is only the beginning of what a voice can become.

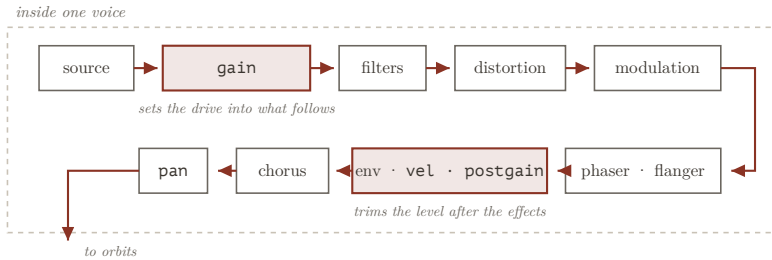


Figure 55: One voice's fixed processing order, from source through filters and effects out to its orbit; gain drives the chain, postgain trims the result.

What are insert effects? (7.1) walks that chain stage by stage and shows why its order, with `gain` driving everything and `postgain` only trimming the result, explains most surprises.

1.2 Voices and orbits

Each voice is independent, with its own oscillator, envelope, and effects, and the engine plays many at once up to a fixed budget (see *What is a doux voice?* (3.1)). The per-voice filters and distortion are insert effects, private to the voice; effects better shared, like reverb and delay, live on orbits, shared send buses that many voices feed, so the signal flows voices, then orbits, then output. How orbits build a sense of space is in *What are orbits?* (8.1).

1.3 What this part covers

The chapter is loosely organized following the signal flow. It also goes from simple topics to more advanced ones. For that reason, it starts with sources: oscillators simple and complex, drums, noise, samples, live input, and soundfonts. Then it moves on to the topic of *Managing voices* (3) which also covers cut groups, held voices, gain, and panning. *Source modulation* (4) will teach you to bend and reshape, and *Envelopes* (6) will teach you what the voice lifecycle is in detail. *Insert effects* (7) are the per-voice filters, distortion, and space effects. *Orbit effects* (8) are the shared reverbs, delays, and feedback on the send buses. *Output* (9) will tell you what happens when a voice is ready to play and goes through the speakers. The *Audio patching* (V) chapter right after this one covers the building blocks to build synthesizers and effects from scratch, on your own, inside `{ ... }` blocks. Let's start the dive by detailing every potential audio source you can use to make music.

Chapter 2

Sources

2.1 Periodic oscillators

The periodic oscillators are the most basic sources in the engine, the raw material you reach for first. There are seven of them, each a waveform that repeats at a steady pitch. You name one as a source and play it like any other sound, and from there you can modulate and reshape it as far as you like, building a full tone out of a plain starting point.

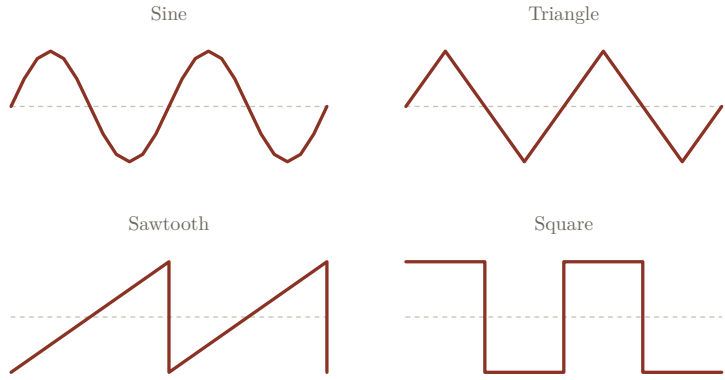


Figure 56: The four basic waveforms: sine, triangle, sawtooth, and square.

```
saw snd .
```

That is a plain sawtooth at the default pitch. You can give it a note, or any other parameter that shapes the sound:

```
saw snd c4 note .
```

The seven shapes fall into a few groups. `sine`, `tri`, and `saw` are the band-limited staples, and they differ in harmonic content: `sine`

carries a single partial, `tri` only the odd harmonics with a steep rolloff, `saw` the full harmonic series. `zaw` and `pulze` are the same shapes without anti-aliasing, cheaper to compute and free to alias at high pitches. `pulse` is a pulse whose width you set, and `vosc` morphs through sine, triangle, saw, and square on one parameter. The table sums up the whole set.

Source	Also	Character
`sine`		single partial, no harmonics
`tri`	`triangle`	triangle, odd harmonics, steep rolloff
`saw`	`sawtooth`	band-limited sawtooth, full harmonic series
`zaw`	`zawtooth`	sawtooth, no anti-aliasing
`pulse`	`square`	pulse, width set by `pw`
`pulze`	`zquare`	pulse, no anti-aliasing
`vosc`	`oscillator`	morphs sine to square via `wave`

The second column is the alias each shape also answers to, so `square snd` and `pulse snd` name the same oscillator. A square is what `pulse` gives you at its default width anyway, which is where that name comes from.

2.1.1 Pulse width

`pulse` and `pulze` take `pw`, the fraction of each cycle the wave spends high, from `0` to `1`. At `0.5` the two halves are equal and you get a square. Other values make the pulse asymmetric, which changes its harmonic content.

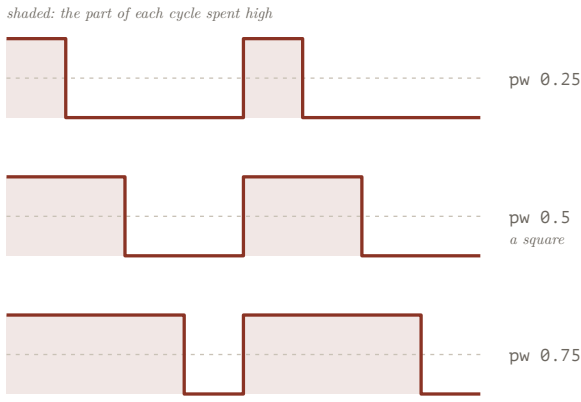


Figure 57: At 0.25 the wave is high a quarter of the cycle, a square at 0.5, high three-quarters at 0.75.

```
pulse snd
0.25 pw
.
```

`pw` takes modulation, so a slow sweep through it keeps the harmonic content moving, the effect known as pulse-width modulation:

```
pulse snd
0.1 0.9 1 lfo pw
.
```

2.1.2 The morphing oscillator

`vosc` reads `wave` from `0` to `1` and crossfades through the shapes as it goes: `0` is sine, `0.33` triangle, `0.67` saw, `1` square. The values between blend smoothly.

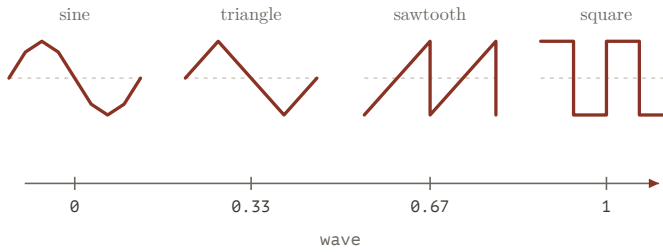


Figure 58: `vosc` *lays the four shapes along the* `wave` *axis: sine at 0, triangle at 0.33, sawtooth at 0.67, square at 1.*

```
vosc snd c4 note 0.67 wave .
```

Because `wave` takes modulation, you can morph the timbre across a held note:

```
vosc snd c3 note 0 1 4 lfo wave .
```

2.1.3 Shaping the phase

The basic shapes also answer to three phase-shaping parameters that bend the waveform without changing its pitch: `size` (`0` to `256`) quantizes the phase into chunky, harmonic-rich steps, `warp` (`-1` to `1`) tilts the cycle so one half rushes and the other drags, and `mirror` (`0` to `1`, the only one of the three that accepts modulation) folds the wave back at a chosen point. A taste:

```
saw snd 8 size .
```

Parameter	Range	Sets
`wave`	0 to 1	`vosc` waveshape morph
`size`	0 to 256	phase quantization steps
`warp`	−1 to 1	phase asymmetry
`mirror`	0 to 1	phase reflection point

These three parameters are the tip of the iceberg. Refer to *Phase distortion* (4.7) to explore this topic in more depth.

2.1.4 A plucked cousin

Not every tonal source is a periodic oscillator. `pluck`, also `ks` or `string`, is a Karplus-Strong string. It is a burst of noise that rings through a tuned, damped loop, retuned every sample so vibrato bends it continuously. You play it like an oscillator and shape it with `bright` (loop damping), `ring` (sustain), and `excite` (the initial noise burst):

```
pluck snd c3 note 0.8 bright 0.9 ring .
```

For thickening with detuned copies see *Spread, detune & glide* (4.1), for an octave layer see *Sub-oscillators* (4.4), for the sweeping sync sound see *Hard-sync and soft-sync* (4.5), and for vibrato see *Pitch modulation* (4.2).

2.2 Wavetable oscillators

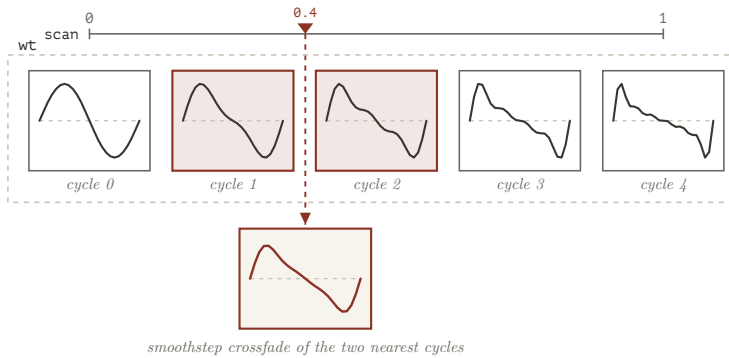


Figure 59: A wavetable is a row of single-cycle waveforms; scan parks between two adjacent cycles and crossfades between them.

Wavetable oscillators are a staple of digital synthesis and sit at the heart of many well-known hardware and software synths. They are cheap for a computer to run and can voice almost any timbre. Collect or build a library of single-cycle wavetables and the same oscillator plays them all. Musicians and engineers have refined the technique over the years and added ways to make the tone evolve.

Any audio sample can become a pitched wavetable oscillator. `snd` opens a sound command that plays a sample as a recording. `wt` opens one that reads the same sample as a row of single-cycle waveforms. It loops one cycle at a time at whatever pitch you give it. Cagire ships with no samples of its own. The examples on this page will stay silent until you load a wavetable yourself under the same name. Point Cagire at a folder of samples first (see *Loading samples* (5.1)), then use the names from your own library in place of `pad` below:

```
pad wt c3 note .
```

A wavetable file holds those single-cycle waveforms end to end. The oscillator reads through one cycle at audio rate, and that sets the pitch. `scan` chooses where in the row you are, from `0` at the first cycle to `1` at the last. Between cycles it crossfades, so sweeping `scan` morphs the timbre smoothly.

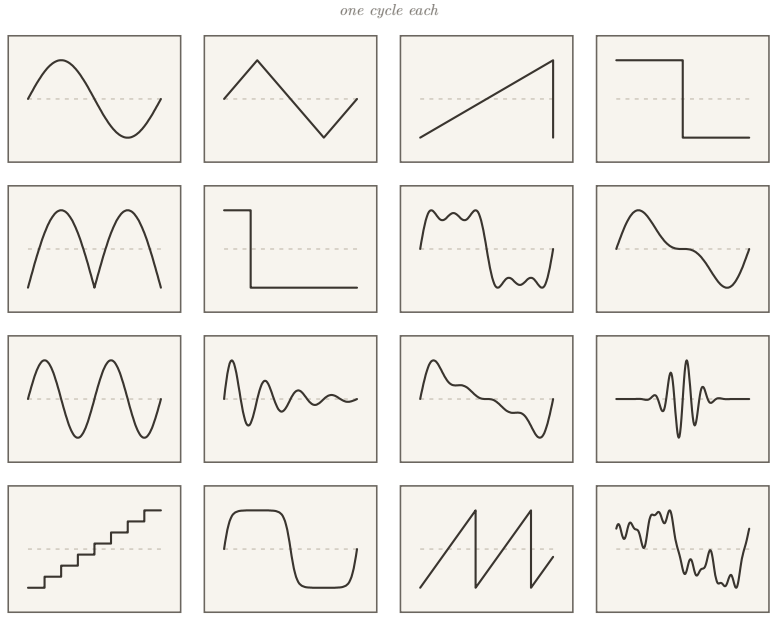


Figure 60: Every frame of a wavetable is a single cycle, and a cycle can take almost any shape.

```
pad wt 0.5 scan c3 note .
```

The same sample still plays as an ordinary recording under `snd`, and `scan` never switches it into a wavetable. It keeps one meaning across both, a position through the buffer. Here it picks a cycle out of the row. Under `snd` it places the playhead in the recording (see *Scanning* (5.5)). Choosing `wt` is the whole difference:

```
pad snd .
```

2.2.1 Cycle length

`wtlen` is how many samples make one cycle, and Cagire works it out from the file so you rarely have to say. Most packs come from tools that write the cycle length into the file itself. For the ones that do not, Cagire reads a length that divides evenly into 2048-sample cycles as a 2048 pack. That covers nearly everything you will download. It holds even when your audio device runs at a different rate than the file. The file keeps the rate it was written at, and a cycle length counts the same samples either way.

Set `wtlen` yourself when a table breaks both rules. The common sizes are powers of two: `256`, `512`, `1024`, and `2048`, the last being the Serum standard. The number of cycles in the table is its total length divided by `wtlen`:

```
pad wt 512 wtlen c3 note .
```

Getting `wtlen` wrong does not put the note out of tune, because pitch comes from the oscillator and not from the file. You hear the wrong waveform instead. The reader takes a slice that is not a whole cycle, so every period has a jump in it, and `scan` lands between frames rather than on them.

A file that holds a single waveform and nothing else has no cycle length to find, so Cagire reads the whole sample as one cycle. That is exactly what you want. A one-cycle sample becomes a clean custom oscillator, and `scan` has nothing to morph between.

2.2.2 Scanning in motion

`scan` takes modulation, so the wavetable comes alive when you move through it rather than parking on one cycle. A slow `lfo` sweeps the timbre back and forth:

```
pad wt 0 1 2 lfo scan .
```

Pitch works the same as on any oscillator, so `note` and `freq` place the wavetable where you want it, and the filters, envelope, and modulation all apply on top:

```
pad wt c3 note tri 4 stepsecs osc 0 1 range scan 800
lpf .
```

The words that shape sample playback have nothing to act on here. `begin`, `end`, `speed`, `stretch`, and the granular words all describe a position in a recording. A wavetable has no such position. It reads one cycle over and over at the pitch you asked for. Reach for `snd` when you want those.

WORD	RANGE	SETS
`wt`	a sample name	opens a sound command that reads the sample as a wavetable
`scan`	0 to 1	position in the row of cycles, takes modulation
`wtlen`	samples per cycle	overrides the cycle length Cagire read from the file

2.2.3 Making a wavetable

A wavetable file is an ordinary audio file built to a few rules. It holds a series of single-cycle waveforms of identical length. Each cycle starts and ends at a zero crossing, so it loops without a click. The cycle length is a power of two. You can find ready-made wavetable packs online, or build your own in a tool like Serum, WaveEdit, or Audacity with zero-crossing snap turned on. Exports from Serum and its relatives carry their cycle length inside the file, so Cagire picks them up with no `wtlen` from you.

2.3 Complex oscillators

Each periodic oscillator gives you one waveform. Each complex oscillator gives you a whole synthesizer. Cagire has ten of them, ported from the Mutable Instruments Plaits module. Every one is a different synthesis method: a resonator you strike, a chord machine, a cloud of detuned saws, a formant filter bank. You play them like any other source. Name one and fire it.

```
plmodal snd c3 note .
```

Every name starts with `pl`. The prefix keeps these sources clear of parameters that already own the obvious words. The engine has a `modal` insert effect, and `chord`, `grain` and `fm2` are parameters. The sources are therefore `plmodal`, `plchord`, `plgrain` and `plfm`.

2.3.1 Three knobs, ten meanings

All ten read the same three parameters and nothing else: `harmonics`, `timbre` and `morph`. Each one runs from `0` to `1`, defaults to `0.5`, and accepts modulation. The effect depends on the engine you named. On `plmodal` the three set the shape of the resonating body, the brightness of the strike, and the length of the ring. On `plchord` the same three pick the chord, the inversion, and the waveform that plays it.

The drums use the same three tone slots, and each drum model gives them names of its own (see *Drum synthesis* (2.5)). The complex oscillators skip the aliases and take the generic names. Ten engines with private names for three slots would give you thirty words to remember, and no two sets would agree.

```
plchord snd c2 note 0.3 harmonics 0.7 morph .
```

The table is the whole map. Read across a row to learn one engine.

Source	Character	`harmonics`	`timbre`	`morph`
`plmodal`	struck resonant body	structure of the partials	brightness of the strike	damping, the length of the ring

Source	Character	harmonics	timbre	morph
plva	virtual analog	detuning between two oscillators	variable square	variable saw and sync
plws	waveshaping	waveshaper curve	fold amount	waveform asymmetry
plfm	two-operator FM	frequency ratio	modulation index	feedback
plgrain	granular formants	formant ratio	formant frequency	formant width
pladd	additive partials	number of spectral bumps	which harmonic carries the bump	bump shape
plwt	wavetable	bank selection	row in the bank	column in the bank
plchord	four-note chord	chord type	inversion and transposition	waveform
plswarm	swarm of eight saws	spread of the pitches	grain density	grain duration and overlap
plnoise	clocked filtered noise	filter type, lowpass to highpass	clock frequency	resonance

2.3.2 Playing them

Everything downstream works as usual. The voice envelope, the filters, the insert chain and the orbit sends all apply on top. A complex oscillator is a source like any other, and you shape it with the same words.

```
plswarm snd c2 note 600 lpf 0.4 verb .
```

`plmodal` acts like a struck object, not a held tone. Each note is one strike, and the ring decays by itself even when you hold the envelope open. `sustain` therefore sets how much of the tail you keep, not how long the sound lasts.

```
plmodal snd c2 note 0.9 morph 3 sus .
```

Modulation is the real reason to reach for these engines. A slow sweep through `timbre` or `morph` moves a complex oscillator through

territory a filter cannot reach. You change how the sound is made, not what a filter removes from it afterward.

```
plwt snd c3 note 0 1 8 lfo timbre .
```

2.3.3 Two things to know

The engines make sound in blocks of sixteen samples, not one sample at a time. Cagire therefore reads the pitch and the three slots three thousand times a second, not at the sample rate. Vibrato, envelopes and any modulation you hear as movement stay far inside that budget, so you will not notice the block rate. It does rule out one thing: audio-rate frequency modulation of a complex oscillator. Everything after the source, the filters and the whole insert chain, still runs per sample.

These engines expect a 48 kHz sample rate. Pitch comes out right at any rate, because Cagire tells the engine which rate it runs at. The decay and smoothing times inside the engine scale with the difference instead. At 44.1 kHz they run about nine percent long. Run your audio device at 48 kHz to hear the engines as their authors designed them.

2.4 Vocal synthesis

`vox` is a physical model of the human voice, in the family of Neil Thapen's Pink Trombone. A glottal pulse drives a waveguide vocal tract: 44 tube segments for the mouth and 28 more for the nose. There are no vowel presets and no formant tables. Vowels emerge from the shape of the tube, and between the vowels live diphthongs, mumbles and creature noises.

```
vox snd e2 note 2 sustain .
```

Pitch comes from `note` as usual, and vibrato through `vib` reaches the glottis directly. The voice sits low by default; it is a bass tract.

2.4.1 The tongue

Two words shape the tract. `tonguex` is the tongue's position from back (`0`) to front (`1`), and `tonguey` is its height from closed (`0`) to

open (1). Both default to a resting mouth. A static setting picks a vowel-like color; the sound is in moving them:

```
vox snd a2 note
sine 0.7 osc tonguex
tri 1.1 osc 0.2 1 range tonguey
4 sustain .
```

The tract follows the tongue at articulator speed, about thirty milliseconds, so even a jump lands as a syllable rather than a click. `drift` babbles on its own:

```
vox snd g2 note
drift 0.3 osc tonguex
drift 0.4 osc 0.2 1 range tonguey
6 sustain .
```

`tx` and `ty` are the short aliases.

2.4.2 Breath and nose

`tense` is the glottal tenseness, from breathy whisper at `0` to pressed at `1`. It reshapes the glottal pulse and mixes in aspiration noise at the same time, so low values whisper and high values buzz. The default is `0.6`.

`velum` opens the nasal port. At `0` the nose is sealed; around `0.4` the voice goes fully nasal, the way an *m* or *n* sits in the mouth. Open it while closing the tongue and the tract hums through the nose alone:

```
vox snd c3 note 0.15 tense 0.4 velum 3 sustain .
```

Everything downstream applies as usual: the envelope, the filters, the insert chain and the orbit sends. The *Vowel formant filter* (7.2.4) is a different, cheaper road to vowels that works on every source; `vox` is the one whose mouth actually moves.

2.5 Drum synthesis

Cagire can synthesize digital drum voices like a regular drum machine that you would buy in a store. Why? Because most people start playing Cagire without a sample collection and drumming is

one of the first things you might want to do on a step sequencer! These digital drum models are not incredible per se, but they happily fill a gap. Use them until you find how to make better drums with synthesis, audio patching or samples. Each drum is already tuned with defaults that make it easily recognizable. Each drum also decays in two moves, a fast body and a slower tail, with a `body` default that fits the model (see *Voice envelope* (6.1)). Here is a good old kick:

```
kick snd .
```

There are nine drums: `kick`, `snare`, `hat`, `tom`, `rim`, `cowbell`, `cymbal`, `plate` and `clap`. Every one of them answers to the same three parameters, and to `drumkit`, which chooses the machinery behind them.

2.5.1 The three macros

`shape`, `punch` and `grit` mean the same three things on every drum. What they move underneath changes from drum to drum and from kit to kit. What they do to your ear does not. Turn `punch` up and the hit gets harder, whichever drum you are playing.

Parameter	What it does
`shape`	spectral structure: sweep depth, FM ratio, inharmonic spread
`punch`	attack energy and brightness
`grit`	texture: saturation, noise, feedback, ring

All three run from `0` to `1` and all three take modulation. An `lfo` or a touch of `jit` walks a drum's texture across a pattern:

```
hat snd 0 1 4 lfo grit .
```

The generic slot names still work as aliases, so `morph` is another way to write `shape`, `harmonics` another way to write `punch`, and `timbre` another way to write `grit`. They are handy when you want one phrase to drive a drum and a `pl*` oscillator alike:

```
kick snd 1 harmonics .
```

Four drums, `kick`, `snare`, `tom` and `rim`, take a fourth parameter named `wave` on the analog kit. It sets their oscillator shape, from `0` sine through `0.5` triangle to `1` sawtooth.

2.5.2 Kits

`drumkit` picks which set of models renders the drums.

Index	Kit	Character
`0`	analog	The TR-style set. Swept oscillators, banks of inharmonic squares, filtered noise, a struck modal resonator.
`1`	fm	The industrial set. Two-operator FM for the pitched drums, ring modulation for the metal. Denser and harder.
`2`	west	The modular set. Six sine partials whose spacing runs from harmonic to inharmonic, summed and driven into a wavefolder.
`3`	chip	The console set. A shift register for noise and a quantised pulse, after the NES. Digital logic on a fixed grid rather than a continuous waveform.
`4`	skin	The acoustic set. Membranes, bars and a bell as banks of struck modes. The metal is the 808 stack under a stick.
`5`	hard	The hardcore set. A pulse or noise into a resonant filter pushed into oscillation, then a rectifier and a drive.
`6`	tube	The waveguide set, after the Volca Drum. A short waveform, folded and driven, into a delay-line body. A tube rings odd harmonics and a string rings all of them. `shape` sizes the body.
`7`	shake	The shaken set, Cook's PhISEM. Random collisions excite the resonances of a material. The objects are a can of beads, a cabasa, a maraca, water drops, sticks, a wrench, sleigh bells, chimes and a tambourine.

```
kick snd 1 drumkit .
```

The index wraps, so any number you hand it plays something. With eight kits installed, `43 drumkit` is kit 3. That means you can drive it straight from a pattern without checking the bounds first:

```
hat snd 3 drumkit .
```

The kits are level-matched to each other, so switching one in the middle of a pattern does not move your mix. Only the timbre changes.

`drumkit` is the one drum parameter that takes no modulation. It swaps the generator, and switching one part-way through a hit would tear the filter and feedback state the drum is holding.

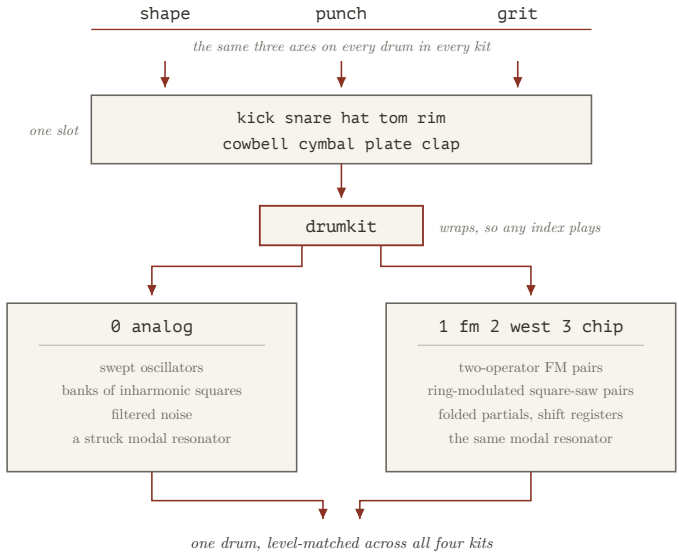


Figure 61: The three macros and a slot name go in, drumkit picks which of the two kits renders that slot, and both kits fill all nine slots at matched levels.

Each drum has a resting pitch it uses when you name no `freq`, and the two kits sit close but not identical. On the analog kit the kick rests at `55` Hz, the tom at `120`, the snare at `180`, the rim at `400`, the hat, cymbal and plate at `420`, the cowbell at `540`, and the clap at `900`. The fm kit drops the kick to `50` and the tom to `110`, and lifts the snare to `190`, the rim, hat, cymbal and plate to around `440`, and the cowbell to `560`.

Every drum answers to a short alias as well as its full name. `sd` is the snare, `hh` and `hihat` the hat, `rimshot` and `rs` the rim, `cb` the cowbell, `cy` the cymbal, `plt` the plate, `cp` the clap.

2.5.3 Kick

A pitched body that falls in pitch as it hits.

On the analog kit, `shape` is how far the pitch falls, up to four octaves, `punch` is how fast it falls together with the beater click, and `grit` is a saturation that is never fully off. Lower the `freq` for a deeper drum:

```
kick snd 45 freq 0.6 shape 0.4 punch .
```

Reach for `wave` and `grit` when you want the body harder and dirtier than a clean sine:

```
kick snd 0.5 wave 0.3 shape 0.5 grit .
```

On the fm kit the pitch drop is much shallower, because the attack comes from somewhere else. `shape` sets the modulator ratio, from a sub-octave that fattens the body to a third harmonic that hollows it out. `punch` sets how deep and how fast the modulation collapses, and that collapse is the click. `grit` feeds the carrier back into itself, running it from a sine toward a sawtooth:

```
kick snd 1 drumkit 0.8 punch 0.7 grit .
```

On the skin kit the body is a membrane struck by a pulse, with a beater of lowpassed noise. A hard hit stretches the head, so the pitch starts high and falls over 30 ms. `shape` sets how long the overtones ring, `punch` the glide depth and the beater, `grit` the skin saturation:

```
kick snd 4 drumkit 0.3 shape 0.7 punch .
```

On the hard kit the cutoff of a screaming filter falls five octaves in 15 ms. The highpass is the click, a second ring the knock, a hard clip the tail. `shape` is the drop, `punch` its speed and the knock, `grit` the scream and the clip:

```
kick snd 5 drumkit 0.8 shape 0.9 grit .
```

On the tube kit a dropping sine and a click go into a tube tuned to the root. `shape` sizes the tube, `punch` the drop and the click, `grit` the fold, the drive and how long the tube rings:

```
kick snd 6 drumkit 0.3 shape 0.8 grit .
```

On the shake kit it is a can with a few dozen beads in it, Cook's coke can tuned an octave and a half down, with resonances at 55, 152, 212 and 320 Hz:

```
kick snd 7 drumkit 0.7 punch .
```

2.5.4 Snare

A tuned shell crossed with a rattle.

On the analog kit the rattle is filtered noise. `shape` moves the upper partial of the shell, `punch` opens the noise brightness, and `grit` balances body against noise:

```
snare snd 0.8 grit 0.7 punch .
```

Lean back toward the body and the snare rounds out:

```
snare snd 0.2 grit 0.9 punch .
```

On the fm kit the rattle is a feedback operator rather than noise, so it tracks the pitch of the drum instead of sitting in a fixed band. Move the `freq` and the rattle moves with it. `grit` is how much of that metallic hash lands:

```
snare snd 1 drumkit 0.8 grit 0.6 shape .
```

On the skin kit six shell modes are struck once. The wires are noise driven by the shell. `shape` spreads the shell, `punch` is the strike hardness, `grit` the wire tension and amount:

```
snare snd 4 drumkit 0.7 grit .
```

On the hard kit the shell sits on one ring and a band of noise is pinged into the other, with the crack on top. `shape` and `punch` move the ping, `grit` the scream and the drive:

```
snare snd 5 drumkit 0.8 punch 0.7 grit .
```

On the tube kit a short sine and a burst of lowpassed noise go into a string tuned to the root. `shape` sizes the string, `punch` the burst and the brightness, `grit` the fold and the ring:

```
snare snd 6 drumkit 0.7 shape 0.6 grit .
```

On the shake kit it is a cabasa with five hundred beads on a gourd. `shape` thins them out:

```
snare snd 7 drumkit 0.3 shape .
```

2.5.5 Hat

A bright wash with no low body. That absence is what separates it from the cymbal.

On the analog kit six inharmonic squares beat against each other and run through a bright band. `shape` spreads their tuning from a tight cluster to a wide clang, `punch` moves the band, and `grit` opens the hat from a tight tick to a loose wash:

```
hat snd 0.6 shape 0.8 punch .
```

```
hat snd 0.2 shape 0.9 punch 0.7 grit .
```

On the fm kit three ring-modulated pairs, a square against a saw, are summed. Both operators are harmonically rich, so the wash comes from the pairs themselves and needs no waveshaper behind it. `shape` spreads the pairs apart, from a single pitched ring at the bottom to a wide hash at the top, and `grit` opens the hat as it does on the analog kit:

```
hat snd 1 drumkit 1 grit 0.1 decay .
```

On the skin kit the 808 stack runs under a grain of noise, the contact of the stick, plus a short stick transient. `grit` opens the hat as on the analog kit:

```
hat snd 4 drumkit 0.6 grit .
```

On the hard kit noise is pinged into a high ring. `shape` turns the tick into a whistle, `punch` moves it up, `grit` opens the hat and drives it:

```
hat snd 5 drumkit 0.8 shape 0.5 grit .
```

On the tube kit highpassed noise goes into a very short string. `shape` sizes the string, `grit` opens the hat and lengthens the ring:

```
hat snd 6 drumkit 0.8 shape 0.7 grit .
```

On the shake kit it is a maraca. `shape` runs it from a few beans to a hundred, `grit` keeps it moving:

```
hat snd 7 drumkit 0.6 shape 0.8 grit .
```

2.5.6 Tom

A pitched body like the kick but gentler, with a touch of stick noise.

On the analog kit `shape` and `punch` shape the pitch fall the way they do on the kick, and `grit` sets the stick-noise amount:

```
tom snd 0.5 shape 0.4 grit .
```

Drop the `freq` for a low floor tom:

```
tom snd 90 freq 0.7 shape .
```

On the fm kit the body is a sine under a pitch drop. The modulator sits at the membrane's second mode and lives only for the first few milliseconds, the strike. `shape` is the drop, `punch` the strike's brightness and speed, `grit` the feedback in the strike and the stick noise:

```
tom snd 1 drumkit 0.6 shape 0.5 punch .
```

On the skin kit the fundamental of a membrane sits on one bank and its five overtones on another, struck by one pulse. `shape` is the glide depth, `punch` the glide speed and the overtone amount, `grit` the stick noise:

```
tom snd 4 drumkit 0.6 shape 0.7 punch .
```

On the hard kit it is the kick's ring with a shallower, slower drop, and a stick on the second ring:

```
tom snd 5 drumkit 0.5 shape 0.8 grit .
```

On the tube kit it is the kick's recipe with a shallow drop into a longer, darker tube:

```
tom snd 6 drumkit 0.2 shape 0.9 grit .
```

On the shake kit it is water drops into a basin, with three sharp resonances:

```
tom snd 7 drumkit 0.4 shape 1 decay .
```

2.5.7 Rim

A short click, over almost before you hear it.

On the analog kit it is two tuned partials and a noise click, with no sweep: a woody tock. `shape` moves the upper partial, `punch` sets the click's edge, and `grit` decides how long it rings on:

```
rim snd 0.5 shape .
```

```
rim snd 0.3 punch 0.9 grit .
```

On the fm kit it is a high-ratio burst with a fixed modulation index: a clank rather than a tock. `shape` walks the ratio from 2 up to 8, which is what turns a woodblock into a length of pipe:

```
rim snd 1 drumkit 0.9 shape .
```

On the skin kit a free bar of four modes is struck hard. `shape` moves the bar from the stick to the shell, `grit` the ring length:

```
rim snd 4 drumkit 0.8 shape 0.6 grit .
```

On the hard kit two short rings sit a fixed interval apart. `shape` moves the upper ring, `grit` the ring length and the drive:

```
rim snd 5 drumkit 0.9 grit .
```

On the tube kit a saw click goes into a short string above the root. `shape` moves the string from twice to five times the root:

```
rim snd 6 drumkit 0.9 shape .
```

On the shake kit it is two sticks hitting each other:

```
rim snd 7 drumkit .
```

2.5.8 Cowbell

A struck bell that rings on two pitches at once.

On the analog kit two detuned squares run through a bandpass. `shape` sets the detune, `punch` moves the bandpass centre, and `grit` adds metallic bite:

```
cowbell snd 0.6 shape .
```

```
cowbell snd 700 freq 0.8 grit .
```

On the fm kit the ratio sits on the same interval, but a fixed modulation index fills the gap between the two partials that squares leave empty. `punch` buys the clank on the onset, `grit` the density:

```
cowbell snd 1 drumkit 0.9 punch 0.7 grit .
```

On the skin kit four stiff modes on the 808 interval are struck once, then saturated. `shape` spreads the interval, `punch` the onset accent, `grit` the saturation:

```
cowbell snd 4 drumkit 0.8 grit .
```

On the hard kit the 808 interval sits on the two rings, rectified, then the bell's band. `grit` holds the rings open:

```
cowbell snd 5 drumkit 0.9 grit .
```

On the tube kit two saws on the 808 interval go into a tube at the root. `grit` is the fold and the ring:

```
cowbell snd 6 drumkit 0.9 grit .
```

On the shake kit it is a wrench, shaken:

```
cowbell snd 7 drumkit 0.7 grit .
```

2.5.9 Cymbal

The hat with a body under it. Both kits keep that relationship.

On the analog kit the hat's six squares are split across two bandpass paths, one bright and long, one low and short. `shape` spreads the tuning, `punch` moves both bands, and `grit` weights the upper path against the lower:

```
cymbal snd 0.7 grit .
```

```
cymbal snd 0.9 shape 0.8 grit .
```

On the fm kit the ring-modulated pairs are spread wider and split the same way:

```
cymbal snd 1 drumkit 0.6 shape 2 decay .
```

On the skin kit the bright path peaks about 25 ms after the hit, with a stick transient in front:

```
cymbal snd 4 drumkit 0.7 grit .
```

On the hard kit noise runs into two high rings, one of them held at the edge of oscillation, driven and highpassed. `shape` pitches the wash, `grit` weights the held ring:

```
cymbal snd 5 drumkit 0.8 shape 0.6 grit .
```

On the tube kit a long band of noise, folded, goes into a string tuned off the harmonic grid. `shape` detunes the string, `grit` holds it:

```
cymbal snd 6 drumkit 0.5 shape 0.8 grit .
```

On the shake kit it is sleigh bells, two on each side:

```
cymbal snd 7 drumkit 0.9 grit .
```

2.5.10 Plate

A struck sheet of metal. Sixteen modes ring at once on ratios that never line up into a harmonic series, which is why the plate reads as metal and not as a note.

Nothing else here rings this long. `grit` runs from a fifth of a second of clank out to four seconds of sustain and rests in the middle, so shortening it changes the drum as much as opening it up:

```
plate snd 0.6 grit .
```

```
plate snd 0.1 grit 0.7 punch .
```

`shape` stretches the mode ratios apart, from something close to a bell out to a wide clang. Spread them wide and keep the tail and you have a gong rather than a plate:

```
plate snd 0.9 shape 0.85 grit .
```

Both kits ring the same sixteen modes. They differ in how the modes are struck. The analog kit strikes with noise, which has energy everywhere and so wakes every mode equally. The fm kit strikes with a ring-modulated burst, which has energy at a handful of inharmonic places, so it wakes the modes it happens to land on and leaves the rest quiet. `shape` moves which ones it lands on:

```
plate snd 1 drumkit 0.4 shape 0.9 grit .
```

Lower the `freq` and the whole bank moves down with it, which is how one model covers everything from a small sheet to a bell. Its high partials always fade before its low ones, the way they do on real metal, so a long tail darkens as it decays instead of holding its colour.

The skin kit rings the modes of a rectangular plate under a mallet of lowpassed noise. Two of its modes sit near unison and beat slowly. `punch` is the mallet hardness and leaves the pitch alone:

```
plate snd 4 drumkit 0.9 punch 0.8 grit .
```

The hard kit holds one ring per side near the edge of oscillation, two percent apart. `shape` adds the rectified octave, `punch` moves the pitch, `grit` the hold:

```
plate snd 5 drumkit 0.6 shape 0.9 grit .
```

The tube kit sends a mallet of noise into a string that barely decays, read at two taps for width. `shape` sizes the string, `punch` the mallet and the brightness, `grit` the hold:

```
plate snd 6 drumkit 0.4 shape 0.9 grit .
```

The shake kit gives bamboo chimes. Three tubes ring for a long time, about one collision at a time:

```
plate snd 7 drumkit 0.2 shape 4 decay .
```

2.5.11 Clap

A clap is more than one hit. It is a short burst of taps a few milliseconds apart, the sound of several hands that do not land together. A wash follows and fills in behind them.

`shape` sets how far apart those taps fall, `punch` moves the band-pass they run through, and `grit` sets how long the wash hangs on:

```
clap snd 0.6 punch .
```

Tight taps and a short wash give a dry, single-hand clap:

```
clap snd 0.2 shape 0.2 grit .
```

Loosen the spacing and stretch the wash for a room full of people:

```
clap snd 0.8 shape 0.9 grit 0.4 punch .
```

The fm kit keeps the tap structure, because that structure is what makes a clap a clap, and fills it with ring-modulated hash instead of noise. The result is a metallic slap:

```
clap snd 1 drumkit 0.8 grit .
```

The skin kit puts the four taps on pink noise through a lower band, with a resonance at 260 Hz for the cupped hands:

```
clap snd 4 drumkit 0.7 grit .
```

The hard kit fires the four taps into one ring per side. The ring gives the clap a tone. `punch` moves the tone, `grit` the ring and the drive:

```
clap snd 5 drumkit 0.6 punch 0.8 grit .
```

The tube kit fires the four taps into a tube around the clap band, wide by a second tap. `punch` moves the band, `grit` the fold and the ring:

```
clap snd 6 drumkit 0.6 punch 0.8 grit .
```

The shake kit gives a tambourine, a shell under its jingles:

```
clap snd 7 drumkit 0.8 punch .
```

2.6 Noise sources

Three noise sources give you the unpitched raw material for percussion, wind, texture, and anything you want to carve with a filter. They differ only in how their energy is spread across the spectrum. `white` has equal energy at every frequency, so it is the brightest of the three. `pink` rolls off toward the highs, with equal energy per octave. `brown` rolls off harder still, leaving most of its energy in the low end.

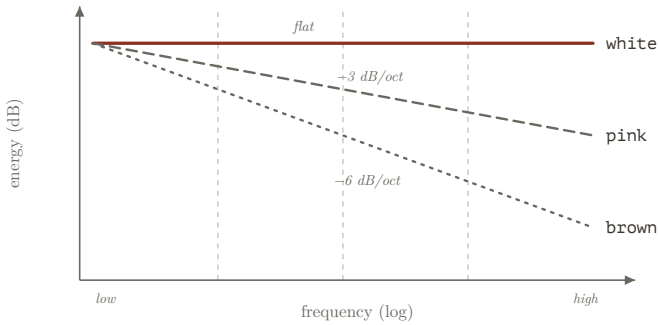


Figure 62: Spectral slope sets the three colours apart: white is flat, pink rolls off at about −3 dB per octave, brown at about −6 dB.

```
white snd 0.3 gain .
```

Noise ignores pitch, so `note` and `freq` do nothing to it. That is the nature of noise: it spreads energy across the whole spectrum rather than sitting on one frequency. You shape it by filtering instead. A lowpass on its own already pulls white noise down toward a wind-like texture:

```
pink snd 4000 lpf 0.4 gain .
```

More happens once the filter moves. Sweep a lowpass with an `lfo` over pink noise and the texture shifts continuously:

```
pink snd 400 4000 4 lfo lpf 0.4 gain .
```

A slow ratio-aimed `rise` up the spectrum, fed into a little reverb, builds a riser, the rising filter sweep used to lead into a section:

```
white snd 2 stepsecs rise 200 8000 expmap lpf 0.4
verb 0.3 gain .
```

A resonant bandpass picks a pitch out of the hiss. Sliding it turns that pitch into a moving tone:

```
white snd 300 6000 2 slide bpf 0.9 bpq 0.3 gain .
```

A band of filtered noise is also the seed of most synthetic percussion. Noise feeds nicely into a comb or a resonant filter when you want a pitched, plucked character out of it (see *Comb* (8.4) under Orbit effects).

2.7 Audio samples

Alongside the synthesized sources, Cagire plays recorded audio. Point it at a folder of sound files and each folder becomes a source you name like any other, so your own kicks, breaks, and field recordings sit right next to `saw` and `kick` in a script:

```
break snd .
```

A folder of files becomes a bank named after the folder, and the files inside it are numbered from `0`. You pick between them with `n`, covered in *Selecting samples* (5.1), and *Loading samples* (5.1) explains how a folder gets registered and which formats Cagire reads.

2.7.1 Pitched playback

A sample plays back at its recorded speed by default, but the engine tracks pitch the moment you ask for one. Its base note is middle C, so a `note` or a `freq` transposes from there:

```
break snd c4 note .
```

That plays the sample as if middle C were its home note, shifting it up or down to land on the pitch you name. Note names and raw frequencies both work, the same as on any oscillator. Changing

pitch this way also changes the playback speed; *Playback speed* (5.2) covers how to separate the two.

2.7.2 Bank variations

Files that share a base name with a `_suffix`, like `break.wav` and `break_hard.wav`, collapse into one source. The plain name plays the unsuffixed file, and `bank` reaches a variation by its suffix:

```
break snd hard bank .
```

This keeps a family of related hits under one name, so you can switch between a soft and a hard version from a pattern without renaming the source. From here, *Selecting samples* (5.1) covers choosing files with `n`, *Playback speed* (5.2) covers `speed` and `stretch`, and *Looping samples* (5.3) covers windowing a sample, slicing it, and fitting it to the beat.

2.8 Live audio

`live`, also `mic`, brings your audio input into the engine. It can be a microphone or whatever your interface is feeding in. It treats that source like any other. Cagire only listens once you have turned audio input on (see *Audio input* (6.1)), and then a plain `live snd .` passes it straight through. If your input is a microphone, wear headphones. A live mic playing back through speakers in the same room feeds into itself and howls:

```
live snd .
```

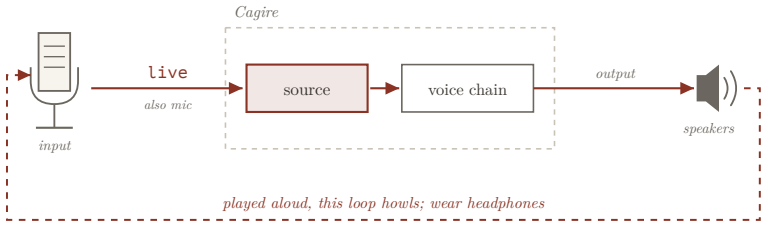


Figure 63: Audio input enters through live (also mic) and runs the full voice chain to the speakers; in the same room it loops back into the mic and howls.

Once it is a source, everything you can do to an oscillator can also be done to the live input. The whole effect chain and the envelope apply to it, so you can filter it, send it to reverb, or gate it into rhythm:

```
live snd 800 lpf .
```

If your interface has more than one input, `inchan` chooses which channel to take, counted from `0`. Raise **In Channels** in the Audio tab first: Cagire opens two channels unless you ask for more, and `inchan` can only reach the channels that are open (see *Audio input* (6.1)):

```
live snd 0 inchan .
```

Live input pairs naturally with recording. It is just recording pointed at the outside world. Capture a phrase as it comes in, then play it back as a sample (see *Live recording* (5.6)).

2.9 Soundfonts

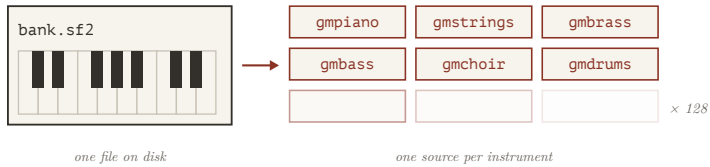


Figure 64: One .sf2 file opens into a whole bank of sources, one gm word per instrument, with the drum kit on its own bank.

A soundfont is a bank of sampled instruments in an `.sf2` file, the General MIDI sound set most people know from old hardware and game music. Cagire ships with one, which gives you pianos, strings, brass, and a full drum kit from a standing start, without leaving the engine. A soundfont behaves like any other source once it is loaded, and the built-in one is loaded before you play your first note. Put an `.sf2` of your own in a folder Cagire already watches and it plays that instead. *Loading soundfonts* (5.4) in the interface section walks through it, and *Loading samples* (5.1) covers registering those

folders in the first place. This page is about playing what you have. A soundfont instrument is a source whose name glues `gm` to the instrument you want, by name or by its General MIDI number from `0` to `127`: `gmpiano`, `gmstrings`, `gm48`. Bare `gm` is the piano, program `0`:

```
gmpiano snd c4 note .
```

That plays a piano at middle C. There is a second way to say the same thing: leave the source as bare `gm` and hand the instrument to `n`, so `gm snd piano n` and `gm snd 48 n` reach the same presets as `gmpiano` and `gm48`. It is the form to use when you want to pattern the instrument itself. Swap the name for any other instrument and add a `note` to pitch it:

```
gmstrings snd c3 note 2 decay .
```

2.9.1 Drums

The percussion kit lives on its own bank, under the source `gmdrums` (or `gmpercussion`), and each MIDI note triggers a different piece: `36` a kick, `38` a snare, `42` a closed hat:

```
gmdrums snd 36 note .
```

```
gmdrums snd 42 note .
```

2.9.2 Envelopes

A soundfont carries its own envelope for each preset, and Cagire uses it, which is why an instrument already sounds shaped before you touch it. Set an `attack`, `decay`, or `sustain` of your own and yours takes over; anything you leave alone keeps the soundfont's value. `release` is the exception, since Cagire fills one in on every emit that does not carry one, so the preset's own release never gets a turn. Name a `release` you want rather than expecting the soundfont's:

```
gmpiano snd 0.01 attack 0.3 decay 2 release .
```

2.9.3 Running with it

The point of a soundfont in Cagire is that it is just a source. Once you have named it, the whole engine is yours: the filters, the distortion, the orbit effects, modulation, polyphony. A clean General MIDI piano becomes a different instrument the moment you push it through a filter and some reverb:

```
gmepiano snd c3 note 1200 lpf 0.4 verb .
```

Drive a bass into distortion for grit it never shipped with:

```
gmbass snd c2 note 0.5 distort .
```

Hand a pad several notes at once and the soundfont voices a chord, then a moving filter and a little delay give it life:

```
gmpad snd
c3 e3 g3 note
1000 4000 4 lfo lpf
0.4 delay
.
```

Send a choir into reverb and widen it for a sustained, wide pad:

```
gmchoir snd
c4 note
0.6 verb
1.5 width
.
```

There is nothing special about a soundfont voice once it is sounding, so treat it as a starting point and run with it. The full list of instrument names is in the *General MIDI presets* (B) appendix.

Chapter 3

Managing voices

3.1 What is a doux voice?

Every sound Cagire makes is modelled as *a voice*, one self-contained instrument inside the engine. A voice bundles a sound source (an oscillator, a sample, a drum model, and so on) with its own envelope, filters, and effects. When you emit a sound, the engine hands the sound register to Doux, which builds a voice from it and starts it playing. From there the voice runs on its own, and when its envelope reaches zero it frees itself and returns its slot to the pool.

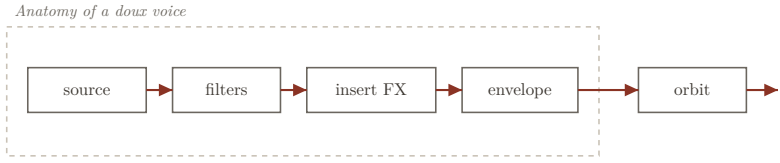


Figure 65: A voice runs its source through its own processing and effects, then feeds an orbit.

```
saw snd
c4 note
0.8 gain
0.3 verb
.
```

That one `.` spins up one voice: a sawtooth at middle C, a little quieter, with a touch of reverb, sounding until its envelope finishes.

3.1.1 Finite and infinite voices

Most voices are finite. They start, they sound, they fade, they free themselves. Even a bare `sine snd .` ends, because Cagire gives a synth a four-step `gate` and a one-step `release`. A sample gets a whole pattern loop and a twelve-step release, while native drums and GM instruments keep the envelopes supplied by their source. `gate` sets how long the note is held. Set it to `0` and the voice never closes on its own: it sustains forever, an infinite voice you manage by hand:

```
saw snd c2 note 0 gate 0 voice .
```

Let's end this voice for now:

```
saw snd c2 note 1 gate 0 voice .
```

That drone holds until you stop it manually or its slot is taken. Infinite voices are how you build pads, drones, and held basses, but each one spends a slot the whole time it sounds, so you keep track of the ones you leave running (see *Monophonic & held voices* (3.3)).

3.1.2 The voice budget

The engine plays a fixed number of voices at once, `64` by default (mainly for safety reasons, you can push it waaay up). That budget is shared by everything sounding: a six-note chord is six voices, a fast run of plucks can be dozens at a time. You can raise the ceiling and watch the active and peak counts on the Engine view. When you ask for more voices than the budget allows, the engine steals one to make room, and it picks carefully rather than taking the oldest. Voices already releasing go first, then voices holding at their sustain, and voices still in their attack go last; within a group the quietest one loses. A burst at the ceiling therefore eats its own fading tails before it eats the notes you just played, and a held drone goes before a fresh hit does. Keep an eye on the peak, and lean on cut groups when you want voices to replace each other on purpose (see *Cut groups* (3.2)).

3.1.3 Tagging a voice

Every . either creates a fresh voice, the default, or speaks to a voice you have already tagged. The difference, and what an emit does to a voice that is already sounding, is what the rest of this page is about. `voice` tags a voice with a stable ID. Emitting with the same tag later addresses that exact voice for as long as it sounds, no matter what else starts or dies around it. This is how you reach back into a long-running or infinite voice and change it from another step:

```
saw snd c2 note 0 voice 16 gate .
```

The tag is just a number you choose. Pick one per voice you want a handle on, and reuse it whenever you want to talk to that voice again.

3.1.4 Update versus retrigger

What a tagged emit does depends on whether you name a source. Without a sound name, the emit updates the voice: the parameters you name move, but the envelope, the oscillator phase, and the sample position are left untouched. The one exception is `gate`: an explicit gate retargets the envelope, so a positive `gate` ends a held voice while `0 gate` holds it open. If nothing is sounding under that tag, the emit is dropped. This is how you sculpt a held voice while it plays:

```
saw snd c2 note 0 voice 16 gate .
```

Then let's add a filter:

```
800 lpf 0 voice .
```

And a little bit of modulated frequency modulation:

```
1.99 2.11 1 lfo fm 0 voice .
```

With a sound name, the emit retriggers the voice: the envelope refires, click-free from its current level, and a sample restarts from the top. The parameters stay sticky, so only the ones you name change:

```
saw snd 0 voice .
```

Add `1 reset` when you want the opposite. It wipes the voice back to the engine defaults before this emit's parameters land, so everything it was still carrying from earlier steps, filters, effects, modulation, is gone and only what you name survives:

```
saw snd c2 note 1 reset 0 voice .
```

A voice in release keeps releasing while you update it. Once it has faded out its tag is gone, and any further updates are dropped.

WORD	RANGE	SETS
`voice`	any number	the tag a later emit addresses this voice by
`gate`	steps, 0 holds	how long the note is held, 4 steps when you set none
`reset`	1 to wipe	clears the voice's sticky parameters on a retrigger

See *Voice envelope* (6.1) for the shape every voice runs through, and *The sound register* (2.5) for how the register feeds each emit.

3.2 Cut groups

One sound command can fan out into many voices, one per value you stack before a parameter, so a chord is just several pitches on `note` at once and the longest list decides the count. That fan-out is *Polyphony* (3.5), covered in full there. This page is about the other side of playing many voices, cut groups, where voices choke each other off on purpose. Every voice also spends one slot of the engine's budget, so cutting is one way to keep a thick, fast part from stealing its own notes (see *What is a doux voice?* (3.1)).

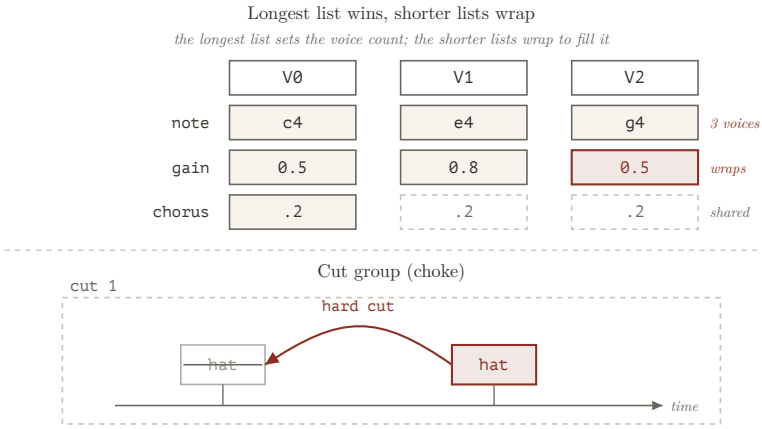


Figure 66: Above, one sound command fans out into one voice per value: the longest list sets the count and shorter lists wrap to fill it. Below, two hats share a cut group and the second one chokes the first.

3.2.1 Choking on purpose

Sometimes you want voices to cut each other off on purpose. `cut` puts a voice in a choke group, a numbered slot where only one sound may ring at a time. When a new voice joins a group it takes over the slot and silences whatever was already playing there:

```
hat snd
1 cut
.
```

The classic case is the hi-hat: an open hat should die the instant the closed hat hits, never ring through it. Put both in the same group and the second emit chokes the first. Here the long, metallic hat is the open one and the short, tight hat that follows cuts it dead:

```
hat snd
0.9 metal 0.9 bright
2 decay
1 cut
.

hat snd
0.2 metal
0.06 decay
1 cut
.
```

Groups are plain numbers you choose, and `0` is a group like any other rather than an off switch: leave `cut` out entirely and the voice belongs to no group. The groups are engine-wide, not per-orbit, so two voices on different orbits sharing a number still choke each other.

`cut` works on any source, synth or sample, and it is also a blunt way to keep a part monophonic, since each new hit kills the last. For a cleaner monophonic line that glides between notes, reach for a held voice instead (see *Monophonic & held voices* (3.3)). Cut groups are really important if you intend to chop and mangle samples a lot!

3.3 Monophonic & held voices

By default every emit is a fresh voice, born and freed on its own. Tag a voice with `voice` and you can speak to it again, which unlocks two things a fresh-every-time voice cannot do: holding a note open to build pads, drones, and evolving basses, and keeping a line monophonic so each note takes over from the last. The mechanism behind both, how a tagged emit updates or retriggers a voice and how `0 gate` holds one open, is in *What is a doux voice?* (3.1). This page is about putting it to use.

3.3.1 Held voices

An open gate, `0 gate`, holds a tagged voice instead of letting it fade, and from any later step you reach back by its tag to move its parameters:

```
saw snd c2 note 0 voice 0 gate .
```

Change its filter while it holds, with no source name so the voice updates rather than restarts:

```
800 lpf 0 voice .
```

When you are done, hand the tag a positive `gate` and it releases; raise `release` in the same line for a gentler fade. That is how you build an evolving pad or bass: set it going once, then sculpt it live:

```
0 voice 1 gate .
```

3.3.2 Monophonic lines

A monophonic line is one voice that never plays more than one note at a time. These can be quite helpful to build certain types of sounds: basses, leads, sirens, etc. You can create a monophonic voice by reusing one tag for every note. Each emit lands on the same voice, so a new note takes over from the last instead of stacking on top of it. Try to evaluate these two examples very quickly:

```
saw snd c2 note 0 voice .
```

```
saw snd eb2 note 0 voice .
```

Both notes are voice `0`, so the `eb2` retriggers that same voice rather than adding a second one. Spread the idiom across the steps of a pattern and you have a true monophonic bassline, no matter how fast it runs. To stop the line, let its last note fade on its own gate, or, if you are holding it open with `0 gate`, end it with a positive `gate` exactly as you would any held voice.

3.3.3 Glide & portamento

`glide` shows itself best on the voices this page is about, since it only bites when one note follows another on the same tag. Arm it on a held voice and each re-pitch slides instead of jumps:

```
saw snd c2 note 0 voice 0.5 glide 0 gate .
```

```
g3 note 0 voice .
```

The word itself, its range and its sticky behaviour, is covered in *Spread, detune & glide* (4.1).

3.4 Gain and amplitude

A voice has two main volume controls, and where they sit in the signal path is the whole point of having both. `gain` is the first gain control applied to the source, before the filters and the distortion. `postgain` is one of the last, after the effects. Between them the filters, the distortion, other effects and the envelope all do their work. The two controls do not mean the same thing even though both make the voice louder or quieter. `gain` runs from `0` and defaults to `1`. You can think of it as the initial gain of your sound source:

```
square snd 0.3 0.8 rand gain 2 decay .
```

Push `gain` up and you drive the later stages harder, not just louder, so the same `distort` bites differently depending on it; *What are insert effects?* (7.1) covers why the chain order does this. `postgain` also runs from `0` and defaults to `1`, but it lands after the effects, multiplied with the envelope. It is the clean final trim: change the voice's level without changing how hard it drove anything upstream:

```
saw snd 2 decay 0.5 postgain .
```

`velocity` sits at that same late point, multiplied in beside the envelope and `postgain`. It carries how hard the note was struck, so it scales the finished voice and never the drive going into the effects. In practice: reach for `gain` when you want the distortion and the filters to hear a hotter signal, for `postgain` when you want the patch quieter, and for `velocity` when you want the loudness to follow the playing.

Both `gain` and `postgain` take modulation, so an `lfo` can move a voice's level up and down or a `slide` can fade it in over a step or two (see *Source modulation* (4)). `velocity` does not: you set it once per note.

```
saw snd 2 decay 0.0 0.5 1 lfo postgain .
```

word	where in the path	sets
`gain`	before filters and distortion	input level, and how hard the voice drives them
`postgain`	after the effects, with the envelope	the final output trim
`velocity`	with the envelope and `postgain`	a level that tracks how hard the note is struck

3.5 Panning

Three parameters can help to place a voice in the stereo field, from a plain left-right position to a wide stereo image. All three can take modulation, so a position can be a moving thing rather than a fixed one.

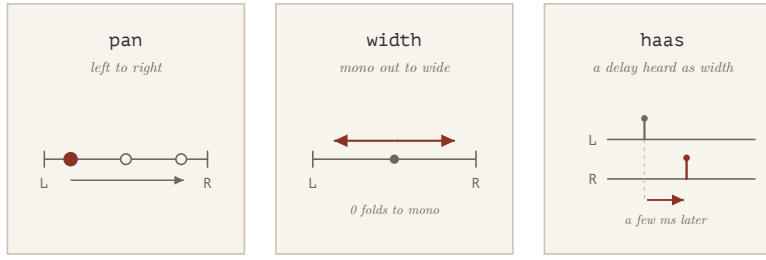


Figure 67: The three placement controls: pan sets left-right position, width scales the stereo image, haas widens with a short inter-channel delay.

3.5.1 Pan

`pan` sets the left-right position. The value goes from `0` (hard left) through `0.5` (centre) to `1` (hard right). It uses constant-power panning, so a sound holds the same loudness as it crosses the field instead of dipping in the middle:

```
hat snd 0 pan .
```

```
hat snd 1 pan .
```

Because `pan` can be modulated, an `lfo` turns it into an autopanner that drifts the voice from side to side:

```
saw snd c3 note tri 2 stepsecs osc 0 1 range pan .
```

3.5.2 Width

`width` works on the stereo image a voice already has, using mid-side processing. At `0` it folds the voice down to mono. At `1` it leaves it alone, and above `1` it pushes the sides wider. It has the most to do when there is real stereo content to act on, so it pairs naturally with `haas` or a unison `spread`:

```
saw snd c3 eb4 g4 note 5000 500 1 slide lpf 8 haas
1.6 width 0.5 gain .
```

3.5.3 Haas

`haas` delays one channel by a few milliseconds. The ear reads that tiny offset as width and placement rather than as an echo, a known psychoacoustic effect. Small amounts, `1` to `10` ms, can widen a mono voice without colouring it noticeably. Push past `10` ms and it starts to sound like a distinct slapback:

```
saw snd 5000 500 1 slide lpf c3 note 4 haas .
```

```
saw snd 5000 500 1 slide lpf c3 note 12 haas .
```

WORD	RANGE	SETS
`pan`	0 to 1	left-right position, constant power
`width`	0 and up	stereo width, 1 leaves it unchanged, 0 folds to mono
`haas`	ms	inter-channel delay that widens a mono voice

All three keep a voice inside the ordinary stereo pair. To move a sound around more than two speakers, on a quad or octophonic rig, the engine has a separate set of tools (see *Multichannel output* (9.1)).

Chapter 4

Source modulation

4.1 Spread, detune & glide

Three controls tune and thicken a source before any of the more specific tools reach it. `spread` stacks detuned copies, `detune` offsets the pitch, `glide` slides from one pitch to the next.

4.1.1 Spread

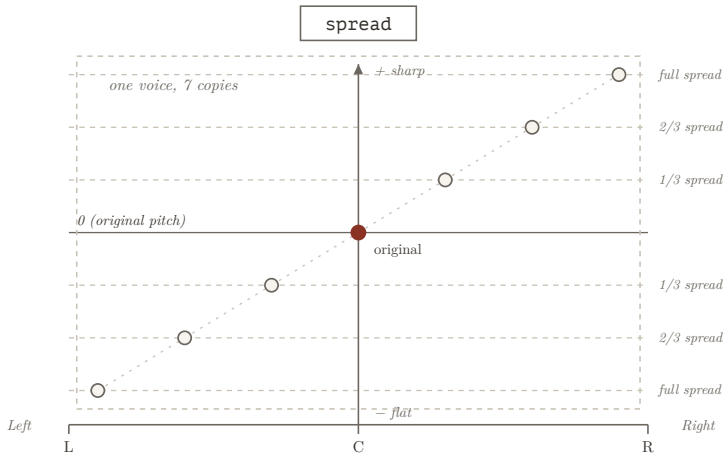


Figure 68: spread fans one voice into seven copies: the original at centre, six detuned in symmetric pairs that pan wider as they detune further.

`spread` turns one oscillator into a stack of detuned copies, the supersaw. At `0` it is off and you hear one voice. Above `0`, measured in cents, the engine adds six more copies around the original pitch and pans them across the stereo field, so the tone thickens and widens at once. A little is a gentle chorusing. A lot is the broad detuned wall of a trance lead:

```
saw snd c3 note 20 spread .
```

`spread` takes no motion. You set it for the voice and it holds.

4.1.2 Detune

`detune` shifts the pitch in cents, where `100` cents is a semitone. A plain number gives a static offset, for nudging one layer out of tune against another or stacking two voices a few cents apart by hand. A motion moves it, which is one way to write your own vibrato (see *Motions* (3.18.2)):

```
saw snd c3 note 7 detune .
```

4.1.3 Glide

`glide` is portamento. The voice slides to a new pitch and does not jump to it. The time is in steps, and the setting is sticky: once a voice has a glide, every later change of `note` or `freq` slides until `0 glide` disarms it. It only has work to do when the pitch changes, so it shows itself on a tagged voice you re-pitch, or on a monophonic line where one note follows another (see *Monophonic & held voices* (3.3)):

```
saw snd c3 note 0.5 glide 1 voice .
```

```
saw snd g3 note 1 voice .
```

The second emit re-pitches the voice tagged `1`, and because the glide is still set it slides from C up to G. Like `spread`, `glide` takes a plain number only.

WORD	RANGE	SETS
`spread`	0 to 100 cents	unison detune of seven stacked copies
`detune`	cents	a pitch offset, static or moving
`glide`	steps	portamento time for pitch changes

Pitch modulation (4.2) covers pitch that moves on its own. *Sub-oscillators* (4.4) covers a layer an octave down.

4.2 Pitch modulation

Pitch modulation moves a voice's pitch while it sounds. Vibrato is the built-in tool, a small oscillation around the note. For slower or wider movement, drive the pitch from a motion directly.

4.2.1 Vibrato

`vib` sets the vibrato rate in Hz. It is `0` by default, which is off, so vibrato starts only once you give it a rate. `vibmod` sets the depth in semitones, how far the pitch swings either side of the note, and defaults to half a semitone:

```
saw snd c4 note 6 vib .
```

```
saw snd c4 note 6 vib 1 vibmod .
```

The first adds a six-cycle-per-second wobble at the default depth. The second widens the swing to a full semitone each way.

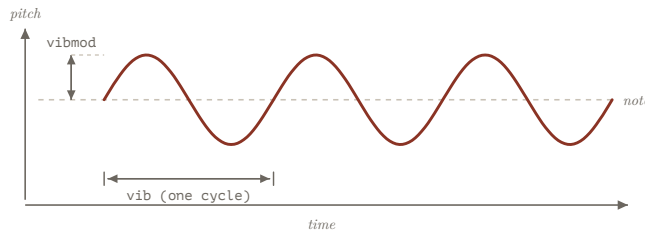


Figure 69: Vibrato oscillates the pitch around the note; vib sets the rate, vibmod sets the depth.

`vibshape` names the shape as a bareword. It defaults to a sine, the smooth back-and-forth of ordinary vibrato. Name `tri`, `saw`, `square`, or `sh` for sample-and-hold. A square jumps between two pitches and turns the vibrato into a trill:

```
saw snd c4 note 8 vib 2 vibmod [ tri saw square sh ]
choose vibshape .
```

`vib` and `vibmod` both take a motion, so the wobble itself can speed up or deepen across a note. `vibshape` names a shape and takes a plain word.

4.2.2 Driving the pitch directly

Vibrato is a fixed oscillation. Pitch is a parameter like any other, so for movement that is slower, wider or differently shaped, set `note` or `freq` from a motion. A `slide` glides between two pitches over a span of steps:

```
saw snd 220 880 4 slide freq .
```

An `lfo` on `note` sweeps the pitch up and down on a period you choose, here over eight steps:

```
saw snd 48 60 8 lfo note .
```

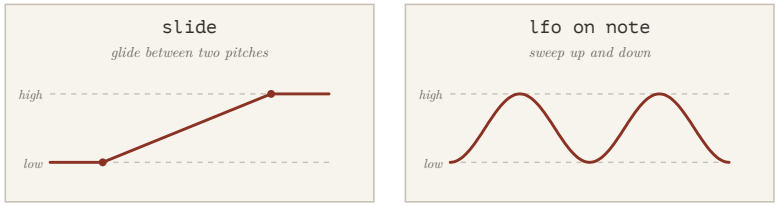


Figure 70: Driving pitch from a generator: a slide glides between two pitches, while an lfo on note sweeps it up and down.

WORD	RANGE	SETS
`vib`	0 to 100 Hz	vibrato rate
`vibmod`	0 to 12 semitones	vibrato depth
`vibshape`	sine, tri, saw, square, sh	vibrato oscillation shape

Spread, detune & glide (4.1) covers sliding between named notes. *Hard-sync and soft-sync* (4.5) covers pitch swept by a synced oscillator.

4.3 Amplitude & Ring Modulation

Amplitude modulation and ring modulation both multiply the voice by a second oscillator running at audio rate. What that multiplier does at its lowest point is the whole difference between them. Both are off by default and start when you give them a frequency.

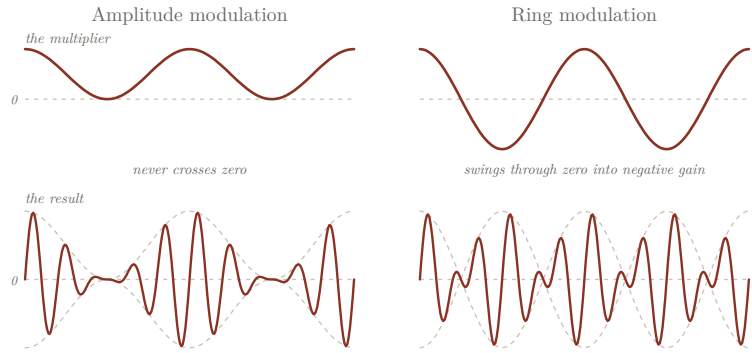


Figure 71: The same modulator against zero: amplitude modulation stays between zero and one, ring modulation crosses into negative and flips the voice's polarity.

4.3.1 Amplitude modulation

`am` sets the modulating oscillator's frequency in Hz. `amdepth` sets how deeply it acts, from `0` to `1`, defaulting to `0.5`. The modulator never drives the signal below zero gain. It scales the voice between full and reduced. At a low frequency this is tremolo, a regular swell in volume. Push the frequency into the audible range and the swell goes too fast to hear as rhythm. It breaks into sidebands, a pair of new tones above and below the original:

```
saw snd
c3 note
6 am
.
```

```
saw snd
c3 note
200 am
1 amdepth
.
```

The first is a six-per-second tremolo. The second, at two hundred cycles, adds metallic sidebands. `amshape` sets the modulator's shape, a sine by default, with `tri`, `saw`, `square` and `sh` available as for vibrato.

4.3.2 Ring modulation

`rm` is the same idea pushed further. Its depth, `rmdepth`, defaults to `1`, and at full depth the modulator swings the voice through zero and out the other side into negative gain. The carrier cancels out, leaving only the sum and difference tones:

```
saw snd c3 note 300 600 1 slide rm .
```

Lower `rmdepth` blends back toward the dry voice, so you can dial the effect in:

```
saw snd c3 note 600 2000 2 slide rm 0.5 1 8 slide
rmdepth .
```

`rmshape` sets the modulator's shape, like `amshape`. Both frequencies and both depths take motions. The two shape words take a plain name.

WORD	RANGE	SETS
`am`	0 to 20000 Hz	amplitude-modulator frequency
`amdepth`	0 to 1	amplitude-modulation depth
`amshape`	sine, tri, saw, square, sh	amplitude-modulator shape
`rm`	0 to 20000 Hz	ring-modulator frequency
`rmdepth`	0 to 1	ring-modulation depth
`rmshape`	sine, tri, saw, square, sh	ring-modulator shape

Frequency Modulation (FM) (4.6) multiplies into the pitch instead of the amplitude. *Gain and amplitude* (3.4) covers breathing the level by hand at the slow end of this.

4.4 Sub-oscillators

A sub-oscillator is a second oscillator pitched an octave or more below the main one, mixed in underneath to add weight at the bottom. It tracks the main pitch and stays in tune as you play, which is the usual way to give a bass or a lead more body without a filter or a second voice. `sub` sets its level, from `0` (off) to `1`. Turn it up and a lower octave fills in beneath the tone:

```
saw snd c3 note 0.7 sub .
```

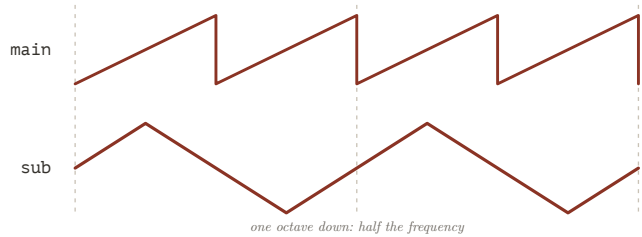


Figure 72: The sub-oscillator runs an octave below the main tone, one sub cycle spanning two of the main, mixed in underneath to add weight.

`suboct` sets how far below the main pitch the sub sits, in octaves. `1` is one octave down and the default, `2` is two, `3` is three. The lower it goes, the deeper the sub and the further it separates from the main tone:

```
saw snd c3 note 0.7 sub
< 1 2 3 > suboct
.
```

`subwave` names the sub's own waveform as a bareword, independent of the main source. It defaults to a triangle, round and clean under most tones. Name `sine` for the purest low end or `square` for a harder sub that cuts through:

```
saw snd
c3 note
0.7 sub
< sine square > subwave
.
```

Only the level takes a motion. `suboct` and `subwave` choose the sub once and hold it for the voice.

word	range	sets
`sub`	0 to 1	sub-oscillator level
`suboct`	1 to 3	octaves below the main pitch
`subwave`	tri, sine, square	sub-oscillator waveform

Spread, detune & glide (4.1) thickens at the same pitch. *Periodic oscillators* (2.1) covers the main oscillator's own shape.

4.5 Hard-sync and soft-sync

Oscillator sync runs a second oscillator and forces it to restart in step with the main one. The synced oscillator is tuned higher but keeps getting reset before it finishes its own cycle, so its waveform develops a hard discontinuity. That is the bright, vocal tone sync is known for. `sync` sets the ratio of the synced oscillator to the base. It defaults to `1`, which is off, because the two then run together. Raise it and the synced oscillator runs faster, so its formant climbs higher above the fundamental:

```
saw snd c3 note 3 sync .
```

`sync` takes a motion, so a `slide` sweeps the ratio across a span of steps. The pitch stays put while the timbre climbs, which is the sound sync is played for:

```
saw snd c3 note 1 8 6 slide sync .
```

4.5.1 Hard and soft

`syncmode` names the reset as a bareword. `hard` is the default and the brighter of the two. It snaps the synced oscillator back to the

start the instant the master completes a cycle, and that sharp edge is the sound. `soft` resets more gently, keeping more of the synced oscillator's own pitch for a rounder tone. Compare these two:

```
saw snd c3 note 4 sync soft syncmode .
```

```
saw snd c3 note 4 sync hard syncmode .
```

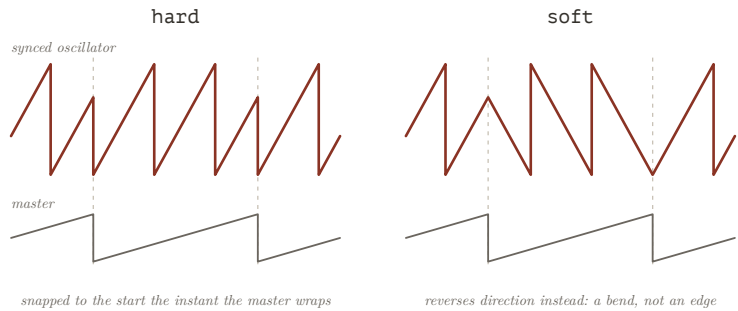


Figure 73: A saw synced to a slower master: hard sync snaps the wave back at each master cycle, soft sync reverses its direction instead.

`syncphase`, also `syncph`, sets the point in the cycle the hard reset snaps back to, from `0` to `1`. Moving it shifts where the discontinuity falls and recolours the formant:

```
saw snd
c3 note
3 sync
0.0 1.0 rand syncphase
.
```

Both the ratio and the phase point take motions. `syncmode` names one of two behaviours and takes a plain word.

WORD	RANGE	SETS
`sync`	1 and up, 1 is off	ratio of the synced oscillator to the base
`syncmode`	hard, soft	how the reset behaves
`syncphase` / `syncph`	0 to 1	hard-reset phase point

Spread, detune & glide (4.1) stacks detuned copies instead. *Phase distortion* (4.7) is the other bright-edge tool on a plain oscillator.

4.6 Frequency Modulation (FM)

Frequency modulation drives the pitch of the oscillator with another oscillator running at audio rate. Slowly, this would be vibrato. At audio rate the pitch moves too fast to hear as movement and sprays sidebands instead, which is how FM builds bright metallic tones from plain sine sources. The engine gives you two modulating operators and a feedback path.

`fm`, also `fmi`, is the modulation index: how hard the modulator drives the carrier, and so how rich the result. At `0` there is no FM and you hear the plain source. Raise it and sidebands pile up:

```
sine snd c3 note < 1 2 3 4 > fm .
```

`fmh` sets the modulator's frequency as a ratio to the carrier, defaulting to `1`. Whole-number ratios stay harmonic and still read as a note. Ratios off the integers turn inharmonic and bell-like. This is the strongest control over the colour of an FM patch:

```
sine snd c3 note 6 fm 2 fmh .
```

```
sine snd c3 note 6 fm 3.5 fmh .
```

The first stays harmonic at a ratio of two. The second, at three and a half, turns bell-like. `fm` takes a motion, so sweeping the index opens the timbre over a note:

```
sine snd c3 note 0 8 4 slide fm 2 fmh .
```

`fmshape` sets the modulator's waveform, a sine by default.

4.6.1 The second operator

A second modulator adds another layer of sidebands. `fm2` sets its depth and `fm2h` its ratio, the same as the first pair.

`fmpivot` wires the two operators together. At `0` they cascade, the second feeding the first. At `0.25` they run in parallel, both driving the carrier side by side. Values between blend the two routings,

and it wraps. In pure cascade operator two reaches your ears only through operator one, so `fm` has to be above zero for it to sound at all. In parallel it needs nothing from `fm`.

`fmfb` adds feedback on the topmost operator, from `0` to `1`, roughening its output toward noise as it climbs. `fmloop` closes the pair into a ring. Operator two already feeds operator one through the cascade side of `fmpivot`, and `fmloop` returns operator one back into operator two. It needs `fm2` above zero, because one operator alone has nothing to loop through. Low values thicken and brighten the tone. Push it and the loop growls and tips over into pitched chaos:

```
sine snd c2 note 6 fm 1 fmh 4 fm2 3 fm2h 0 1 8 slide
fmloop .
```

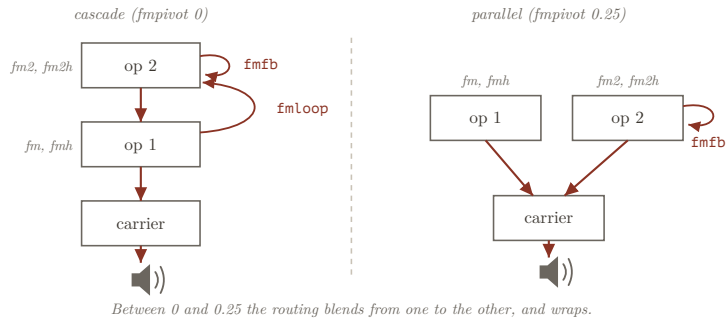


Figure 74: The three routing controls: fmpivot blends between cascade and parallel, fmfb feeds the top operator back into itself, fmloop rings op 1 into op 2.

```
sine snd
c3 note
1 8 rand fm
[ 0.5 1 2 ] choose fmh
3 fm2
5 fm2h
.
```

Every one of these takes a motion except `fmshape`, which names a waveform.

word	range	sets
`fm` / `fmi`	0 to 100	modulation index of operator one
`fmh`	0 to 32	frequency ratio of operator one
`fmshape`	sine, tri, saw, square, sh	operator one waveform
`fm2`	0 to 100	depth of operator two
`fm2h`	0 to 32	frequency ratio of operator two
`fmpivot`	0 to 1, wraps	how the two operators are routed
`fmfb`	0 to 1	feedback on the topmost operator
`fmloop`	0 to 1	return from operator one into operator two (needs `fm2`)

Pitch modulation (4.2) drives the pitch slowly instead of at audio rate. *Amplitude & Ring Modulation* (4.3) multiplies the amplitude.

4.7 Phase distortion

Phase distortion bends the steady ramp an oscillator reads its waveform through. The pitch holds, because the wave still repeats at the same rate, but the shape changes and the harmonics with it. This is the Casio CZ line of synthesis, a cheap cousin of FM. Three controls do the bending. *Periodic oscillators* (2.1) introduces them. Here they go further and into motion.

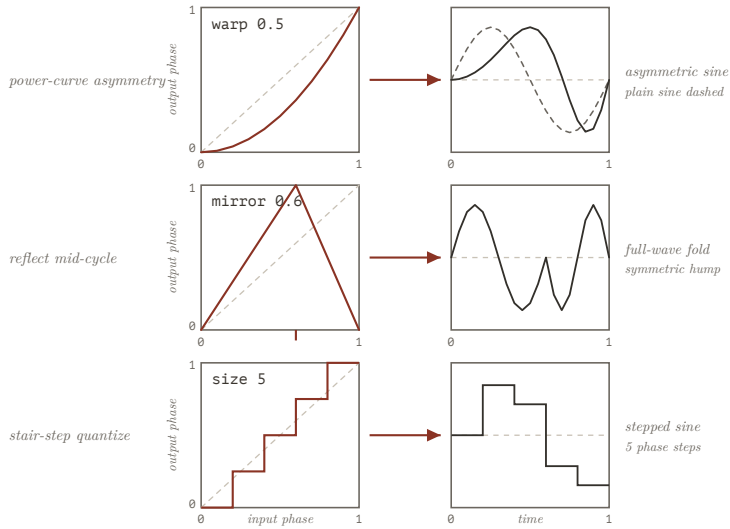


Figure 75: The phase ramp bent three ways by warp, mirror, and size, each bent ramp (left) beside the waveform it produces (right).

`mirror` reflects the phase at a point you choose, from `0` to `1`. At `0` it is off. Move it in and the waveform folds back on itself partway through each cycle, adding harmonics. It is the one of the three that takes a motion, so you can sweep the reflection across a held note:

```
sine snd c3 note 0.0 1.0 rand mirror .
```

```
sine snd c3 note 0 1 4 lfo mirror .
```

`warp` bends the phase with a power curve, from `-1` to `1`, compressing one half of the cycle and stretching the other. At `0` the wave is untouched. Toward either end the asymmetry hardens or softens the edge:

```
tri snd c3 note 0.5 warp .
```

`size` quantizes the phase into a number of steps, from `0` (smooth) up to `256`. Low values stair-step the wave into the coarse aliased shapes of early digital hardware, the chiptune end of the range:

```
saw snd c3 note 8 size .
```

`warp` and `size` take plain numbers. Set them for the voice and they hold.

word	range	sets
`mirror`	0 to 1	phase reflection point, takes modulation
`warp`	−1 to 1	phase asymmetry
`size`	0 to 256	phase quantization steps

Periodic oscillators (2.1) has these controls in their plainest form. *Amplitude & Ring Modulation* (4.3) adds harmonics by multiplying the signal instead of bending its phase.

Chapter 5

Sample mangling

5.1 Selecting samples

A bank is a folder of samples that share one name. Cagire numbers the files inside it from `0`, in alphabetical order, and `n` is how you choose between them. Without `n` you get the first.

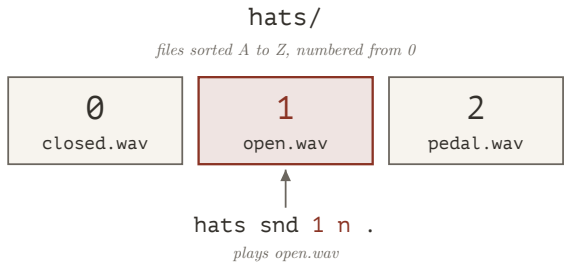


Figure 76: Cagire numbers the hats bank from 0; 1 n selects open.wav, and n wraps an index past the end back to the start.

```
hats snd .
```

That plays file `0` of the `hats` bank. Set `n` to reach the others:

```
hats snd 1 n .
```

`1 n` selects the second file, value first, word after, postfix like every other parameter (see *The sound register* (2.5)). `n` wraps. If a bank holds three files, an `n` of `3` comes back to `0`, `4` to `1`, and so on, so you can drive it from a counter or a pattern without first checking how many files the bank holds:

```
hats snd 7 n .
```

In a bank of three this plays file `1`, since `7` wraps round to `1`.

See *Loading samples* (5.1) for how a folder of files becomes a bank in the first place.

5.2 Playback speed

A sample has two parameters you can change: how fast it plays and what pitch it sounds at. By default they move together. `speed` is the plain control over both at once, and `stretch` is the one that pulls them apart. `speed` multiplies the playback rate. It defaults to `1`, the recorded speed. At `2` the sample plays twice as fast and an octave up; at `0.5` it plays half as fast and an octave down. Speed and pitch move together because that is what changing the playback rate of a recording does, the same as spinning a record faster:

```
break snd 0.5 speed .
```

Pitching a sample with `note` or `freq` is the same mechanism seen from the other side. The sample's home pitch is middle C, so naming a note sets the speed needed to land there (see *Audio samples* (2.7)). Setting `speed` directly is the way in when you are thinking in playback rate rather than in pitch.

5.2.1 Playing backwards

A negative `speed` plays the sample in reverse, starting from the end and running to the start. There is no separate reverse word. The sign of the speed is the whole of it:

```
break snd -1 speed .
```

5.2.2 Stretching without pitching

`stretch` changes the duration without touching the pitch. It defaults to `1`, and a `stretch` of `2` makes the sample take twice as long at its original pitch, while `0.5` makes it half as long. This is time-stretching done with a phase vocoder, so it holds the pitch steady while the clock changes underneath it:

```
break snd 2 stretch .
```

Set `stretch` to `0` and the clock stops. The sample holds on a single point and sustains it for as long as the voice lasts, a freeze:

```
break snd 0 stretch .
```

With `stretch` in play, `speed`, `note`, and `freq` still set the pitch, but now independently of the duration. That is the difference between the two controls: `speed` ties pitch and time together, `stretch` lets you hold one while moving the other. The phase vocoder is only one way of getting there; setting `grain` points the same `stretch` at a grain cloud instead, which trades the smooth reading for an audibly fragmented one (see *Granulation* (5.4)).

5.2.3 Modulating speed

`speed` takes modulation like most parameters, and because it sets the playback rate, modulating it bends pitch and time at once. A slow `lfo` on `speed` is a tape wow, the pitch sagging and lifting under the sample:

```
break snd 0.8 1.2 4 lfo speed .
```

Push the same oscillation up to audio rate, a period of a fraction of a step, and the wobble stops reading as movement and becomes a tone of its own. The speed is now changing faster than the ear can follow, so it sprays sidebands across the sample the way audio-rate pitch modulation does, a buzzing, metallic growl laid over whatever is playing (see *Frequency Modulation (FM)* (4.6)). It is easy to overdo and worth a light hand:

```
break snd 0.5 2 0.02 lfo speed .
```

WORD	RANGE	SETS
`speed`	−100 to 100	playback rate and pitch together, negative plays in reverse
`stretch`	0 to 100	duration independent of pitch, 0 freezes

Both of these set a rate and let the playhead run at it. To set the position itself, and to sweep it as a parameter, see *Scanning* (5.5).

For choosing which file in a bank plays, see *Selecting samples* (5.1); for playing only a portion of one, see *Looping samples* (5.3); for the granular reading of `stretch`, see *Granulation* (5.4).

5.3 Looping samples

Playing a whole sample from start to finish is the default. Often you want it to lock to the clock instead: a break that fills the bar however long the file is, the back half of a phrase, one drum out of a loop. The word that fits a sample to the grid is `loop`. Three more words sit around it. `begin` and `end` window the sample down to a portion. `slice` with `pick` chops it into equal parts and steps between them. `stut` retriggers one emit into copies that tile the gate.

5.3.1 Fitting to the grid

`loop` takes a number of steps and fits the sample to span exactly that many. A step is a sixteenth note, so a count of sixteen fills one bar. Whatever the file's native length, `loop` lands it on the grid by setting the playback speed. A count shorter than the file speeds it up and raises the pitch. A longer count slows it down and lowers it, the same as spinning a record at a different speed. The pitch moves with the speed because this is a change of playback rate, not the phase-vocoder stretch from the previous page:

```
break snd 16 loop .
```

That fits the break across one bar and holds the voice open for the whole span, since `loop` sets the speed and the gate together. It is the quickest way to make a sampled loop sit in time, without working out a `speed` factor by hand. Because `loop` takes a value like any parameter, an alternation changes the length as it plays, doubling and halving the break against the bar:

```
break snd < 16 8 > loop .
```

5.3.2 Windowing

`begin` and `end` mark where playback starts and stops, each a fraction of the whole sample from `0` to `1`. `begin` defaults to `0` and `end` to `1`, the whole file. Move them in and you play only the slice between:

```
break snd 0.5 begin .
```

That starts halfway through and runs to the end. Set both to take a window out of the middle:

```
break snd 0.25 begin 0.5 end .
```

If you set `begin` past `end` the two swap, so the order you give them never matters.

5.3.3 Slicing

For a break you want to chop into even pieces, `slice` divides the sample into a number of equal parts and `pick` chooses which one plays, counted from `0`. This is the usual way to take a drum loop apart and rebuild it. Here an eight-slice break plays its fourth slice:

```
break snd 8 slice 3 pick .
```

`pick` wraps, so a `pick` of `8` on an eight-slice sample comes back to `0`, and it takes negatives, counting in from the end. Because it takes a value like any parameter, an alternation steps through slices and rearranges the break as it plays:

```
break snd 8 slice < 0 4 2 6 > pick .
```

Feed `pick` the run counter and it walks the break one slice per pass. `runs` reports how many times the step has fired, and `pick` wraps whatever value it is given:

```
break snd 8 slice runs pick .
```

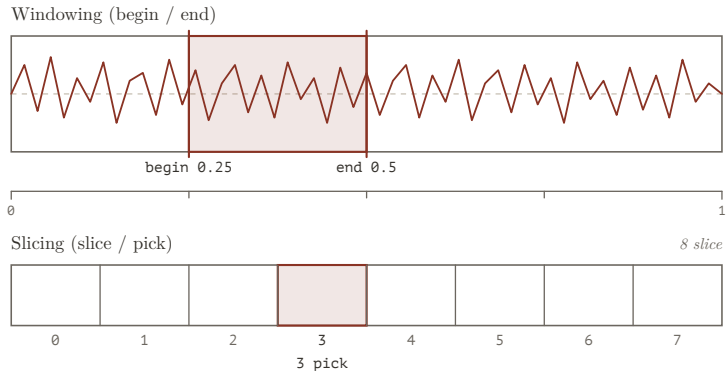


Figure 77: Two ways to address a sample: begin and end window an arbitrary fraction, while slice cuts it into equal numbered parts and pick selects which one plays.

`begin` and `end` take precedence over `slice`, so use one approach or the other on a given voice, not both at once.

5.3.4 Choosing between them

`loop` fits the whole file to a step count and is the fastest way onto the grid. It fits by changing the playback speed, so the pitch moves with the count. `slice` and `pick` fit when you already think of the sample as a row of equal hits and want to reorder them. `begin` and `end` fit when you want an arbitrary portion rather than an even division. For a fit that holds the pitch, or for fitting by ear rather than to a step count, set `stretch` or `speed` directly (see *Playback speed* (5.2)). A held soundfont note loops on its own while the note sounds, but that is built into the source rather than something you switch on (see *Soundfonts* (2.9)).

5.3.5 Stuttering

`stut` turns one emit into a burst of retriggers. Give it a count and the voice fires that many times in a row, contiguous across the note. The count divides the gate, so the retriggers tile the original span exactly. Four `stut` on a one-step gate lands four hits inside that step, each a quarter as long. The copies fill the span edge to edge instead of leaving one note to ring:

```
"kick" snd 1 gate 4 stut .
```

It composes with `slice` and `pick`. A stuttered slice is the classic ratchet fill: pull one hit out of a break and repeat it fast across the step.

```
break snd 8 slice 3 pick 4 stut .
```

`stut` wraps the count into the gate rather than the sample, so it is not sample-specific. A stuttered kick, a stuttered sine, and a stuttered soundfont note all ratchet the same way.

WORD	RANGE	SETS
`loop`	step count	fit the sample to span N steps by changing playback speed, pitch moves with it
`begin`	0 to 1	where playback starts
`end`	0 to 1	where playback stops
`slice`	count	divide the sample into N equal slices
`pick`	slice index, wraps	which slice plays, negatives count from the end
`stut`	count	retrigger the emit N times, tiling the gate span

Every word on this page fences off a region and leaves the playhead to cross it on its own. `scan` is the other half. It measures across whatever region you have set here and places the playhead inside it (see *Scanning* (5.5)).

For the speed control `loop` leans on, and the pitch-held `stretch` beside it, see *Playback speed* (5.2); for choosing the file in the first place, see *Selecting samples* (5.1).

5.4 Granulation

Cagire has two ways of moving a sample through time, and `grain` is the switch between them. Left alone, `stretch` runs a phase vocoder. It holds the pitch steady while the clock changes underneath it, and it works hard to sound like the original recording taken slower (see *Playback speed* (5.2)). Granulation does the opposite.

Set `grain` and Cagire stops reading the sample as one continuous stream. It cuts the sample into short fragments that fade in, fade out, and overlap their neighbours. You hear the grains, and that is the point.

`grain` is the switch and the size at once. It sets a grain length in milliseconds and defaults to `0`, meaning off, so a sample plays as it always has until you ask for otherwise. Any positive value hands the sample to the granular reader:

```
break snd 60 grain .
```

Sixty milliseconds is long enough that every grain still carries a recognisable piece of the source. Take it down to a few milliseconds and the grains stop being fragments of a recording. They become a pitch of their own, because the launch rate itself turns into a tone over whatever is underneath:

```
break snd 3 grain .
```

5.4.1 Where the grains come from

A scan head walks through the sample and grains launch from wherever it currently sits. `stretch` sets how fast that head travels, keeping the meaning it has everywhere else: `1` is the recorded pace, `4` takes four times as long, `0` stops the head entirely. Only the method differs, granular instead of phase-vocoded. The same word covers both, and `grain` decides which one you hear:

```
break snd 4 stretch 40 grain .
```

Set `stretch` to `0` and the head parks. Grains keep launching from that one spot for as long as the voice lasts, which turns any fraction of a second of source material into a sustained drone:

```
break snd 0 stretch 120 grain 4 gate .
```

Where the head parks is wherever playback had reached, since `stretch` only ever sets its rate. To place the head yourself, and to sweep it independently of the grain pitch, see *Scanning* (5.5).

`speed`, `note`, and `freq` still set the pitch, and here they pitch the grains rather than the head. That is the decoupling granulation

gives you. The head walks at whatever rate `stretch` asks for. The grains sound at whatever pitch you name. Neither one drags the other along.

```
break snd 0.25 stretch 50 grain 12 note .
```

5.4.2 Spraying the cloud

With `spray` at its default of `0` the grains follow the head in order and all arrive dead centre, which is tidy and close to ordinary playback. Raise it and each grain lands somewhere random around the head, measured as a fraction of the playback region. Each grain also lands somewhere random across the stereo field. Both scatters move together, because `spray` is one control over how much of a cloud this is. A little of it blurs the edge off the source. At `1` the grains land anywhere in the region and anywhere in the image:

```
break snd 0 stretch 80 grain 0.15 spray 4 gate .
```

`dens` is how many grains overlap at any moment, from `1` to `8`, defaulting to `2`. Two is the orderly tiling, one grain fading in as the last fades out. Higher counts thicken the texture into something closer to a wash. Combine that with `spray` and you have a true grain cloud:

```
break snd 8 stretch 25 grain 0.4 spray 8 dens .
```

At `dens 1` the grains stop overlapping altogether. They still run back to back, but nothing sits underneath them. Each window falls to silence at both ends, so the level notches to nothing at every boundary. That reads as a stuttering, gated version of the sample rather than a smooth one.

5.4.3 What it composes with

`begin` and `end` bound the region the head walks and the grains come from. Window the sample down to a phrase and granulate just that phrase (see *Looping samples* (5.3)). A cloud ends when the head reaches the end of its region. If `stretch` has frozen the head, the gate decides instead.

`pan` still positions the whole cloud, working as a balance on top of wherever the individual grains landed, so a sprayed cloud panned right leans right without collapsing. `spread` does nothing here. It is a unison control for the oscillators and never reaches a sample.

All three words take modulation. Sweeping `grain` moves the texture from tonal to fragmentary while it plays, and modulating `spray` opens and closes the cloud around the head:

```
break snd 2 stretch 5 80 8 lfo grain 0.3 spray .
```

Granulation does not carry one thing over. A fractional `n` normally crossfades between two neighbouring files. Here Cagire ignores the fraction and granulates the first file.

WORD	RANGE	SETS
`grain`	0 to 1000	grain size in ms, 0 plays the sample normally
`spray`	0 to 1	how far grains scatter from the scan head, in position and in stereo placement
`dens`	1 to 8	grains overlapping at once, 2 is an even tiling

For the pitch-held `stretch` on its own and for the plain playback rate, see *Playback speed* (5.2); for windowing the region first, see *Looping samples* (5.3); for placing the scan head rather than walking it, see *Scanning* (5.5).

5.5 Scanning

Every other sample word sets how the playhead moves. `speed` gives it a rate. `stretch` gives it a rate that leaves the pitch alone. `loop` picks the rate that lands the file on the grid. `begin` and `end` fence off the stretch of sample it is allowed to cross. None of them say where the playhead actually is. `scan` says it, and it says it continuously. The read position stops being something Cagire works out and becomes something you play.

`scan` is a fraction from `0` to `1` across whatever region is currently in force. `0` is the start of that region and `1` is the end. It is a position, not a rate. Leave it out and every reader keeps the motion it would

have had, which is why nothing you already have changes until you reach for it.

5.5.1 Scratching a sample

Give a plain sample a moving `scan` and you are scratching it. The playhead goes wherever you point it. The rate is how fast you move it, and the pitch follows that rate the way it does when you push a record by hand. One pass across the whole file over four steps:

```
break snd 0 1 4 slide scan 4 gate .
```

An `lfo` in place of the `slide` runs the same gesture back and forth instead of once, which is closer to what a hand actually does:

```
break snd 0 1 2 lfo scan 4 gate .
```

`speed` stops applying here, and it has to. A playhead cannot both be placed and advance on its own. `note` and `freq` go with it, because they are the same control seen from the pitch side. Whatever pitch you hear is the one your movement implies.

A record held still makes no sound, and neither does this. A `scan` that does not move reads the same frame over and over, which is silence. A static `scan` becomes useful only when a reader underneath it makes motion of its own. The next two sections cover those readers.

5.5.2 Scanning a grain cloud

Set `grain` and position stops dragging pitch along with it. Granulation already walks a scan head through the sample, and `scan` places that head (see *Granulation* (5.4)). The grains launch from it and sound at whatever pitch you name, so position and pitch come apart completely. Park the head halfway in and the cloud drones on that spot for as long as the gate holds:

```
break snd 60 grain 0.5 scan 4 gate .
```

That is the drone `0 stretch` already gives you, except you chose the spot rather than taking wherever playback had reached. Since the pitch is free, you can play it:

```
break snd 60 grain 0.5 scan 12 note 4 gate .
```

Sweep it instead and the cloud crosses the sample at a rate you set, at a pitch that stays put:

```
break snd 40 grain 0 1 8 slide scan 8 gate .
```

An alternation moves the cloud a step at a time, which is the granular answer to `slice` and `pick`:

```
break snd 60 grain < 0 0.25 0.5 0.75 > scan .
```

5.5.3 Scanning the phase vocoder

`stretch` runs the phase vocoder, and `scan` places its analysis window the same way it places the grain head. Pitch stays where you put it and the reading stays smooth rather than fragmented, which is the trade these two readers have always made (see *Playback speed* (5.2)). Sweeping down plays the sample backwards without touching the pitch, which a negative `speed` cannot do:

```
break snd 2 stretch 1 0 8 slide scan 8 gate .
```

`stretch` is still what picks the reader, so it needs a value other than `1` for the vocoder to run at all. With `scan` driving, that value no longer sets a rate. It is only the switch. `grain` does the same double duty, where the size you give it is also the thing that turns it on.

The vocoder re-reads its window once per analysis frame rather than once per sample, so a fast `scan` arrives here in steps rather than continuously. At sweep rates that is inaudible. Pushed to audio rate it becomes part of the sound, and the grain cloud is the better instrument for that.

5.5.4 What it composes with

`scan` is measured across the current region rather than the whole file, so windowing first and scanning second does what you would expect. This sweep covers only the quarter of the sample between `begin` and `end`:

```
break snd 0.25 begin 0.5 end 0 1 4 slide scan 4
gate .
```

`slice` and `pick` narrow the region the same way, so `scan` sweeps inside the chosen slice rather than across the file (see *Looping samples* (5.3)).

A scanned voice never ends by running out of sample, because a driven playhead is free to leave the region and come back. The gate decides instead, the same rule a frozen cloud follows.

5.5.5 Scanning a held voice

`scan` is an ordinary parameter, so a held voice takes a new position from any later emit carrying the same `voice` tag. Open a cloud with a `0` gate and move it by hand as often as you like:

```
break snd 60 grain 0 gate 0 voice .
0.6 scan 0 voice .
```

That is a step-rate move rather than a continuous one, so it reads as jumping the head rather than sweeping it (see *The sound register* (2.5)).

WORD	RANGE	SETS
`scan`	0 to 1	the read position across the current region, driving the playhead instead of letting it run

For the rates `scan` replaces, see *Playback speed* (5.2); for the region it is measured across, see *Looping samples* (5.3); for the cloud it places, see *Granulation* (5.4).

5.6 Live recording

Cagire can record its own output, or whatever is coming in live, straight into a sample bank, and then play it back like any other sample. This is how you capture a phrase you have just built and fold it back into the music, or sample an incoming instrument and chop it. Recording is driven by paired words, one to start and one to stop, and the result lands under a name you choose. `rec` starts recording the engine's master output to a named sample. You push

the name and call `rec`; everything sounding from that moment is captured. `endrec` stops it and registers the result as a bank. This will start the recording:

```
"phrase" rec
```

And this will end it:

```
endrec
```

Between those two the recording runs. Once `endrec` has registered it, `phrase` is a bank like any other, played with `snd`, indexed with `n`, windowed and pitched with everything on the other sample pages:

```
"phrase" snd .
```

Overdub onto a take with `dub` to build a loop up in passes, the way a looper pedal stacks layers, and `orec` captures a single orbit instead of the whole mix. The full set of recording words, with their signatures, is in *Recording* (3.19.3).

Recording pairs naturally with live input. Turn audio input on, bring it in with `live`, and record the result to sample an incoming instrument (see *Live audio* (2.8)). If that input is a microphone, wear headphones; monitored through speakers, the mic hears itself and the loop builds into a howl.

For bringing an external source into the engine to record, see *Live audio* (2.8); for the full set of recording words, see *Recording* (3.19.3); for playing the result back, see *Selecting samples* (5.1).

Chapter 6

Envelopes

6.1 Voice envelope

Every voice has an amplitude envelope, the contour that carries it from silence to full and back. Even a bare `sine snd .` has one, because Cagire supplies a synth `gate` and `release` when you set none. Native drums and GM instruments keep the envelopes designed for their source instead.

The full shape runs six stages, which makes it a DAHDSR. A delay before anything sounds, an attack up to full, a hold at the top. Then a decay down to the sustain level, the sustain itself while the note is held, and a release back to silence once the note ends. The stages are curves, not lines: attack, decay, and release each start fast and end slow.

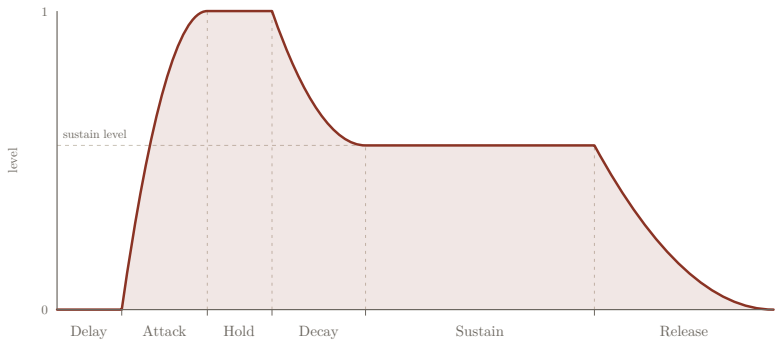


Figure 78: The six stages of the DAHDSR envelope: delay, attack, hold, decay, sustain, and release. Only attack, decay, and release are curved.

Stage	Word	Sets
Delay	`envdelay` (`envdly`)	silence held before the attack

Stage	Word	Sets
Attack	`attack` (`att`, `a`)	time to rise from silence to full
Hold	`hold` (`hld`)	time held at full before the decay
Decay	`decay` (`dec`, `d`)	time to fall to the sustain level
Sustain	`sustain` (`sus`, `s`)	the level held while the note sounds
Release	`release` (`rel`, `r`)	time to fall to silence after the note ends

You set each stage with its own word, in any order:

```
saw snd 0.01 attack 0.3 release .
```

A fast attack and a gentle release, a plucked shape. The times are in musical time, the same unit as `gate`, so they stretch with the tempo and are never pinned to seconds. A `decay` of `1` lasts one step, a `release` of `4` lasts four. `sustain` is the exception. It is a level from `0` to `1`, the height the voice holds at after the decay:

```
sine snd 0.2 decay 0.5 sustain .
```

Here the voice falls to half level over its decay and stays there until it is released.

6.1.1 What you get for free

Set nothing and the voice still has a shape. A bare `saw snd .` gives you a 3 ms attack, no delay, no hold, no decay, full sustain, a `gate` of four steps and a `release` of one step. A sample keeps its longer defaults, a whole pattern loop for `gate` and twelve steps for `release`. A GM instrument gets the four-step gate but keeps the complete envelope from its SoundFont, while a native drum keeps its kit envelope and engine gate. This is why each source has a useful natural shape before you touch its envelope. Name `gate` or an envelope stage and your value wins.

6.1.2 Gate and length

`gate` sets how long the note is held, the moment the release begins, in musical time like the rest of the envelope. At `0` the release never comes and the note holds until you end it yourself, the infinite voice of *What is a doux voice?* (3.1):

```
saw snd c2 note 0 gate 0 voice .
```

```
;; run me to stop the sound
saw snd c2 note 1 gate 0 voice .
```

With no `gate` set, what happens depends on what you are playing. A synth or GM instrument is held for four steps. A sample gets a whole pattern loop, because a sample already stops itself when the file runs out. Its `gate` is only a ceiling: it decides when a long sample gets cut off, never how long a short one sounds. A native drum uses the engine gate and normally fades through its kit decay long before that gate closes. Set `gate` yourself and your number wins in every case.

6.1.3 The short forms

Setting four stages one word at a time is wordy, so two shorthands take their values straight from the stack. `adsr` sets attack, decay, sustain and release in that order:

```
saw snd 1 1 0.5 2.3 adsr 8 gate .
```

`ad` sets just attack and decay and pins sustain to `0`, the percussive shape you want for most drums and stabs:

```
kick snd 0.001 0.2 ad .
```

A word on the aliases. `a`, `d`, `s` and `r` are short for attack, decay, sustain and release. They are handy when you type fast and a trap when you name things. Those single letters belong to the envelope, so never use them as names of your own. A stray `s` sets the sustain.

6.1.4 The decay in two moves

A `decay` can split into two phases: a fast body and a slower tail. `body` sets how much of the decay the body takes, a fraction from `0` to `1`. At `0` the decay is one curve, the way it has always been. At `0.2`, the first fifth of the decay falls hard to a low level, then the rest fades the tail out. `body` is a fraction, not a time, so it stretches with the tempo exactly as `decay` does:

```
kick snd 0.25 body .
```

The drums set their own `body` by default, so a kick keeps a short punch and a long boom without you touching anything. Name a drum and yours replaces the default, the same way `decay` does.

6.1.5 The envelope does not move

This envelope shapes the voice's amplitude and nothing else. Its stages take no motion. `envdelay`, `attack`, `hold`, `decay`, `body`, `sustain` and `release` want plain numbers and refuse a curve.

To open and close a filter with the note, build a second envelope as a motion and hand that to the parameter. `ead`, `eadr` and `eadsr` build one, `env` being another name for the last (see *Shape words* (3.18.1)). Those run four stages, not six. A modulation envelope has no delay and no hold.

Chapter 7

Insert effects

7.1 What are insert effects?

An insert effect lives inside one voice's own chain and processes only that voice. It is wired in series with the source: the filter, the distortion, the chorus, and all the other effects each act on that single voice and nothing else. That is the contrast with an orbit effect. A reverb or a delay sits on a shared send bus and several voices feed one space at once (see *What are orbits?* (8.1)). The filters, the distortion, the movement effects, the chorus, and the pan are all insert effects; they sit in the voice's own chain:

```
tri snd 1500 lpf .
```

That lowpass belongs to that one triangle voice. Fire another voice and it gets its own filter, set however you like.

7.1.1 Inside the voice

Whatever the source, a voice runs the same fixed path from raw tone to finished sound: source → `gain` → the filters and `wah` → the lo-fi and distortion group → `vinyl` → `am` · `rm` → `phaser` · `flanger` → `fshift` · `pshift` → the EQ and `tilt` → `smear` → an `fx` patch, if the voice carries one → envelope · `velocity` · `postgain` → `chorus` → `width` · `haas` → `pan`. The insert effects are every stage after the source. The path is fixed no matter how you type the words. Writing a filter before or after a distortion changes nothing about which one runs first. The path decides that, and the rest of this section follows it. Several of these stages take modulation. Every period and duration in that modulation is counted in steps, where a step is a 16th note. A period of `4` means four steps, not four seconds or four beats. Here a lowpass on pink noise wanders on a four-step period:

```
pink snd 200 4000 4 drunk lpf .
```

The per-effect pages all rely on this; when one of them says a period of `8`, it means eight steps.

7.1.2 Why order matters

Because `gain` sits before the filters and the distortion, it sets how hard the voice drives them. Because `postgain` sits after the effects, it trims the final level without touching that drive. So the same patch can be made quieter in two very different ways, and only one of them changes how the distortion bites:

```
saw snd
0.9 gain
1200 lpf
0.4 distort
0.6 postgain
.
```

Turn `gain` down and the `distort` stage sees a softer signal and bites less; turn `postgain` down instead and the drive is untouched, only the output is quieter. When an effect does not do what you expect, the order is usually the reason.

7.1.3 Insert versus orbit

An insert effect is per-voice and private. An orbit effect is shared, because some effects sound more natural when several voices feed one space than when each carries its own, so those live on a send bus rather than in the voice's chain (see *What are orbits?* (8.1)). Everything in this section stays inside the voice, on its own.

7.2 Filters

7.2.1 Lowpass / bandpass / highpass

Filters remove or emphasise a band of frequencies. Cagire gives you three filter shapes at three slopes, plus an envelope-driven auto-wah. A lowpass keeps the lows and drops the highs, a highpass does the opposite, and a bandpass keeps a slice in the middle and

discards the rest. The plainest one is a single word and a cutoff in Hz.

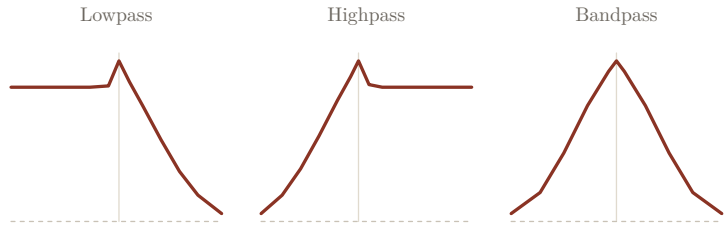


Figure 79: Frequency response of the three filter shapes, with the resonance peak sitting right at the cutoff.

Run this example a few times:

```
saw snd 500 4000 rand lpf 1 decay .
```

The three filters

Each shape comes as a cutoff word and a resonance word. `lpf` sets the lowpass cutoff in Hz and `lpq` its resonance from `0` to `1`. `hpf` and `hpq` are the highpass. `bpf` and `bpq` are the bandpass, where `bpf` sets the centre of the band. Resonance lifts a peak right at the cutoff, and the higher you push it the more the filter rings. Moving the filter cutoff value while the note plays is an all-time classic. Slide the lowpass open across a saw bassline and the note blooms from dark to bright over a few steps. A touch of `lpq` sharpens the edge it climbs:

```
saw snd
c2 note
200 2000 4 slide lpf
0.6 lpq
.
```

A highpass thins a sound from below. A sample-and-hold `jit` on the cutoff makes a hat skitter, picking a new corner frequency every couple of steps:

```
hat snd
500 4000 0.5 jit hpf
.
```

A bandpass picks one slice out of the spectrum, which is exactly what you want for carving pitched tone out of noise. Sweep the centre with a saw ramp and lift the resonance with `bpq` for a vocal-like effect:

```
white snd
saw 2 stepsecs osc 400 4000 range bpf
0.6 bpq
0.5 gain
.
```

word	sets
`lpf`	lowpass cutoff in Hz
`lpq`	lowpass resonance, 0 to 1
`hpf`	highpass cutoff in Hz
`hpq`	highpass resonance, 0 to 1
`bpf`	bandpass centre in Hz
`bpq`	bandpass resonance, 0 to 1

A lowpass also pairs naturally with the note's own shape. Set an amplitude envelope with `ad` first. `lpg` then ties that envelope to a cutoff sweep, so the voice opens and closes its filter as it swells and fades. The depth value sets how far the cutoff travels:

```
saw snd c3 note 0.01 2 ad 200 6000 2 lpg .
```

Three slopes

Every plain filter rolls off gently, at 12 dB per octave. This is quite musical and stays out of the way. Put an `s` in front and you get the steeper 24 dB versions: `slpf`/`slpq`, `shpf`/`shpq`, and `sbpf`/`sbpq`. They cut harder past the cutoff and give you a cleaner single resonance peak. Near a resonance of `1` that peak rings the cutoff

into an audible pitch. These filters never break into self-oscillation. Reach for them when the gentle slope lets too much through:

```
saw snd
c3 note
8000 300 2 slide slpf
0.7 slpq
4 decay
.
```

The ladder filter

The `l` versions model a Moog ladder: `llpf`/`llpq`, `lhpf`/`lhpq`, and `lbpf`/`lbpq`. They are warmer and more aggressive than the other two slopes. At high resonance they self-oscillate into a pure tone of their own. An envelope is the classic way to drive one. Open a ladder lowpass with `env` and hold the resonance high for that singing, rubbery sweep:

```
saw snd
c3 c4 c2 note
200 6000 0.02 2 0.2 1 env llpf
0.7 llpq
2 decay
.
```

SLOPE	LOWPASS	HIGHPASS	BANDPASS
gentle (12 dB)	`lpf`	`hpf`	`bpf`
steep (24 dB)	`slpf`	`shpf`	`sbpf`
ladder	`llpf`	`lhpf`	`lbpf`

Each cutoff word in the table has a matching resonance word, so `slpq` and `llpq` work just like `lpq` does for the gentle slope.

Key follow

A cutoff in Hz stays where you put it, whatever note plays. Set `lpf` to `800` and a low note keeps a dozen harmonics, while a high

note loses its fundamental and turns muffled and quiet. The patch falls apart as you climb.

`kf` ties the cutoff to the note instead. At `0`, the default, nothing changes. At `1` the cutoff rises an octave for every octave you play, so the sound keeps its tone the whole way up. Middle C is the pitch at which a cutoff means exactly the number you wrote. Play a line across four octaves and listen to the top notes come back:

```
saw snd
c2 c3 c4 c5 note
800 lpf
1 kf
1 decay
.
```

Half tracking is usually the musical answer. High notes stay a little darker than low ones, the way an acoustic instrument behaves, without the filter chasing the pitch exactly:

```
saw snd
c2 c3 c4 c5 note
600 lpf
0.5 kf
0.6 lpq
1 decay
.
```

One `kf` moves every cutoff at once, across all three slopes and all three shapes. It reads the note you played rather than the pitch you hear, so vibrato and glide leave the filter alone.

Auto-wah

`wah` is a resonant bandpass that the engine sweeps for you. It opens as the input gets louder and closes as the input dies away, the classic auto-wah voice. `wah` sets the wet mix from `0` to `1`. `wahpeak` sets the resonance of the swept band. `wahsens` sets how strongly the input envelope moves the filter. `wahmanual` sets the base cutoff in Hz the sweep starts from. A static setting on a plucky `vosc` already sounds like speech:

```
vosc snd
0 1 4 slide wave
c3 note
0.8 wah
0.7 wahpeak
0.5 wahsens
400 wahmanual
4 decay
.
```

word	sets
`wah`	wet mix, 0 to 1
`wahpeak`	resonance of the swept band
`wahsens`	envelope sensitivity
`wahmanual`	base cutoff in Hz

For broader, gentler tone shaping that lifts or drops a whole region, see *Equalization & cut/boosting* (7.2.2). For a tuned resonator that rings a pitch out of any input, see *Comb* (8.4) under Orbit effects.

7.2.2 Equalization & cut/boosting

Where a filter works by a moving cutoff, an EQ shapes tone band by band. A filter carves the spectrum from one edge; an EQ lifts or dips a chosen region and leaves everything else where it was. Cagire gives you a three-band parametric EQ and a one-knob tilt, so you can boost the lows, scoop a muddy mid, or open up the highs without a filter sweep. Unlike the filters, the EQ sits late in the voice, after the distortion and the swept effects and just before the envelope, so it shapes what those stages produced rather than what they are about to chew on (see *What are insert effects?* (7.1)):

```
pink snd 4 eqlo -6 eqhi 0.4 gain .
```

That boosts the low shelf and cuts the high shelf, turning flat `pink` noise into something darker and rounder.

Three bands

The three band gains are `eqlo`, `eqmid`, and `eqhi`, each set in decibels. A negative value cuts that band and a positive value boosts it. `0` should leave the band untouched. `eqlo` is a low shelf centred near 200 Hz, `eqmid` is a peak centred near 1000 Hz, and `eqhi` is a high shelf centred near 5000 Hz. Reach for them when you want to colour a voice's balance rather than carve a hard edge. Each band gain takes modulation, so the EQ can move. A triangle oscillation on `eqhi` rocks the high shelf up and down for a slow shimmer over a held chord:

```
saw snd
c3 note
maj7 chord
tri 8 stepsecs osc -6 6 range eqhi
0.2 gain
.
```

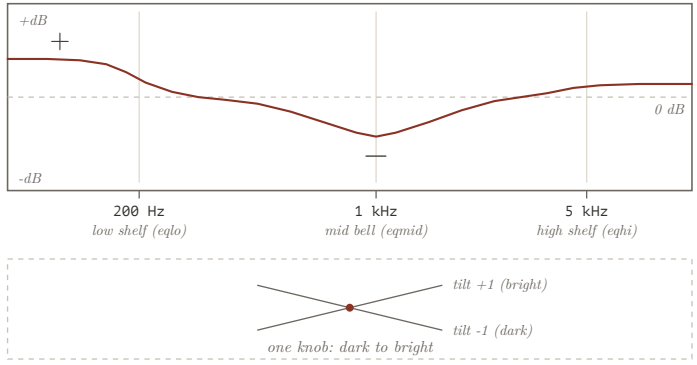


Figure 80: The three-band EQ lifts or dips each region in decibels; the inset shows tilt pivoting the whole spectrum from dark (-1) to bright (+1).

Moving the bands

The default centres suit most voices, but you can move each one with `eqlofreq`, `eqmidfreq`, and `eqhifreq`, all set in Hz. Push `eqmidfreq` up to chase a resonance, or drop `eqlofreq` to slide the

shelf under the bass rather than through it. The mid band also has a width control, `eqmidq`. At its default of `0.7` the mid is a broad, gentle bell. Higher values narrow it toward a notch, so a sharp `eqmidq` with a deep cut scoops a thin slice out of the tone:

```
white snd
-8 eqmid
6 eqmidq
0.4 gain
.
```

Tilt

`tilt` is the one-knob version of the whole idea. A single value runs from `-1` for dark, through `0` for flat, to `1` for bright, dropping the highs and lifting the lows as it goes negative and doing the reverse as it climbs. It is the fastest way to shade a voice without thinking in bands. Because it takes modulation, a `slide` can walk it from dark to bright across a phrase:

```
saw snd c2 note -1 1 2 slide tilt .
```

WORD	RANGE	SETS
`eqlo`	−24 to 24 dB	low shelf gain
`eqmid`	−24 to 24 dB	mid peak gain
`eqhi`	−24 to 24 dB	high shelf gain
`eqlofreq`	20 to 2000 Hz	low shelf frequency, 200 by default
`eqmidfreq`	100 to 10000 Hz	mid peak frequency, 1000 by default
`eqmidq`	0.2 to 8	mid bandwidth, 0.7 by default, higher is narrower
`eqhifreq`	1000 to 20000 Hz	high shelf frequency, 5000 by default
`tilt`	−1 to 1	one-knob spectral tilt, dark to bright

An EQ shapes balance where a moving cutoff carves from one edge (see *Lowpass / bandpass / highpass* (7.2.1)), and a band boosted hard enough is where tone-shaping turns into grit (see *Distortion* (7.3.2)).

7.2.3 Modal resonator

A modal resonator is a bank of eight tuned filters standing in for the modes of a struck body. Feed it anything and it rings. You meter how much of the voice pours into the bank with `modal`, tune it with `modalfreq`, set how long it rings with `modaldecay`, and morph what it is made of with `modalstruct`. It is a filter, not a voice: silence in, silence out.

The bank sits inside the voice, right after the filters, so the ring lives and dies with that voice's envelope. Give the voice a long `sustain` and hit it with something short:

```
rim snd 0.9 modal 220 modalfreq 6 modaldecay 4
sustain .
```

What excites it

The resonator hears whatever the source hands it, so the source is the mallet. Something already impulsive, a drum hit, a pluck, a clipped sample, strikes the bank and then gets out of the way, and what you hear afterwards is the ring:

```
clap snd 0.9 modal 330 modalfreq 8 modaldecay 6
sustain .
```

A source that never stops, like `white` or a `saw`, keeps driving the bank instead of striking it. That is a resonant filter rather than a struck body, hollow and vocal rather than plucked, and it is just as useful. It only stops sounding like a bell:

```
saw snd 55 note 0.7 modal 440 modalfreq 2 sustain .
```

The one thing the bank cannot do is outlast its voice. Because it lives inside the voice, closing the envelope closes the ring with it. If you want a tail that keeps sounding after the exciting voice has ended, that is what the orbit's comb is for. See *Comb* (8.4).

Tuning and ring time

`modalfreq` is the fundamental in Hz, the pitch mode 1 rings at, and every other mode sits at a ratio above it. Because it takes a

value like any other parameter, an alternation turns the resonator into a melody:

```
rim snd 0.9 modal < 110 220 165 > modalfreq 4
modaldecay 3 sustain .
```

`modaldecay` runs from `0.05` to `20` seconds and is mode 1's ring time. Raising it buys ring time, not loudness: the strike stays at the same level and simply hangs on longer. A `slide` walks it across eight steps, so each hit rings longer than the last:

```
hat snd 0.9 modal 165 modalfreq 0.2 8 8 slide
modaldecay 8 sustain .
```

Structure and brightness

`modalstruct` runs from `0` to `1` and decides what the body is made of. At `0` the partials are the harmonic series and the bank reads as a string. At `0.5` they are the ratios of a struck bar, wooden and slightly out of tune with themselves. At `1` they are a bell's minor-third partials, clangorous and metallic. Everything between is a blend, so a `jit` rolls the body to a fresh shape every couple of steps:

```
rim snd 0.9 modal 220 modalfreq 5 modaldecay 0 1 2
jit modalstruct 4 sustain .
```

`modalbright` runs from `0` to `1` and sets how long the upper modes ring next to mode 1. At `0` they die almost at once and only the fundamental is left ringing, which is the sound of a damped, muffled body. At `1` they hang on with it and the body stays bright the whole way down:

```
kick snd 0.9 modal 110 modalfreq 8 modaldecay 0.8
modalstruct 1 modalbright 6 sustain .
```

WORD	RANGE	SETS
`modal`	0 to 1	how much of the voice pours into the bank, 0 is bypass
`modalfreq`	Hz	the fundamental the bank rings at
`modaldecay`	0.05 to 20 s	mode 1's ring time

word	range	sets
`modalstruct`	0 to 1	the partials, 0 string, 0.5 bar, 1 bell
`modalbright`	0 to 1	how long the upper modes ring next to mode 1

The same bank is a patch word, so you can put it anywhere in a graph rather than at its fixed place in the voice. See *Patch filters* (4.4).

7.2.4 Vowel formant filter

A vowel is a chord of resonances. The `vowel` filter carves five of them into any source, tuned from the classic CSound formant tables, and the values interpolate across the vowel axis: `0` is *a*, `1` is *e*, `2` is *i*, `3` is *o*, `4` is *u*, and every fraction between is a mouth mid-morph. Setting it turns the filter on; leaving it unset costs nothing.

```
saw snd e2 note 0 vowel 2 sustain .
```

A rich source works best. Saws, pulses and noise carry energy everywhere the formants sit; a sine has nothing to carve:

```
white snd 2 vowel 0.3 decay .
```

Morphing

Because the formant values interpolate, a motion on `vowel` sings through the alphabet:

```
saw snd e2 note
saw 3 osc 0 4 range vowel
4 sustain .
```

`drift` mumbles instead:

```
saw snd a2 note
drift 0.5 osc 0 4 range vowel
6 sustain .
```

Voice type

`vowtype` picks which row of the formant tables the filter reads: `0` alto, `1` bass, `2` countertenor, `3` soprano, `4` tenor. The default is bass. Match it roughly to the register of the source: the soprano tables sit high and do little to a bass line.

```
saw snd c4 note 0 vowel 3 vowtype 2 sustain .
```

The first two formants bend with the note, so high notes stay singable instead of falling off the top of the tables. For a voice whose mouth actually moves, see *Vocal synthesis* (2.4).

7.3 Distortion & Degradation

7.3.1 Bitcrushing & sample rate

Two effects are used to destroy a signal on purpose, each in its own way. `crush` throws away bits of resolution, quantising the signal to fewer levels until it turns gritty and square. `coarse` throws away samples in time, holding each one for several outputs so the effective sample rate drops. Both are the sound of cheap digital audio.

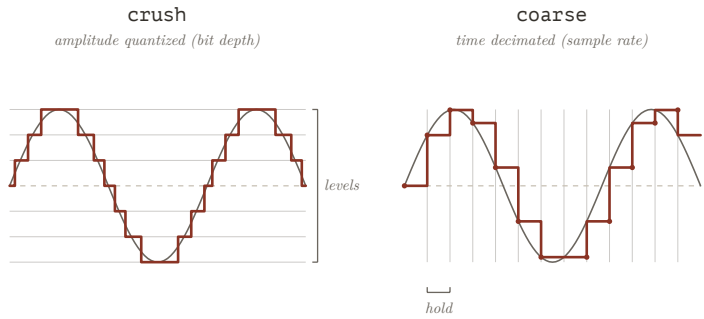


Figure 81: The same sine through each lo-fi effect: crush stair-steps the wave vertically in amplitude, coarse stair-steps it in time.

Bit depth

`crush` sets the bit depth, from `1` for a square crunch up to `16` for clean. Lower numbers leave fewer levels for the signal to land on, so the waveform stair-steps and the tone hardens. Near `16` you barely hear it; down around `1` only the gross shape of the sound survives. A `slide` walks the bit depth itself, stepping from `16` down over a few steps so a saw decays from clean to crunchy:

```
saw snd
c2 note
16 1 4 slide crush
8 decay
0.5 gain
.
```

Sample-rate reduction

`coarse` is a decimation factor. At `1` every sample passes. At `4` the engine holds each sample for four outputs and the effective rate falls to a quarter, smearing the highs into aliased ringing. The higher the factor, the coarser the sound. A saw ramp sweeps the factor up and back, a classic chiptune rate-sweep on a pulse:

```
pulse snd
c2 note
saw 2 stepsecs osc 1 8 range coarse
0.5 gain
.
```

word	range	sets
`crush`	1 to 16 bits	bit depth, 16 is clean
`coarse`	1 to 128	decimation factor, 1 passes every sample

`crush` and `coarse` are the lo-fi corner of a larger family; for harder clipping of the signal see *Distortion* (7.3.2), and for folding it back on itself see *Wavefolding* (7.3.3).

7.3.2 Distortion

Distortion adds harmonics by reshaping the waveform, anywhere from a gentle warmth to a hard clip. Cagire also bundles the analog degradation of vinyl and cassette in the same stage. You run a source through `distort`, pick the curve it bends through, and trim the level coming out the other side. A `pluck` driven through the `tanh` curve warms up without losing its attack:

```
pluck snd
c3 note
6 distort
tanh dmode
.
```

Drive

`distort` sets the drive amount, and it is effectively unbounded. A small value adds a touch of warmth, a large one folds the signal into a hard clip, so keep pushing it when you want more bite. How hard that drive bites also depends on what feeds it. `gain` sits before the distortion and sets the level going in (see *Gain and amplitude* (3.4)), so it is as much a distortion control as `distort` is: the same drive amount sounds tame on a quiet voice and savage on a loud one. `distortvol` trims the makeup level coming out, from `0` to `2`, so you can claw back the loudness a heavy drive piles on. Here is a short example of distortion used on a long kick:

```
kick snd
0 9 2 slide distort
0.7 distortvol
4 decay
.
```

Saturator curves

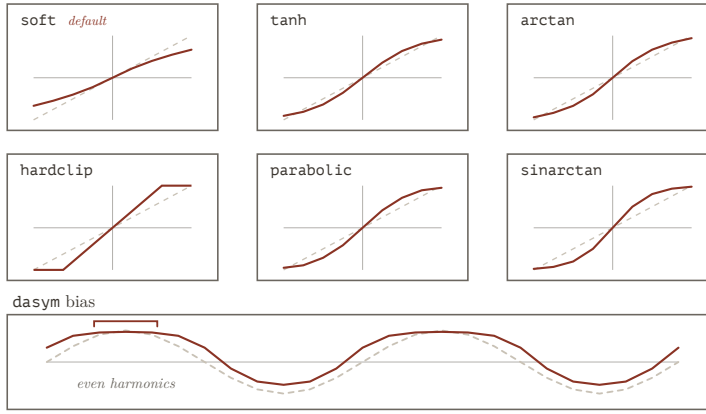


Figure 82: The six saturator curves dmode selects, each an input-to-output transfer against the no-distortion diagonal, with a dasym bias clipping one waveform unevenly.

`distortmode`, also `dmode`, chooses the curve the saturator bends the signal through. It defaults to `soft`, a gentle rounding. The others trade that softness for harder edges and different harmonic flavours.

CURVE	CHARACTER
`soft`	gentle rounding, the default
`tanh`	smooth, rounded saturation
`arctan`	similar to `tanh` with a slightly sharper knee
`hardclip`	a flat hard clip, the most aggressive
`parabolic`	a parabolic shaping curve
`sinarctan`	a sine and arctangent blend

Drop an alternation in place of a single name and the saturator changes shape on every iteration, so a held pattern cycles through soft, smooth, and brutal in turn:

```
saw snd
c2 note
7 distort
< soft tanh hardclip > dmode
0.5 postgain
2 decay
.
```

Bias

`distortasym`, also `dasym`, adds a bias from `-1` to `1` before the saturator. An offset waveform clips unevenly, which brings in even harmonics and the tube-like asymmetry a symmetric curve cannot. At `0` the distortion stays symmetric; push it either way for more of that even-harmonic colour. Move the bias with a slow triangle and the even harmonics swell in and out across the note:

```
saw snd
c2 note
8 distort
tri 2 stepsecs osc -0.6 0.6 range dasym
2 decay
.
```

Vinyl and cassette

The same stage carries Cagire's analog degradation. `vinyl` sets the character mix from `0` to `1`, blending in the wear of a record or a tape. `vinylwow` sets the wow and flutter depth, the slow pitch drift of a warped disc or a slipping capstan. `vinylnoise` adds hiss. `vinyltone` tilts the tone from `-1` dark to `1` bright. `vinyltype` picks the voicing, one of `dull` (the default), `clear`, or `cassette`. None of the vinyl controls take modulation: you set the character once per voice and the stage stays out of the chain entirely while `vinyl` is `0`.

VOICING	CHARACTER
`dull`	muffled record wear, the default
`clear`	a cleaner, brighter record

voicing	character
`cassette`	tape, with its own wow and hiss colour

A `break` sample takes well to a cassette voicing, mixed in with a little wow:

```
break snd 0.6 vinyl cassette vinyltype 0.4 vinylwow .
```

Lay a heavy dose of character under a clean `sine` and let a touch of `vinylnoise` settle beneath it:

```
sine snd c4 note 0.8 vinyl 0.3 vinylnoise -0.4
vinyltone .
```

word	range	sets
`distort`	0 and up	drive into the saturator
`distortvol`	0 to 2	makeup level after the saturator
`distortmode` / `dmode`	soft, tanh, arctan, hardclip, parabolic, sinarctan	the saturator curve
`distortasym` / `dasym`	−1 to 1	bias before the saturator, for even harmonics
`vinyl`	0 to 1	analog character mix, 0 is bypass
`vinylwow`	0 to 1	wow and flutter depth
`vinylnoise`	0 to 1	hiss level
`vinyltone`	−1 to 1	tone tilt, dark to bright
`vinyltype`	dull, clear, cassette	the voicing

For the digital side of lo-fi, bit-depth and sample-rate reduction, see *Bitcrushing & sample rate* (7.3.1); for pushing the waveform past a clip into a fold, see *Wavefolding* (7.3.3).

7.3.3 Wavefolding

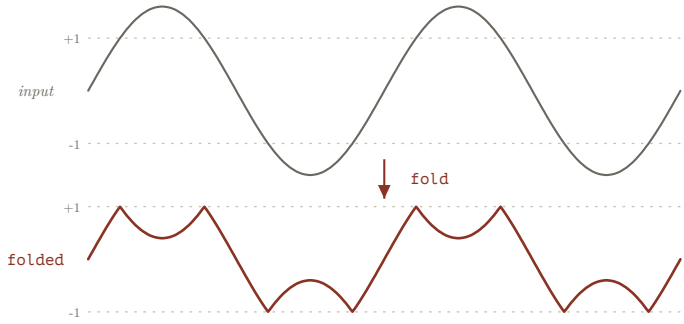


Figure 83: Once the driven signal crosses the threshold it folds back on itself, and each reflection adds bright partials.

Wavefolding is when you fold and reflect a signal back on itself once it crosses a threshold. Every reflection adds bright harmonics that grow with the level driving them. It is most audible on clean tones, a `sine` or a `tri`, where there is little harmonic content to begin with and the new partials stand out clearly. A small amount already changes the timbre.

```
sine snd c4 note 0.4 fold .
```

Folding

`fold` sets the fold amount from `0` to `1`. At `0` the signal passes untouched. As you raise it, the peaks fold back across the threshold and the tone brightens. This ends up turning a pure `sine` into something harsh and metallic. Because the effect feeds on level, how hard `gain` drives the voice changes how much it bites (see *Gain and amplitude* (3.4)):

```
sine snd
c2 note
8 stepsecs rise 0.9 0 expmap fold .
```

Reading one value per step gives a slower, looser drift. `sample` takes a single number off a sine whose cycle you give in beats, so `2`

`beatsecs` completes one every two beats; scale it with `range` and feed that into `fold` to breathe the brightness in and out:

```
sine snd c4 note sine 2 beatsecs osc sample 0 0.8
range fold .
```

Fold shapes

`foldmode`, also `fmode`, chooses the curve the signal reflects against. `triangle` is the default and folds with hard corners for the brightest, most odd-harmonic result. `sine` rounds those corners for a softer, hollower fold. `wrap` is a harsher saw-style wrap that snaps rather than reflects.



Figure 84: The three curves the signal reflects against: triangle keeps hard corners, sine rounds them off, wrap snaps back like a saw.

SHAPE	CHARACTER
`triangle`	default, hard corners, odd harmonics
`sine`	rounded, softer and hollower
`wrap`	saw wrap, harsher and snappier

Pairing a moving `fold` with a chosen shape is where the character lives. A slow triangle rocks the amount up and down while the `sine` shape keeps the fold rounded on a `tri`:

```
tri snd c4 note sine foldmode tri 4 stepsecs osc 0
0.7 range fold .
```

You can also cycle the shape itself. An alternation hands `fmode` a different curve on each iteration, so one steady `fold` reads differently every note:

```
tri snd c4 note < triangle sine > fmode 0.5 fold .
```

Wrapping

`wrap` is a related but more aggressive reflection, set from `0` to `10`. Where `fold` mirrors the signal smoothly, `wrap` lets the phase run past its bound and jump straight back, so the result is harder and richer even at modest settings. A saw ramp sweeping the amount turns a `vosc` into a slowly tearing drone:

```
vosc snd c3 note saw 8 stepsecs osc 0 6 range wrap .
```

WORD	RANGE	SETS
`fold`	0 to 1	fold amount, 0 is bypass
`foldmode` / `fmode`	triangle, sine, wrap	the curve the signal reflects against
`wrap`	0 to 10	saw-style wrap amount, harder than a fold

For the broader saturation and clipping stages that sit alongside it, see *Distortion* (7.3.2); for the lo-fi cousin that thins resolution rather than reflecting it, see *Bitcrushing & sample rate* (7.3.1).

7.4 Space & time effects

7.4.1 Chorus

Chorus, flanger, and phaser are a family of movement effects built on the same trick: a short delay whose time an oscillator sweeps, mixed back against the dry voice. That moving delay is what adds motion and width, and the length of the delay is the whole difference between them, from a subtle thickening to a jet-engine sweep. Chorus is the gentlest of the three, a slow sweep on the longest delay:

```
tri snd c4 note 1 chorus .
```

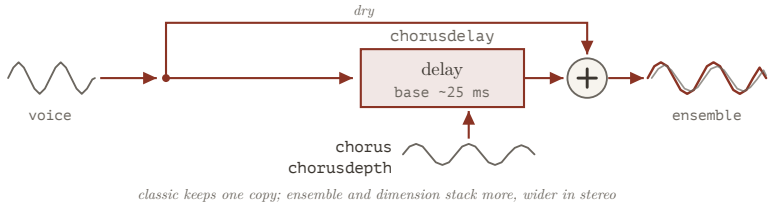


Figure 85: An oscillator sweeps the short delay time against the dry voice; chorusdepth sets how far it swings.

`chorus` sets the rate of the sweep, in Hz from `0` (bypass) to `100`, and `chorusdepth`, from `0` to `1`, its intensity, how far the delay swings. Together they take a single oscillator and turn it into a small, detuned ensemble. `chorusdelay` is the base delay the sweep moves around, a fixed time in milliseconds from `0` to `100` and about `25` by default, so you leave it alone more often than not. Stretch it toward the top of its range and the detuning widens into a slow, seasick vibrato:

```
saw snd c3 note .3 chorus .5 chorusdepth 90
chorusdelay .
```

`chorustype`, also `ctype`, chooses the voicing. It defaults to `classic`, a single clean voice. `ensemble` is a lush Juno-style four-voice wash, and `dimension` is a wider stereo spread:

```
saw snd
c3 note
maj7 chord
.2 chorus
0.1 chorusdepth
< ensemble classic dimension > chorustype
.5 postgain
.
```

VOICING	CHARACTER
`classic`	single clean voice, the default
`ensemble`	lush Juno four-voice wash
`dimension`	wide stereo spread

Here are all the parameters you can use to set the chorus effect module:

WORD	RANGE	SETS
`chorus`	0 to 100 Hz	sweep rate, 0 is bypass
`chorusdepth`	0 to 1	modulation intensity
`chorusdelay`	0 to 100 ms	base delay the sweep moves around
`chorustype` / `ctype`	classic, ensemble, dimension	voicing

For the shorter, feedback-driven cousin see *Flanger* (7.4.2), and for swept notches rather than a swept delay see *Phaser* (7.4.3). For the longer, blurrier versions of these effects see *Smearing* (7.4.4), and for shifting pitch rather than sweeping it see *Frequency & pitch shifting* (7.5).

7.4.2 Flanger

A flanger uses a shorter delay than a chorus and feeds its own output back in. Where the chorus thickens, the flanger combs: the swept delay carves a row of evenly spaced notches that slide up and down the spectrum, the hollow, swooshing sound of a jet passing overhead. It bites hardest on bright, sustained material, so a held saw makes the sweep plain.

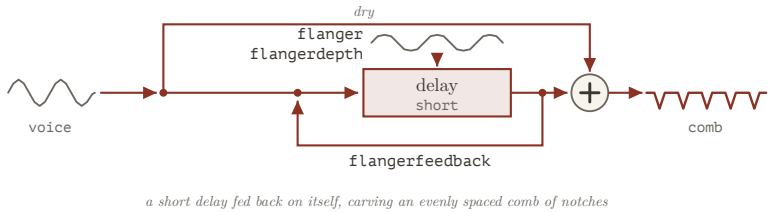


Figure 86: A short delay feeds back on itself and sweeps against the dry voice, carving an evenly spaced comb of notches that slide across the spectrum.

```
saw snd c2 note 0.2 flanger 0.7 flangerdepth .
```

`flanger` is the speed of the sweep, `0` (bypass) to `100` Hz, and `flangerdepth`, from `0` to `1`, how far the delay swings, which is how wide a band the notches travel. Slow and deep is the classic setting, the sweep rolling across the held tone all on its own. `flangerfeedback` runs from `0` to `0.95` and feeds the comb back into itself. The higher you push it the harder the notches ring, and near the top the flanger tips into a metallic, whistling resonance that almost sounds a pitch of its own:

```
saw snd c2 note 0.2 flanger 0.7 flangerdepth 0.85
flangerfeedback .
```

The effect is loudest on broadband material, where the comb has every frequency to chew on. White noise through a flanger is the textbook demonstration, the notches turning a flat hiss into a breathing, resonant whoosh:

```
white snd 0.4 flanger 0.9 flangerdepth 0.8
flangerfeedback .
```

`flangermode`, also `flmode`, defaults to `classic`. Switch it to `throughzero` for the deep zero-crossing sweep, where the moving delay crosses the dry signal and briefly cancels it outright, the tape-flanger roar. With the feedback near the ceiling on noise it is at its most extreme:

```
white snd 0.3 flanger 0.9 flangerdepth 0.9
flangerfeedback throughzero flangermode .
```

The rate is a parameter like any other, so you can sweep it. A slow `slide` spins the flanger up from nearly still to a fast shimmer over eight steps, the jet accelerating as the note holds:

```
saw snd c2 note 0.05 3 8 slide flanger 0.8
flangerdepth .
```

MODE	CHARACTER
`classic`	one-sided sweep, the default
`throughzero`	the delay crosses the dry signal, deep tape-flanger roar

The flanger module answers to these parameters:

WORD	RANGE	SETS
`flanger`	0 to 100 Hz	sweep rate, 0 is bypass
`flangerdepth`	0 to 1	how far the delay swings
`flangerfeedback`	0 to 0.95	feedback into the comb
`flangermode` / `flmode`	classic, throughzero	sweep mode

For the gentler, longer-delay version see *Chorus* (7.4.1), and for swept notches rather than a swept delay see *Phaser* (7.4.3). *Smearing* (7.4.4) holds the longer, blurrier relatives of these effects, and *Frequency & pitch shifting* (7.5) moves the pitch instead of sweeping it.

7.4.3 Phaser

A phaser sweeps a stack of notches across the spectrum like a flanger, but it carves them with allpass filters rather than a delay. The notches are fewer and unevenly spaced, so the motion is smoother and more vocal, less metallic than the flanger's comb, the slow rotary swirl of a guitar through a phase pedal. It shows best on a sustained, harmonically rich source, so a held saw is the place to start.

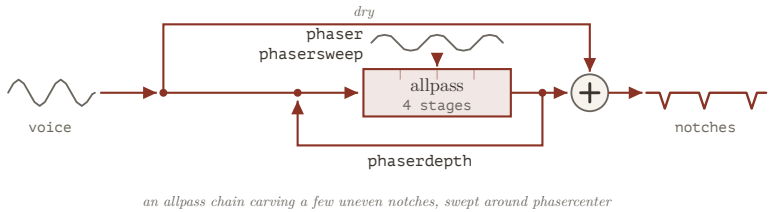


Figure 87: A phaser mixes the dry voice against a chain of allpass stages, sweeping a few unevenly spaced notches across the spectrum.

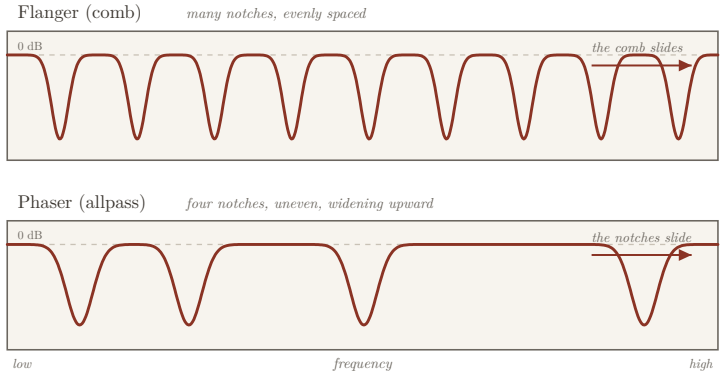


Figure 88: The flanger's comb carves many evenly spaced notches that slide together; the phaser's allpass stages carve four uneven ones that widen toward the top of the spectrum.

```
saw snd c2 note 0.3 phaser .
```

`phaser` drives the sweep, from `0` (bypass) up to `100` Hz. On its own, with everything else left at its default, that already gives the gentle, rolling movement above. `phaserdepth` is the feedback resonance, from `0` to `0.95`. It sharpens the notches as it climbs, deepening them from a soft wash into a pronounced, almost-vocal sweep that whistles toward the top of its range:

```
saw snd
c2 note
0.3 phaser
0.9 phaserdepth
.
```

`phasersweep` sets how far the notches travel, measured in cents from `0` to `20000`, and `phasercenter` the base frequency they sweep around, in Hz over the same range. Together they place the effect: a narrow sweep low down is a subtle wobble, a wide one higher up is a dramatic, full-range swirl:

```
saw snd
c2 note
0.2 phaser
0.8 phaserdepth
900 phasersweep
800 phasercenter
.
```

Like the flanger, the rate is a parameter you can drive, and modulating it is where the phaser comes alive. A slow `slide` walks it from nearly still up to a gentle wobble over eight steps:

```
saw snd
c2 note
0.1 2 8 slide phaser
0.7 phaserdepth
.
```

Sweeping `phasersweep` with a saw ramp opens and closes the reach of the notches instead, a slow widening and narrowing that pairs well with a single held tone:

```
saw snd
c2 note
0.5 phaser
0.8 phaserdepth
saw 8 stepsecs osc 200 1200 range phasersweep
.
```

word	range	sets
`phaser`	0 to 100 Hz	sweep rate, 0 is bypass
`phaserdepth`	0 to 0.95	feedback resonance, sharpens the notches
`phasersweep`	0 to 20000 cents	how far the notches travel
`phasercenter`	0 to 20000 Hz	base frequency they sweep around

For the harder-edged comb built from a delay see *Flanger* (7.4.2), and for the gentle thickening of detuned copies see *Chorus* (7.4.1). When the movement should blur rather than sweep, go to *Smearing* (7.4.4); when the pitch itself should move, *Frequency & pitch shifting* (7.5).

7.4.4 Smearing

`smear` is a chain of twelve allpass filters that blurs a sound's transients in time without touching its pitch. It runs late in the voice, after the phaser and the flanger and just before the envelope (see *What are insert effects?* (7.1)). Each allpass holds the frequencies around its break point back by a different amount, so a sharp hit spreads into a soft wash while the note itself stays exactly where it was. Because nothing changes but phase, a static setting is gentle by design; you hear the effect most when its break frequency moves:

```
saw snd
100 200 freq
4 gate
0.6 0.9 rand smear
0.0 0.9 rand smearfb
.
```

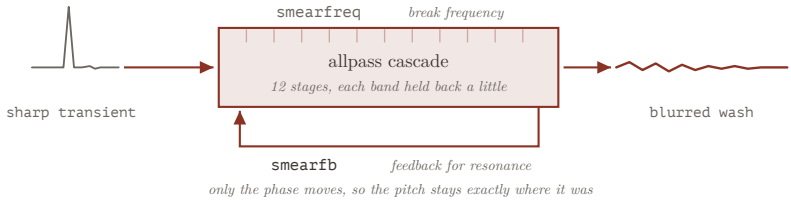


Figure 89: A transient enters a cascade of twelve allpass filters and spreads into a soft wash; smearfb feeds the chain back for resonance.

Diffusion

`smear` sets the wet-dry mix from `0` to `1`, where `0` is bypass and `1` is the fully smeared signal. It works on the edge of a sound, so a transient-rich source like a hat, a rim, or a snare shows it off best; the leading click softens and trails into the body of the hit. On a sustained, slowly evolving tone there is little transient to spread, so the same mix does much less. With the break frequency held still, as in the example above, the blur is quiet and easy to miss. That is the point: `smear` on its own is a finishing touch, not a statement.

Moving the break frequency

`smearfreq` is the allpass break frequency in Hz, from `20` to `20000`, and modulating it is the move that turns a subtle blur into the whole reason you reached for the effect. As the break point slides, the band of frequencies being delayed sweeps with it, and the smear shimmers across the spectrum. A triangle oscillation walking `smearfreq` up and down under a cymbal is the headline gesture:

```
cymbal snd tri 2 stepsecs osc 200 3000 range
smearfreq 0.7 smear .
```

Resonance

`smearfb` feeds the smear back into itself, `0` to `0.95`. Low values keep it airy and diffuse. Push it up and the allpass chain starts to

ring around its break frequency, lending the blur a pitched, metallic edge. Pair a high `smearfb` with a slow `slide` sweep of `smearfreq` over a `pluck` and the ring slides through the decay like a weird alien noise:

```
pluck snd c3 note 200 5000 8 slide smearfreq 0.8
smearfb 0.7 smear .
```

Keep `smearfb` short of its ceiling unless you want the resonance to dominate the sound; near `0.95` the ring outlasts the note it came from.

word	range	sets
`smear`	0 to 1	wet/dry mix, 0 is bypass
`smearfreq`	20 to 20000 Hz	allpass break frequency
`smearfb`	0 to 0.95	feedback for resonance

For the related phase and movement effects, see *Chorus* (7.4.1), *Flanger* (7.4.2), and *Phaser* (7.4.3), and for shifting pitch rather than blurring it in time, see *Frequency & pitch shifting* (7.5).

7.5 Frequency & pitch shifting

There are two ways to move the pitch of a voice, and they are not the same operation. Both run late in the chain, after the phaser and the flanger, so they transpose the voice as the earlier effects left it rather than the bare source (see *What are insert effects?* (7.1)). A frequency shifter adds a fixed number of Hz to every partial at once, so the harmonics stop lining up and the tone goes inharmonic and metallic. A pitch shifter transposes by musical semitones, multiplying every frequency by the same ratio, so the harmony stays intact and a note simply moves up or down the scale. Cagire gives you one word for each.

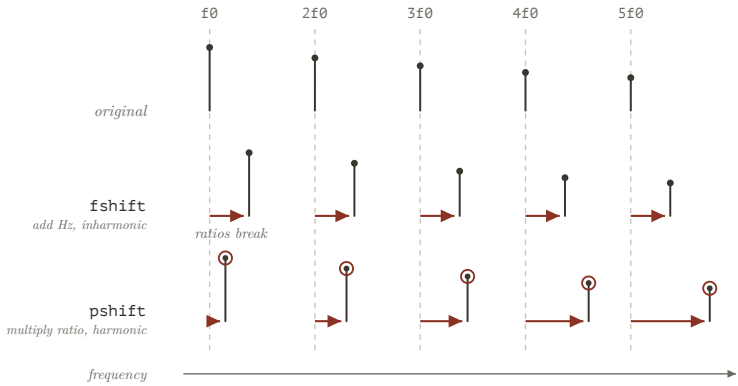


Figure 90: Frequency shifting adds a fixed number of Hz to every partial; pitch shifting multiplies each by a common ratio.

7.5.1 Frequency shifting

`fshift`, also `fsh`, shifts the whole spectrum by a signed amount in Hz, from about `-2000` to `2000`, with `0` as bypass. A small shift detunes the partials against each other and makes them clang and beat; a large shift drags the tone fully metallic and bell-like. Because the offset is a flat number of Hz rather than a ratio, low partials move proportionally further than high ones, which is exactly what breaks the harmonic series. A slow upward `slide` on `fshift` smears a plain sine off its own overtones and into an inharmonic haze:

```
sine snd c4 note 0 400 8 slide fshift .
```

Drift the shift with a smooth-random `drift` oscillation and a morphing `vosc` turns into a metallic bell that never quite settles:

```
vosc snd c5 note drift 4 stepsecs osc -50 50 range
fshift .
```

7.5.2 Pitch shifting

`pshift`, also `psh`, transposes the voice by a signed number of semitones, from `-24` to `24`, with `0` as bypass. Twelve semitones is an octave, seven a fifth. Unlike `fshift` it keeps the harmony, so it is the word to reach for when you want a harmony line, an

octave doubling, or a transposition that still sounds like the same instrument.

Feed `pshift` an alternation and a single source spells out a chord one iteration at a time. Here a `pluck` walks root, fifth, and octave:

```
pluck snd c3 note < 0 7 12 > pshift .
```

A square oscillation flips between two values with no glide, so it snaps a `break` sample between its own pitch and an octave up on every cycle:

```
break snd square 4 stepsecs osc 0 12 range pshift .
```

7.5.3 Grain window

`pshift` works by chopping the voice into short grains and respacing them, and `pshiftwin`, also `pwin`, sets how long each grain lasts in milliseconds, defaulting to `40`. A small window sounds grainy and stuttered, a large window smoother but more smeared in time. The window does nothing on its own; it shapes the texture of whatever `pshift` is doing. The same fixed `pshift` sounds rough through a short window:

```
saw snd c4 note 5 pshift 8 pshiftwin .
```

Widen the window and the same transposition turns smooth and liquid:

```
saw snd c4 note 5 pshift 120 pshiftwin .
```

word	range	sets
`fshift` / `fsh`	−2000 to 2000 Hz	flat spectral shift, 0 is bypass
`pshift` / `psh`	−24 to 24 semitones	musical transposition, 0 is bypass
`pshiftwin` / `pwin`	5 to 200 ms	grain window, 40 by default

For the gentler movement of detuned and swept copies rather than outright transposition, see *Chorus* (7.4.1), *Flanger* (7.4.2), and *Phaser* (7.4.3).

Chapter 8

Orbit effects

8.1 What are orbits?

An orbit is a shared send bus. An insert effect lives inside one voice and dies with it (see *What are insert effects?* (7.1)). An orbit sits outside the voices, and many voices can feed it at once. A reverb on an orbit is one room that every voice sent to it plays into together, so they share the same tail instead of each carrying a private copy. This is the SuperDirt model, which is pretty popular among live coders. It is why the reverb, the delay, the comb, the feedback, and the compressor all live here rather than in the voice's own chain.

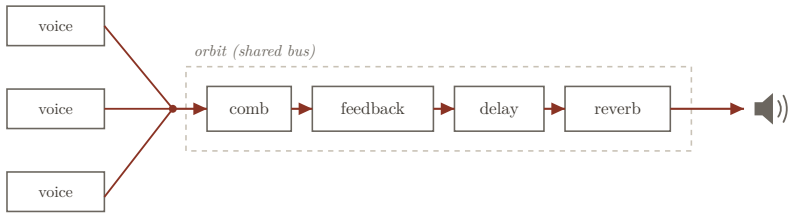


Figure 91: Many voices feed one orbit, which runs a fixed effect chain and passes the shared result on.

There are eight orbits, numbered `0` to `7`. A voice picks one with `orbit`, and `0` is the default, so everything lands on the same orbit until you say otherwise. Because they all arrive at the same output, you cannot tell them apart by ear once they are mixed. Cagire meters each one separately so you can (see *Monitoring orbits* (9.4)). The send words meter how much of a voice pours into an orbit's effects. `verb` feeds the reverb, `delay` the delay, `comb` the comb, and

`feedback` the feedback. A send of `0` is dry, and turning it up pours more of that voice into the shared effect:

```
saw snd
c4 note
0.3 verb
.
```

8.1.1 A shared space

The point of a send bus is sharing. Send two different voices to the same orbit and they wash into one reverb together, the way instruments in a room share its acoustics. Here a kick and a snare both feed the default orbit's reverb, so one tail carries both:

```
kick snd 0.4 verb .
```

```
snare snd 0.4 verb .
```

Because the effect lives on the orbit and not on the voice, its tail keeps sounding after the voice that fed it has ended. A short stab into a long reverb rings on well after the stab is gone. This is the same behaviour the comb relies on to turn a noise burst into a sustained pluck (see *Comb* (8.4)).

8.1.2 The chain on each orbit

Each orbit runs its effects in a fixed order: comb, then feedback, then delay, then reverb. A patch of your own runs last if you have hung one there with `dsp` (see *Patches as effects* (2.5)). In that regard, it is rather similar to the insert effects chain we described earlier. The order runs from the short, tonal effects to the long, spatial ones. The reverb sits last of the built-in sends, so it catches the delay's echoes and washes them into the same space. As with the insert chain, typing the sends in a different order changes nothing. The orbit decides what runs first.

The four built-in effects are sends. Each one adds its wet on top of a dry that always reaches the output, which is why you set them as amounts rather than as mixes. The `dsp` patch is the exception. It is an insert that replaces the bus with what it returns, so it is

the only place on an orbit where an effect can take something away rather than add to it.

8.1.3 Sticky settings

An orbit holds its settings. Set a `verbdecay` or a `delaytime` and it stays that way for every voice on that orbit, across later emits, until you change it. Cagire converts `delaytime` to seconds when it emits the update, so a later BPM change does not retime the stored setting. See *Delays* (8.3) for the whole-note fractions it accepts. The send level is what you tend to vary from voice to voice. You set the character of the space once and leave it:

```
saw snd c3 note 0.3 verb 0.8 verbdecay .
```

The pages that follow take the shared effects one at a time: *Reverbs* (8.2), *Delays* (8.3), *Comb* (8.4), *Feedback* (8.5), and the compressor that ducks one orbit from another in *Compression & sidechain* (8.6).

8.2 Reverbs

Reverb is a great way to add space to any sound, and it is totally okay to put too much of it on everything. You send a voice into it with the `verb` word, which takes values between `0` to `1` where `0` is dry. Everything on the orbit shares one reverb, so the voices sound as if they are in the same room (see *What are orbits?* (8.1)). A small send is enough to seat a sound in a space. Push it up and the voice swims in the tail:

```
pluck snd c4 note 0.4 verb .
```

8.2.1 Two algorithms

`verbtype` picks the reverb algorithm, named as a bareword. There are two reverb models available. `space` is the default, a general-purpose hall. `cloud` is a denser, more diffuse tail that smears the input rather than echoing it. If you are lazy, you can just type `0` for `cloud` and `1` for `space`. If you are feeling witty, you can even pattern that:

```
hat snd
0.5 verb
< 0 1 > verbtype
.
```

8.2.2 Sizing the tail

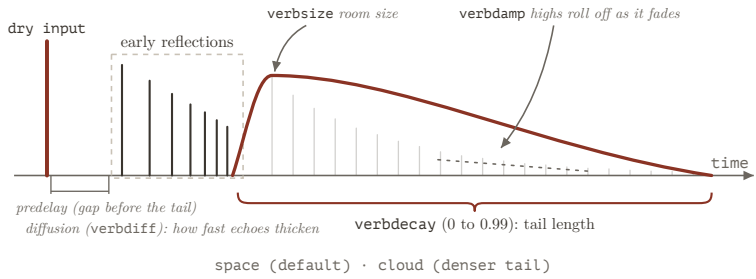


Figure 92: A reverb impulse over time: dry input, a predelay gap, early reflections, then a diffuse tail whose length verbdecay sets.

A handful of parameters shape the space. `verbdecay` runs from `0` to `0.99` and sets how long the tail rings, the single biggest control over how large the space feels. `verbsize`, also `vsize`, sets the room size, and both algorithms answer to it. `verbpredelay` holds the tail back a moment after the dry sound, on `space` only. `verbdiff` sets diffusion, how quickly the early echoes thicken into a wash, on `cloud` only. Not every control reaches both models, and the table at the end of the page names the algorithm each one belongs to. Here `verbdecay` opens the tail right up:

```
rim snd 0.5 verb 0.9 verbdecay .
```

`verbdecay` takes modulation, so the size of the space can move under a part. Here it opens up across eight steps:

```
snare snd 0.5 verb 0.2 0.9 8 slide verbdecay .
```

8.2.3 Damping and tone

`verbdamp` rolls the high end out of the tail as it decays, from `0` (bright) to `1` (dark), so the reverb loses its top as it fades the

way a real room does. It belongs to `cloud`, so reach for it with that algorithm named:

```
saw snd c3 note 0.5 verb cloud verbtype 0.8
verbdamp .
```

The tail also has a three-band tone control, and this one works on both algorithms. `verblowcut` and `verbhighcut` set where the bands split, and `verblowgain` and `verbhighgain` lift or drop the low and high bands. On `space`, `verbprelow` and `verbprehigh` filter the signal going in, before it ever reaches the tail:

```
saw snd c3 note 0.5 verb 0.2 verblowcut 0.8
verbhighgain .
```

8.2.4 Movement

`verbchorus` modulates the tail so it moves slightly instead of sitting still, and `verbchorusfreq` sets how fast that movement runs. Both go from `0` to `1`. A little keeps a long tail from sounding static:

```
saw snd c3 note 0.5 verb 0.6 verbchorus .
```

word	range	algorithm	sets
`verb`	0 to 1	both	send level into the reverb
`verbtype`	space, cloud	both	algorithm
`verbdecay`	0 to 0.99	both	tail length
`verbsize` / `vsize`	0 to 1	both	room size
`verbdamp`	0 to 1	cloud	high-frequency damping of the tail
`verbdiff`	0 to 1	cloud	diffusion
`verbpredelay`	0 to 1	space	gap before the tail
`verbprelow` / `verbprehigh`	0 to 1	space	input filtering
`verblowcut` / `verbhighcut`	0 to 1	both	band crossover points
`verblowgain`	0 to 1	both	low-band gain
`verbhighgain`	0 to 1	both	high-band gain

word	range	algorithm	sets
`verbchorus`	0 to 1	both	tail modulation amount
`verbchorusfreq`	0 to 1	both	tail modulation rate

For the orbit these sends live on, see *What are orbits?* (8.1); for the echoes a reverb often sits behind, see *Delays* (8.3).

8.3 Delays

The delay is the orbit's echo. Send a voice into it with `delay`, a level from `0` to `1`, and it repeats the sound at an interval you set, each repeat quieter than the last. Like the reverb it sits on the orbit and is shared, so several voices can feed one echo (see *What are orbits?* (8.1)):

```
pluck snd c4 note 0.4 delay .
```

8.3.1 Time and feedback

`delaytime` sets the gap between repeats as a fraction of a whole note. Cagire uses the global BPM to convert that fraction when it emits the command: one whole note lasts `240 / BPM` seconds. This does not use a pattern's local speed or an `at` window.

musical gap	fraction of a whole note	decimal value
sixteenth	`1/16`	`0.0625`
eighth triplet	`1/12`	approximately `0.0833333333`
eighth	`1/8`	`0.125`
dotted eighth	`3/16`	`0.1875`
quarter	`1/4`	`0.25`

Write the fraction directly: `1/8 delaytime` and `0.125 delaytime` mean the same thing. At 120 BPM, this eighth-note echo repeats every 250 milliseconds:

```
rim snd 0.4 delay 1/8 delaytime .
```

The units are the same for a sound, a bare orbit update, a global `all` setting, and the values of a motion. Modulation bounds are whole-note fractions too:

```
hat snd 0.4 delay drift 0.5 stepsecs osc 1/160 1/4
range delaytime 0.7 delayfeedback .
```

The conversion happens when Cagire emits the command. The orbit stores the resulting seconds. A later BPM change does not retime that stored setting, so emit `delaytime` again to set a new delay at the new BPM.

For a fixed time in seconds, divide by the duration of a whole note before setting the parameter: `0.25 4 beatsecs / delaytime` sets a 250-millisecond delay at any tempo.

Only `delaytime` uses whole-note fractions. `fbtime` and all other effect parameters keep their existing units.

`delayfeedback` sets how much of each repeat feeds back into the line, from `0` to `1`, which is how many echoes you hear before they die away. Near `0` you get a single slap; toward `1` the repeats stack up into a long trailing tail. It is held just under unity inside the engine, so it thins out eventually instead of building forever:

```
hat snd 0.4 delay 3/16 delaytime 0.7 delayfeedback .
```

`delayfeedback` takes modulation, so the length of the trail can move. Here a slow `lfo` breathes it between sparse and dense:

```
rim snd 0.4 delay 1/8 delaytime 0.2 0.8 8 lfo
delayfeedback .
```

8.3.2 Four flavours

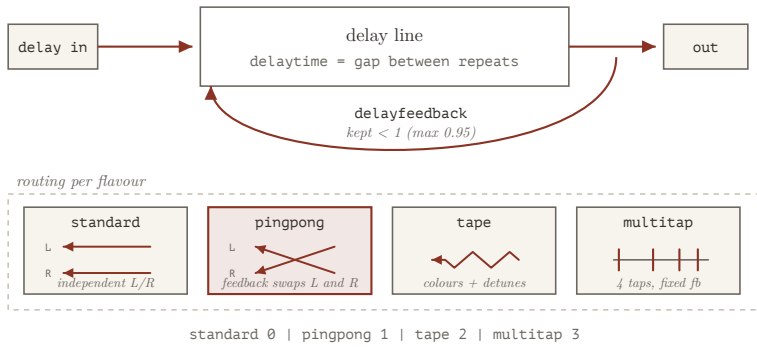


Figure 93: The four delaytype flavours share one feedback line and differ only in routing: standard, pingpong, tape, and multitap.

`delaytype` picks how the echo behaves, named as a bareword. `standard` is a plain repeating delay, the default. `pingpong` bounces each repeat between left and right. `tape` colours and detunes the repeats as a tape echo would. `multitap` lays down several taps at once for a rhythmic pattern rather than an even decay. If you are lazy, you can just use numbers to pick the algorithm: `0`, `1`, `2` and `3`:

```
hat snd 0.4 delay 1/12 delaytime pingpong delaytype .
```

WORD	RANGE	SETS
`delay`	0 to 1	send level into the delay
`delaytime`	fraction of a whole note	gap between repeats
`delayfeedback`	0 to 1	how many repeats before they fade
`delaytype`	standard, pingpong, tape, multitap	echo behaviour

For a much shorter feedback line that rings rather than echoes, see *Feedback* (8.5); for the space the echoes usually sit in, see *Reverbs* (8.2).

8.4 Comb

A feedback comb is a very short tuned delay that rings. Send any input into it, even a single burst of noise, and it behaves like a Karplus-Strong resonator, ringing at a pitch you pick and turning that input into a plucked tone. You meter the send with `comb`, tune the pitch with `combfreq`, set how long it rings with `combfeedback`, and darken the tail with `combdamp`. Excite it with a quick stab of noise and the ring is the note you hear:

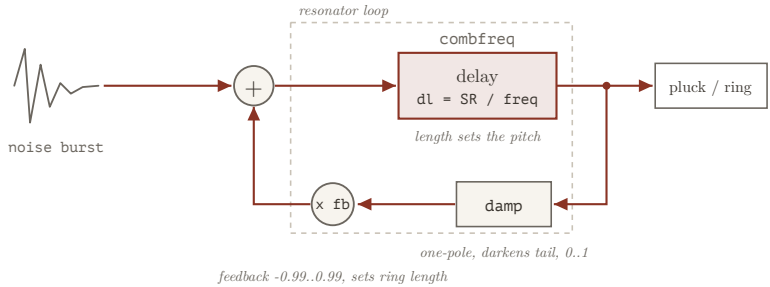


Figure 94: A noise burst enters a short delay tuned by combfreq, then feeds back through the damping one-pole and feedback multiplier before summing with the input.

```
white snd
0.001 0.08 ad
0.7 comb
[ 220 110 55 ] choose combfreq
.
```

8.4.1 A resonant pluck

`comb` is the send level, from `0` to `1`, where `0` is bypass. Turn it up and the voice pours into the resonator instead of staying dry. The comb is not an insert that lives and dies with its voice. It is a send resonator that sits on an orbit, so its ring tail keeps sounding after the exciting voice has already ended. Send several voices to the same orbit and they pile into one shared resonator.

This is what makes a noise burst pay off. The exciting voice can be gone in a couple of steps while the comb keeps singing the pitch you tuned, exactly like plucking a string and letting it ring on.

8.4.2 Tuning and feedback

`combfreq` sets the resonator's fundamental in Hz, the pitch the tail rings at. It defaults to a midrange hum, so set it where you want the note. `combfeedback` runs from `-0.99` to `0.99` and decides how long the tail rings. Near `0` the resonator barely speaks; push it toward `0.99` and the ring stretches out across many steps. Negative values ring too, with a hollower colour. Modulate `combfreq` and the pitch of the tail moves. Here a `slide` ramps it across eight steps under a long ring, a rim hit setting it off:

```
rim snd 0.7 comb 0.8 combfeedback 200 1600 8 slide
combfreq .
```

Because `combfreq` takes a value like any other parameter, an alternation turns the comb into a melody. A kick drives the resonator while the fundamental steps through three pitches:

```
kick snd 0.8 comb 0.7 combfeedback < 110 220 330 >
combfreq .
```

8.4.3 Damping

`combdamp` runs from `0` to `1` and rolls the high frequencies out of the tail as it decays. At `0` the ring stays bright and metallic; raise it and each repeat loses its top end sooner, so the pluck darkens as it fades, closer to a damped string. Modulating it keeps the tail from settling into one colour. A `jit` rolls the damping to a fresh random value every couple of steps, here exciting the comb with pink noise:

```
pink snd 0.7 comb 400 combfreq 0.8 combfeedback 0.1
0.9 2 jit combdamp .
```

WORD	RANGE	SETS
`comb`	0 to 1	send level into the resonator, 0 is bypass
`combfreq`	Hz	the resonator fundamental

word	range	sets
`combfeedback`	−0.99 to 0.99	how long the tail rings
`combdamp`	0 to 1	high-frequency damping of the tail

For how orbits carry a shared effect like this one, see *What are orbits?* (8.1); for the per-voice filters that colour a tone without ringing it into a pitch, see *Lowpass / bandpass / highpass* (7.2.1); for a resonator that rings inside the voice, with a whole bank of modes rather than one tuned delay, see *Modal resonator* (7.2.3).

8.5 Feedback

Feedback is a very short delay that re-injects its own output, sitting on the orbit like the other sends (see *What are orbits?* (8.1)). Where the delay repeats at musical intervals, the feedback line works in milliseconds, short enough that the repeats blur into resonance rather than into separate echoes. Send a voice into it with `feedback`, also `fb`, from `0` to `0.99`:

```
hat snd 0.5 fb 100 fbtime .
```

8.5.1 Time

`fbtime`, also `fbt`, sets the delay length in milliseconds, from a fraction of a millisecond up to around `680`. Short times ring at a pitch, like the comb, because the repeats are fast enough to fuse into a tone. Longer times open into a stuttering, metallic echo. It takes modulation at audio rate, so sweeping it bends the pitch of the resonance continuously:

```
rim snd 0.6 fb 40 fbtime .
```

```
rim snd 0.6 fb 5 800 4 lfo fbtime .
```

8.5.2 Damping and crossing

`fbdamp`, also `fbd`, rolls the high end out of the feedback path, from `0` to `1`, so the resonance darkens with each pass instead of staying bright. `fbcross`, also `fbc`, decides how the two channels feed

each other: at `0` each channel feeds only itself, and at `1` they cross over completely, so the repeats bounce left to right like a ping-pong:

```
hat snd 0.6 fb 30 fbtime 0.5 fbdamp 1 fbcross .
```

Because the line re-injects its own output, a high `feedback` pushes it toward self-oscillation, where it rings on its own after the input has stopped. Held just below that edge, a single hit sets off a long ringing tail.

WORD	RANGE	SETS
`feedback` / `fb`	0 to 0.99	re-injection amount
`fbtime` / `fbt`	0 to 680 ms	delay length
`fbdamp` / `fbd`	0 to 1	high-frequency damping of the path
`fbcross` / `fbc`	0 to 1	cross-channel feed, 0 self to 1 ping-pong

For longer, tempo-locked repeats, see *Delays* (8.3); for a tuned resonator on the same orbit, see *Comb* (8.4).

8.6 Compression & sidechain

`comp` is an orbit compressor, not a per-voice insert. It does not sit in one voice's chain the way a filter does; it watches a level and pulls a whole bus down whenever that level climbs past a threshold. Which level it watches is the one decision that changes what the effect is for. Left alone it watches the orbit's own output, which gives you glue: the bus tightens, the loud moments come down, and everything on it sounds like it belongs together. Point it at another orbit and the same machinery becomes a ducker, the classic pumping pulse breathing under a four-on-the-floor kick.

Either way it travels with `orbit` and its companion words (see *What are orbits?* (8.1)).

8.6.1 Gluing a bus

The plainest use needs one orbit and three words. `comp` is how much of the compression you want, from `0` for none to `1` for the full law. `compthresh`, also `cthresh`, is the level in dB where it starts

working. `compratio`, also `cratio`, is how hard it pulls once it is over that line:

```
"break" snd 1 orbit 0.8 comp -18 compthresh 4
compratio .
```

Below the threshold nothing happens at all and the bus passes untouched. Above it, a ratio of `4` lets a quarter of every extra decibel through, so a bus that was jumping around settles into something steadier. Low ratios near `2` even things out gently; high ratios near `20` stop the bus dead at the threshold and are where you go when you want the effect to be obvious.

`comp` itself is a dry/wet on the gain rather than a second threshold, so `0.5 comp` gives you exactly half the gain reduction the settings ask for. That makes it the right word to modulate when you want the compression to come and go.

8.6.2 Ducking from another orbit

Give `comporbit`, also `corbit`, the index of a different orbit and the compressor stops listening to itself. Now it watches that orbit and pulls this one down whenever it sounds. Put the trigger on one orbit and the voice you want to duck on another:

```
kick snd 0 orbit .
pad snd c3 note 1 orbit 0.9 comp -30 compthresh 12
compratio 0 comporbit .
```

The pad never hears its own level; it only responds to the kick. That means the kick can be as loud or as quiet as you like without changing how the pad reacts, and several ducked voices pointed at the same trigger orbit all pump together. A low threshold and a high ratio is what makes the duck deep enough to hear as rhythm rather than as mixing.

`comp` takes modulation, so the depth can move while it plays. A `slide` eases the pumping in from nothing over a couple of steps, a slow build under a held tone:

```
kick snd 0 orbit .
saw snd c2 note 1 orbit 0 0.9 4 slide comp -30
compthresh 12 compratio 0 comporbit .
```

8.6.3 Attack and release

Two words shape the movement itself, both in seconds. `compattack`, also `cattack`, is how fast the level clamps down when the signal crosses the threshold; a short attack snaps it, a longer one lets the leading edge through before the gain follows. `comprelease`, also `crelease`, is how slowly the level breathes back up afterwards, and a longer release is what gives a duck its long musical tail between kicks:

```
kick snd 0 orbit .
pad snd c3 note 1 orbit 0.9 comp -30 compthresh
12 compratio 0 comporbit 0.005 compattack 0.25
comprelease .
```

Short attack with a long release is the classic setting for ducking: the level bites instantly on the kick and then opens back slowly, so the pad seems to surge in the gaps. For glue you usually want the opposite instinct, a slower attack that lets transients through and a release short enough to recover before the next hit.

word	what it sets
`comp`	compression amount, 0 (off) to 1 (full)
`compthresh` / `cthresh`	threshold in dB, where compression starts
`compratio` / `cratio`	ratio above the threshold, 1 is none
`compattack`	attack of the gain movement, in seconds
`comprelease`	release of the gain movement, in seconds
`comporbit` / `corbit`	orbit to listen to; unset means this one

This is per-orbit, so it shapes one bus at a time. The limiter on the master output is a different thing entirely, always on and not addressable; see *Master output* (9.2) for what it does and how to tell when it is working. To build the buses themselves see *What are*

orbits? (8.1), and for the space most often layered under a pumping `comp` see *Reverbs* (8.2).

Chapter 9

Output

9.1 Multichannel output

Most of the time a voice lands in the ordinary stereo pair, placed with `pan` and widened with `width` or `haas` (see *Panning* (3.5)). That is all you need on headphones or two speakers. On a larger rig, a quadraphonic, octophonic, or wider ring of speakers, the engine can place a voice anywhere around that ring instead of folding everything down to two channels. How many output channels you have is set once in the engine configuration and defaults to two. Internally every voice is still stereo, and the multichannel tools decide which of the output speakers that stereo signal reaches and at what level. Speakers are counted in pairs, so a channel count of eight is four pairs, and a voice is always placed onto a pair rather than a single cone.

9.1.1 Orbits as speaker routing

The simplest way to reach a different output pair is the `orbit` parameter. An orbit is a send bus, the thing that carries the shared reverbs and delays (see *What are orbits?* (8.1)), but it also decides where a voice lands at the output. With two output channels every orbit sums into the same stereo pair, so `orbit` only groups effects. Open up more channels and the orbits spread out: orbit `0` drives the first pair, orbit `1` the second, orbit `2` the third, and so on, wrapping back round once you run out of pairs:

```
kick snd 2 orbit .
```

On an eight-channel rig that sends the kick, and everything else sharing orbit `2`, to the third pair of speakers. This is the coarse tool. It places a voice on a fixed pair and keeps it there.

9.1.2 Panning around the ring

`superpan`, also `span`, is the fine tool. It rings a voice around the whole set of output pairs with equal-power azimuth panning, the same idea as SuperCollider's `PanAz`. The value wraps from `0` to `1`, where `0` and `1` are the same point on the ring, so `0.25` is a quarter of the way round and `0.5` is the far side. It is off until you set it, and setting it switches the voice off the plain `pan` and orbit routing onto the ring:

```
saw snd c3 note 0.25 superpan .
```

Because `superpan` takes modulation like any other parameter, an `lfo` turns it into a voice that circles the room. Here the position sweeps a full rotation every four steps:

```
saw snd c3 note 0 1 4 lfo superpan .
```

`superwidth`, also `swidth`, sets how many adjacent pairs sound at once. It defaults to `2`, which keeps the voice localised to roughly one spot on the ring. Raise it and the voice smears across more pairs, until at a high enough width it comes from everywhere at once. The gains stay normalised as you widen, so the loudness holds steady.

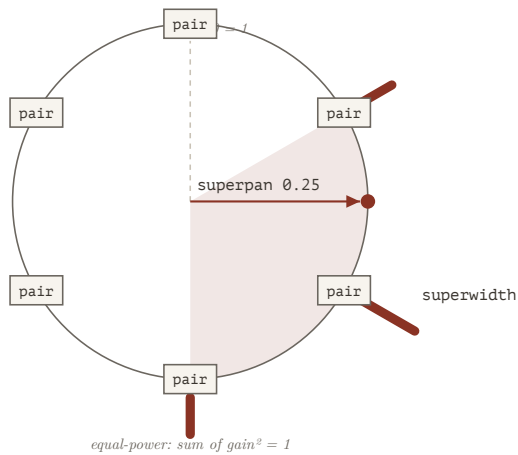


Figure 95: A top-down ring of output pairs; superpan sets the voice's azimuth and superwidth lights the neighbouring pairs with a cosine taper.

```
saw snd c3 note 0.25 superpan 4 superwidth .
```

9.1.3 Choosing the speakers

By default the ring spans every output pair. `speakers`, also `spk`, narrows it to the pairs you list, counted from `1`. It is variadic, so you push the pair numbers and the word collects them:

```
saw snd c3 note 0 1 4 lfo superpan 1 3 speakers .
```

That confines the ring to the first and third pairs, so the voice circles between just those two positions and ignores the rest of the rig. Bracketed pairs work the same way, so `[ 1 3 ] speakers` says exactly what `1 3 speakers` says. An empty list, which is the default, uses them all.

9.1.4 The master bus

Whatever the routing, every output channel passes through one final chain before it leaves the engine. Bass mono runs first if you have asked for it, then a DC blocker removes any constant offset, then the master level, then a peak limiter holds the result under the ceiling, then a soft clip catches anything still over the top. The limiter is linked across all channels, applying one gain to the whole set rather than ducking each channel on its own, so the stereo image never lurches sideways when a single speaker gets loud. The three stages you can set, bass mono, the master level and the switch that takes the limiter and its clip out of the chain, live in the Engine view rather than in the language, and *Master output* (9.2) covers all three. The rest is not addressable. It is there to keep the output safe.

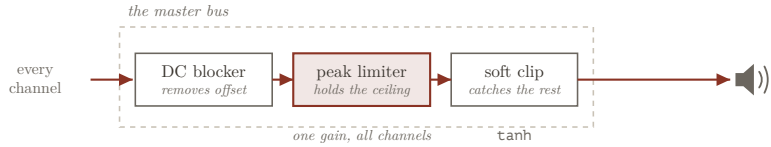


Figure 96: Every output channel ends in the master bus, a three-stage chain: a DC blocker, a linked peak limiter, then a tanh soft clip.

WORD	RANGE	SETS
`orbit`	0 to 7	output pair the voice's bus lands on
`superpan` / `span`	0 to 1, wraps	position around the ring of pairs
`superwidth` / `swidth`	1 and up	how many adjacent pairs sound at once
`speakers` / `spk`	1-based pair list	which pairs the ring spans

For placing a voice inside the ordinary stereo pair, see *Panning* (3.5); for what orbits carry besides this routing, see *What are orbits?* (8.1).

9.2 Master output

Everything Cagire makes arrives at one place. The orbits have been mixed, the room has been spread, and what is left is a block of output channels about to leave for the sound card. Three of the stages waiting there are yours to set, and they exist for the same reason: playing on a system you did not build.

They are not words. You will not find `mastergain` in the dictionary, and that is deliberate. A master level belongs to the room you are standing in, not to the pattern you are writing, and a mistyped number in a live-coded line should never be able to reach a PA. All three live in the Engine view, on the `AUDIO` tab, and all three are saved with your other audio settings rather than in the project file. Carry a project to another venue and you carry the music, not the last room's level.

9.2.1 Setting the level

`Master` is the output level in dB, and it is the first thing to reach for when the house engineer asks you for more or less. It runs from `OFF` at the bottom to `+6.0 dB` at the top, in half-decibel steps, and it takes hold immediately: Cagire ramps the move across the next block, so you can ride it during a set without hearing the step.

The level lands *before* the limiter, which is the part worth knowing. Turning yourself up drives the safety chain harder rather

than sneaking past it, so a boost can never produce something the limiter has not seen. If you push far enough, you will watch the limiter start working rather than watch the output distort.

9.2.2 Watching the limiter

The limiter is on by default and never announces itself, so a mix that sounds flat and lifeless looks exactly like one that is simply loud. Cagire shows you the difference in two places, one for playing and one for diagnosing.

While you play, the stereo level meter down the right edge of the Sequencer view lights a bar across the top of both channels whenever the limiter is working. It holds for a second and a half after the last time it engaged, so a brief flash still registers, and it lights both channels together because the limiter applies one linked gain to the whole output rather than ducking each channel on its own. It is deliberately something you can read without looking away from the pattern you are editing.

When you want the number, the `STATUS` column on the Engine view's `AUDIO` tab carries a `Limiter` readout next to the DSP load, showing how much gain reduction was applied in the last block. At `0.0dB` the limiter is idle and the level you set is the level you get. As the number goes negative the ceiling is holding your mix down, and past about 6 dB the bar turns red.

If either one is buried, the fix is to come down, not to push further.

The meter's scale ends at the engine's real ceiling, so a full bar means genuinely full output and there is no unreachable strip at the top pretending to be headroom.

9.2.3 Taking the limiter out

`Limiter` has two states, `On` and `Bypass`, and either arrow flips it. Bypass is a real bypass. Cagire skips the whole stage, so nothing is left holding the mix down, nothing delays the master by the lookahead, and the soft clip that backs the limiter up goes with it. Whatever the mix peaks at is what leaves for the sound card, past full scale included.

That last part is the reason to think before you reach for it. On the way out, samples above full scale are the converter's problem, and the noise it makes is worse than anything the limiter was doing. Bypass is for a rig where something downstream already owns the ceiling: a desk, an outboard limiter, a mastering chain you are recording into. It is not a way to get louder.

Switching either way clicks. The lookahead appears or disappears and the output steps by that much, which is a millisecond and a half of the master jumping. Set it between tracks, not mid-drop.

While the limiter is bypassed both readouts change what they tell you. The `Limiter` bar reads `Bypass` instead of a gain reduction, since there is none to report, and the level meter's indicator now lights on clipping, the same thing it means on the sampler's input meter. A lit bar there is no longer the ceiling working. It is the output leaving the building too hot.

9.2.4 Keeping the low end mono

`Bass mono` collapses everything below a crossover corner to the centre. It is off by default, steps in 10 Hz increments, and most rooms want it somewhere near 100 or 120 Hz.

The reason is physical rather than musical. A club sums its subwoofers, and stereo content down there arrives at them out of phase and cancels, so a bass line that sounds enormous on headphones can half disappear on a big rig. Anything that decorrelates the low end causes it: `width`, `haas`, `superpan`, the chorus, a hard-panned kick. Turning bass mono on removes the difference between the channels below the corner and leaves everything above it exactly as it was.

It costs you nothing in the mix a mono system hears. The stage discards only the low-band difference, so the sum of the channels is untouched: what changes is that the two channels now agree down there instead of fighting.

9.2.5 Moving around

Key	What it does
`Tab`	Cycle to the `AUDIO` tab

Key	What it does
`Up` / `Down`	Move between settings
`Left` / `Right`	Change the focused setting

One more stage sits ahead of these, on its own `DYNAMICS` tab: a three-band compressor that squeezes the mix from both ends at once. It is off by default and it is the loudest thing you can do to a finished mix, so it has a page of its own. See *Master dynamics* (9.3).

The rest of the chain, the DC blocker ahead of the limiter, is fixed. See *Multichannel output* (9.1) for the full order and for how voices reach these channels in the first place, and *Compression & sidechain* (8.6) for the per-orbit compressor, which is a different tool for a different job.

9.3 Master dynamics

Cagire carries a three-band compressor on the master bus that squeezes your mix from both ends at once. It is the loudest, most obvious thing in the whole signal path: dense, forward, every detail pushed to the front, reverb tails sitting as loud as the transients that caused them. It has its own `DYNAMICS` tab in the Engine view, and it runs first in the master chain, ahead of the level and the limiter.

Like the master level, these are not words. You will not find them in the dictionary. They belong to the rig rather than to the pattern, they are saved with your audio settings, and a mistyped number in a live-coded line should never be able to reach a PA.

9.3.1 Both directions at once

An ordinary compressor is a ceiling. Anything above the threshold gets pushed down, and everything below is left alone. This one does that, and then does the opposite as well: anything *below* the threshold gets pulled *up*. The dynamic range collapses from both sides toward the middle instead of only from the top.

Then it does that three times, on a low band below roughly 88 Hz, a middle band, and a high band above roughly 2.5 kHz, each

with its own envelope. The low band reacts slowly because low frequencies take time to establish a level, and the high band reacts in well under a millisecond. A kick drum can be held down without the hi-hats being held down with it.

The thresholds sit very low, around 40 dB below full scale, and that is what makes this different from ordinary multiband compression. Almost nothing in a real mix is quiet enough to fall outside the working range, so almost everything is being pushed or pulled somewhere at all times. That is why it does not sound like compression. It sounds like the mix has been inflated.

The upward half is the half you notice. Room tone, reverb tails, the ring after a snare and the noise floor of a sample all get lifted toward the level of the loud material. Cagire caps that lift at 24 dB so a silent passage cannot be dragged up to become the loudest thing in the track, but within that ceiling it is aggressive, and it will find whatever your samples have hiding in them.

9.3.2 Amount

`Amount` runs from `off` to `100`, and its two halves do different things.

From `off` to `50` the processed signal is blended in *alongside* your untouched mix, which is parallel compression. Nothing is taken away. The density arrives on top of what you already had, and the mix gets louder as you turn it up. At `50` you are hearing a full dry mix and a full wet one together, which is the loudest setting and the one most worth living at.

From `50` to `100` the dry mix fades out and the processed one takes over. At `100` you hear nothing but the compressor. That end is level-matched rather than quiet: Cagire applies a fixed makeup gain sized so a normal program level comes back out at unity, so turning it all the way up changes the character without gutting the volume.

At `off` the stage is genuinely gone. The band-splitting filters are not merely turned down, they never see the signal at all, so a compressor you are not using cannot colour a mix you are.

Learn it at `100` before you learn it anywhere else. At `20` you are hearing a fifth of the effect against a full dry mix, which is not enough to work out what any of the other four controls are doing.

9.3.3 Drive

`Drive` sets the level going in, from `-12` to `+24 dB`, and it is the real intensity control rather than `Amount`.

The thresholds are fixed and very low, so this is what decides how far past them your material actually lands. Turning it up does not mainly make the output louder, it makes the compression bite deeper. Past a certain point more drive buys you almost no extra level and a great deal more squash.

9.3.4 Sat

`Sat` runs from `off` to `100` and adds saturation to each band separately, after that band's gain and before the three are summed back together.

Doing it per band rather than on the finished mix is the whole point. The three bands carry very different amounts of energy, so one saturator across the lot either barely touches the top or turns the bottom to mud. Separately, Cagire can drive the low band hard for weight while leaving the high band nearly alone, which is what stops the result sounding brittle.

The curve is deliberately asymmetric, so it makes even harmonics as well as odd. Odd harmonics alone sound hard and squared off; the even ones are the warm half, and they are most of what people mean when they call saturation musical. At full `Sat` the low band picks up a second harmonic about 19 dB below the fundamental, which is a lot. Halfway up it is nearer 32 dB down, which is the range most material wants.

It is level-compensated, so turning it up trades character rather than volume. Across the whole range the output moves by about a fifth of a decibel. If it sounds louder, that is the harmonics, not the gain.

Two things worth knowing. It is antialiased, using a curve that cancels most of the harmonics that would otherwise fold back

down the spectrum as inharmonic grit, so it does not need the oversampling that usually goes with this. And it sits downstream of the upward compression, so it drives whatever the compressor has already lifted. At sensible settings that is exactly what you want, because everything gets driven consistently instead of only the loud parts. Pushed to extremes it will also be saturating a lifted noise floor.

9.3.5 Time

`Time` scales every band's attack and release together, from `10%` to `1000%`, with `100%` the setting the rest of the design assumes.

Short values make it chatter. The envelopes track individual cycles of the low band, the gain moves within each drum hit rather than across it, and you get distortion and a hard, gritty edge. That is a usable sound, not a mistake, but it is a sound rather than a mix tool.

Long values turn it into slow breathing. A loud passage pulls everything down and it takes most of a second to come back, so the track pumps against its own dynamics. Somewhere near `100%` is where it stops being an effect and starts being glue.

9.3.6 Color

`Color` runs from `-50` to `+50` and is the only tone control here. It leans the crossover points and the low and high band gains together in one move, so positive values lean bright and negative values lean heavy.

It is not an EQ after the fact. Moving the crossovers changes which material lands in which band and therefore which envelope is deciding its level, so `Color` changes what the compressor is reacting to as much as what comes out. A bass-leaning setting hands more of the mix to the slow low-band envelope and sounds calmer. A treble-leaning setting hands more of it to the fast high-band envelope and sounds sharper.

At `0` the three bands come out matched, within a fraction of a decibel.

9.3.7 Trim

`Trim` sets the level coming out, from `-24` to `+12 dB`. Use it to match what you had before you turned any of this on, so you are judging the character rather than the volume.

9.3.8 Finding a setting

Set `Amount` to `100` with everything else at its default, so you can hear what the thing actually does. Bring `Drive` up until it is as aggressive as you want it. Add `Sat` until the result has some weight to it. Set `Time` for character, low for grit and high for pumping. Lean `Color` for tone. Pull `Trim` back until it matches what you started with. Then walk `Amount` down to taste, which for most material lands somewhere between `30` and `60`.

9.3.9 Moving around

Key	What it does
`Tab`	Cycle to the `DYNAMICS` tab
`Up` / `Down`	Move between rows
`Left` / `Right`	Change the focused row
`Shift` + `Left` / `Right`	Change it in larger steps

9.3.10 Where it sits

This runs before the master level, which runs before the limiter. That order matters: whatever it adds still has to pass the safety chain, so you cannot use it to sneak past the ceiling. If you push it hard you will see the `Limiter` readout on the `AUDIO` tab start working, and the answer then is to come down on `Trim`, not to push further. See *Master output* (9.2) for the level and the limiter, and *Compression & sidechain* (8.6) for the per-orbit compressor, which is a much gentler tool for a completely different job.

9.4 Monitoring orbits

The master meter tells you what the sum is doing. It cannot tell you which orbit made it.

The orbit meters fill that gap. Cagire has eight orbits and they all arrive at the same place. A mix that has gone wrong at the master looks identical whether one bus is far too loud or all eight are slightly too loud. On a stereo card Cagire sums the orbits into the same pair before anything downstream can tell them apart, so staring at the output will never separate them. Cagire has to measure inside the engine, on each orbit's own audio, and that is what these meters show.

9.4.1 Reading the meters

The meters are a visualizer, alongside the scope, the spectrum and the phase scope, so they live wherever you have put your visualizer panels on the Sequencer view. They are on by default. There is one bar per orbit, in orbit order from `0` to `7`. They turn with everything else when you change the layout: a bar per row when the panels are wide, a column per orbit when they are tall. To keep the columns whatever the layout is doing, set `Orbit orientation` in Options. Double-clicking the panel does the same thing without leaving the Sequencer view. See *Visualizers* (2.9) for the panels themselves and how to arrange them.

Cagire also falls back to columns on its own when a panel is too short to give every orbit a row. Eight buses do not fit in five rows, and dropping the last three would put you right back where you started.

A bar shows the loudest sample that orbit produced since the last frame. A marker rides above it and holds the recent peak for a second and a half before it falls. The number is that marker's value rather than the bar's, because a figure redrawn every frame is a blur you cannot read. The bar is what is happening now. The number and the marker are what just happened. Reading the peak rather than an average is deliberate. A short loud transient is the thing that makes a mix feel wrong without ever looking wrong, and an averaging meter hides it.

The scale runs from `-48 dB` to `+3 dB`, the same scale the Engine view uses for its own readouts. A level means the same thing wherever you read it. Unity sits near the top of the bar rather

than at the end of it, and the strip above unity is real rather than decorative. Cagire measures an orbit before the master chain, so an orbit can run past full scale and you need to see how far.

When it does, the reading turns to the warning colour and stays there for a second and a half, so a single transient over the top still registers. It is a different colour from the top of the bar on purpose. The bar is already red near full scale, and a hot but perfectly fine orbit should not look like a problem.

9.4.2 What is being measured

The reading is that orbit's bus after its effects and before its compressor.

The order matters if you use `comp`. Cagire applies an orbit compressor when it mixes the orbits together, which is after this measurement, so a heavily compressed orbit reads hotter here than it sounds. That is the useful way round. You are looking at what the voices send to the bus, which is the thing you can actually change. If the bar is buried and the orbit still sounds quiet, the compressor is doing more than you thought.

A room-routed orbit is the one special case. Those orbits send only their wet signal to the room, and their dry never leaves. Metering the whole bus would light a bar for audio nobody hears. Cagire meters the wet for them instead, the same signal `orec` would capture. See *Multichannel output* (9.1) for what puts an orbit in the room in the first place.

9.4.3 What to do about it

There is no orbit fader, and there is not going to be one. An orbit is a send bus, not a mixer channel, and the level on it is the sum of what the voices put there.

So the meters are a diagnosis, and the cure is upstream. If one bus is swamping the others, come down on the voices feeding it with `gain` or `postgain`, or pull back the send that feeds the effect. If everything is hot together, the fix is at the master, and *Master output* (9.2) is the page for that. A quiet orbit you expected to hear is usually a routing mistake rather than a level one. Check that the

voices you meant to send there are carrying the `orbit` you think they are.

9.4.4 Moving around

Key	What it does
`F3`	Open Options, where `Show orbit levels` turns the panel on and off
`o`	Cycle the visualizer layout on the Sequencer view
Double-click	Flip the meters between rows and columns

See *What are orbits?* (8.1) for what an orbit is and how voices reach one, and *Compression & sidechain* (8.6) for the compressor this measurement sits ahead of.

Part V

Audio patching



Chapter 1

A second language

Up to this point, we have only talked about the main language that you can use in Cagire: the sequencing language. This language is the one you will use 90% of the time. Its purpose is to define sound and events, set parameters, and help you make music. It is wonderful for arranging sound you already have: oscillators, drums, audio samples, live sources, and so on. What it never touches is the core process inside an instrument, the synthesis or sound processing itself. Curly braces give you access to a second language that does exactly that, a language called Arf. Where the sequencing language decides which sound plays and when, Arf describes the sound itself, sample by sample. If you feel constrained by the palette of sound sources and effects already available, this is how you build new ones from scratch.

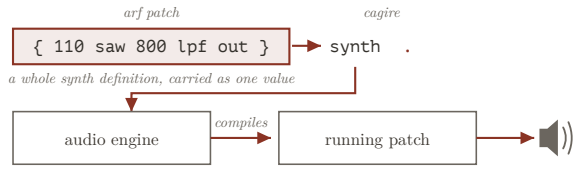


Figure 97: Cagire runs the script as sequencing language, but captures everything between the braces as one synth definition that becomes the voice's source.

Everything between the braces forms one audio patch. The patch travels through your script as a single value, from the moment you open the braces to the word that consumes them:

```
{
  110 saw
  800 lpf
  out
} synth
.
```

Read the content inside the braces just like you would read any Cagire script. `110` goes on the stack. `saw` turns it into a band-limited sawtooth at 110 Hz. `800 lpf` rolls off everything above 800 Hz, and `out` marks the result as the sound this voice will play. Outside the braces, the sequencing language takes over again. `synth` hands the captured text to the audio engine, which compiles it and makes the result the sound of this event. The trailing `.` fires the event like any other. You have just used two languages in a single script. Cagire never reads anything between `{` and `}` as sequencing language. The braces capture their contents verbatim and hand them to the patch compiler, which has its own vocabulary, syntax, and semantics. The two languages share many spellings, so you always want to know which one is reading at any given point. *Inside and outside the braces* (2.4) is the section that helps you do exactly that.

1.1 What this part covers

The *Basics* (2) come first, and they are about fitting a patch into the language and world you already know. *Inline synths* (2.1) shows how `synth` compiles a `{ ... }` patch on the spot and makes it the sound of the event you are building. You can build an instrument in a single step and hear each edit as you make it. *Named patches* (2.2) adds `defsynth`, which installs a patch under a name the rest of your project can play. *The voice contract* (2.3) is the one to read slowly. It explains what the engine feeds a running patch, the `notefreq`, `gate`, and `vel` that arrive from the sequencer. It also explains how the voice's own envelope, gain, and orbit effects wrap whatever the patch sends to `out`. *Inside and outside the braces* (2.4) untangles the shared spellings, so the `lpf` you write inside a patch never gets

confused with the `lpf` you write outside it. And when a patch should process sound rather than make it, *Patches as effects* (2.5) shows how to install it on an orbit with `dsp` or inside a voice with `fx`.

After the *Basics* (2), the *Language tour* (3) teaches Arf itself one idea at a time. It starts with *Oscillators & noise* (3.1) and works forward through filters, saturation, modulation, naming, feedback, and stereo. It then turns the same vocabulary around to build effects, and finishes on three complete instruments in *Putting it together* (3.9). Read it in order the first time, because each page assumes the one before it. Every page points to a matching page in the Reference where every word is listed with its signature and a compile-checked example. Every Arf word also lives in the F1 Dictionary under the Arf tab. Cagire generates that list from the engine's own tables, so it always matches what the compiler will accept. Reach for Arf when the built-in sound palette is no longer enough for you, but not before you have mastered the sequencing language!

Chapter 2

Basics

2.1 Inline synths

An inline patch is a synthesizer you build in place. You write the instrument between the braces and fire it in the same script. This synthesizer has no name. `synth` is the word that connects the two ends: it takes the `{ ... }` block and hands the captured text to the audio engine, which compiles it and sets the result as the sound of the event you are building. From there the patch behaves like any built-in source. It works with everything else in your script: parameters, orbits and insert effects, polyphony words, and the rest. Use an inline patch to build a one-off instrument in a single step and hear each edit the moment you make it.

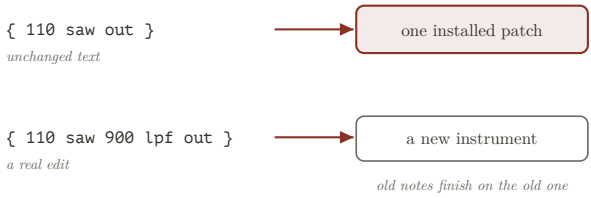


Figure 98: Cagire derives a patch's identity from its text. Unchanged text reuses the same instrument, any real edit compiles a fresh one.

Here is an instrument and the note that plays it in a single script:

```
{
  110 saw
  800 lpf
  out
} synth
.5 verb
.
```

That script builds a band-limited sawtooth at 110 Hz, rolls the top off above 800 Hz with `lpf`, and marks the result with `out`. `synth` installs the patch, `.5 verb` wraps it in a little reverb, and `.` fires the note. The indentation you see here is purely visual: it separates the audio patch from the surrounding sequencing language and has no effect on how the code runs.

You never name an inline patch. Cagire derives its identity from the text between the braces. Whitespace does not count. Re-indenting or reflowing a patch changes nothing. Any real edit makes it a *different instrument*, compiled fresh on the next step. Unchanged text is free: your script re-evaluates every cycle, and an already-installed patch is recognized and skipped. This is what makes live iteration comfortable: tweak a number inside the braces, re-evaluate, and the next note plays the new sound while old notes finish on the old one. One corollary: two steps that carry the identical text play one installed patch, not two. That is usually what you want, and it also means they share the same pool.

2.1.1 Audio patch parameters

A patch can declare its own controls with `param`: `param cutoff 400` names one inside the braces and gives it a starting value, and `3000 "cutoff" param` sets it from the sequencing language. Setting a param from outside leaves the braces untouched, so the instrument responds without becoming a different patch. Declaring and setting patch params is covered in *The voice contract* (2.3).

2.1.2 Two limits worth knowing

Two limits are worth knowing when you build audio patches:

- every patch has to route a signal to `out`. You should also declare whether this signal is mono or stereo.
- every patch has a voice pool. Each distinct patch can spawn as many instances at once as the Voices setting in the Engine view allows, sixty-four by default. If you overshoot that limit, nothing will sound at all: the event will be silently dropped.

Long-releasing patches can reach the voice pool limit sooner than you would expect, because each voice stays alive for as long as its release lasts. Shorten the release, or split the sound into two variants with `defsynth`. The pool is sized when a patch is installed, so raising Voices reaches a patch the next time you re-evaluate it.

2.1.3 What Cagire keeps, and what it lets go

An inline patch lives as long as your script keeps mentioning it. Cagire keeps the sixty-four most recently used ones and quietly drops the coldest beyond that, which is what stops an afternoon of editing from filling memory with every intermediate version of a sound. The one you are working on is safe: a playing pattern re-evaluates its text every cycle, and that counts as using it. Named patches are never dropped, because nothing tells Cagire when you are done with a name (see *Named patches* (2.2)).

2.2 Named patches

An inline `synth` never asks for a name because it never needs one. It lives on the step that plays it and is gone by the next. Naming a patch is how you keep a synthesizer around for later, the way you would save a preset. `defsynth` installs a patch under a name you choose, and from that moment the rest of your project can use it like any built-in sound. You just call the name on any step, in any pattern. A named patch is an instrument, in other words, and the name is how the project refers to it.

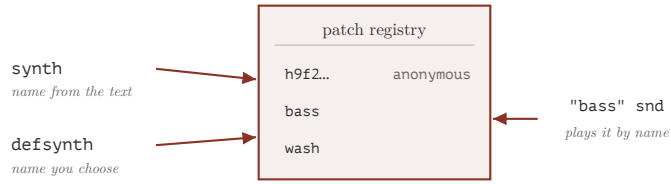


Figure 99: `synth` *and* `defsynth` *install into one patch registry;* `synth` *names the patch from its text,* `defsynth` *from a name you choose.*

`defsynth` installs a patch under a name of your choosing and emits nothing. It is a definition, not a sound:

```
{
  notefreq saw
  600 lpf
  out
} "bass" defsynth
```

Cagire captures and compiles the braces exactly as `synth` would. But instead of becoming this event's sound, the patch is stored under `bass`, ready for any step to use. From then on it is a sound like any other, played by its bare name:

```
"bass" snd
a2 note
.
```

That plays `bass` exactly as `saw snd` plays a sawtooth. It takes `a2 note` and every other sequencing word you write around it, because an installed patch is simply another source name the engine knows.

2.2.1 Where definitions live

A `defsynth` in a step script re-installs on every firing. Unchanged text is recognized and skipped, so this costs nothing. More often an instrument definition lives in the *Bank prelude* (4.2) or the *Global prelude* (4.1), where it installs once when the bank loads. Every pattern in the bank then plays `"bass" snd` without carrying the definition around. To live-edit an instrument under a running

pattern, re-evaluate its `defsynth`. With a pattern looping `"bass"`, change `600 lpf` to `1200 lpf` and re-evaluate. Voices already sounding finish on the old patch, and the very next note plays the new one. No restart, no pattern edit. Give the two-oscillator patch a name of its own and play it back the same way:

```
{
  [ 110 saw 111 saw ]
  mix
  out
} "wide" defsynth
"wide" snd
.
```

That installs the patch under `wide` and plays it back by that same name, the same round trip as `bass`, only wider.

2.2.2 Choosing a name

A patch name must be a single word. It may hold no whitespace, no `/`, and no `:`. Everything else is allowed. A name can carry digits, hyphens, or underscores to tell one variant from another, so `"bass2"`, `"kick_sub"`, and `"weird-one"` all install fine. One name is spoken for. `off` is reserved, because `"off" dsp` and `"off" fx` are how you clear an effect slot (*Patches as effects* (2.5)).

The names worth avoiding are the ones already taken. When a script names a sound, Cagire looks for a built-in source first, then an installed patch, then a sample folder. A patch called `saw` therefore never plays, because `saw` resolves to the oscillator before the search reaches your instrument. Nothing warns you either. You simply hear the built-in. In the other direction the order works for you. A patch beats a same-named sample folder, which is how you replace a sampled instrument with a synthesised one without renaming anything that plays it.

Names also do not expire. Cagire reclaims an inline patch once your script stops mentioning it. A named one stays installed for the session, because a name is a promise that some pattern may still call on it. Redefining a name replaces what it holds, so iterate by re-evaluating rather than by minting `bass2`, `bass3`, `bass4`.

2.2.3 Named or inline?

The two words install into the same registry and build the same kind of patch. The choice between them is really a choice about names, not about what a patch can do. Inline `synth` is for the sound you are working on right now, the one you want to hear change as you type. It lives on the step that plays it. Cagire derives its identity from the text between the braces, and it needs no name because nothing else refers to it (see *Inline synths* (2.1)). `defsynth` is for the instrument the rest of the project plays. The moment two patterns need one sound, or you want to edit an instrument while patterns keep playing it, the name is what lets them all reach the same patch. Either way the result is an ordinary voice source. The only difference between the two is who picks the name: you, or the text itself.

word	what it does
`synth`	Set a `{ ... }` patch as this event's sound
`defsynth`	Install a `{ ... }` patch under a name; emits nothing
`"<name>" snd`	Play an installed patch as a sound

2.3 The voice contract

A patch never plays alone. Cagire makes it the *source* of a doux voice (see *What is a doux voice?* (3.1)), and the voice keeps all of its usual machinery. Knowing what flows into the patch and what wraps around its output will help you write patches that sit naturally in a pattern.

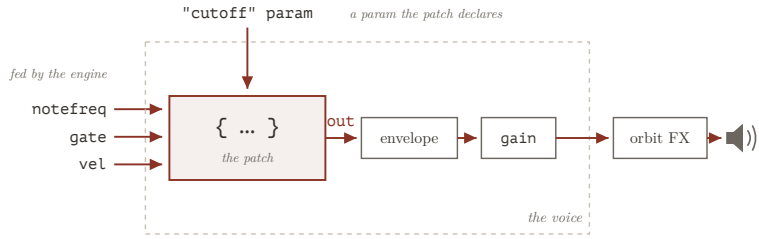


Figure 100: The engine feeds `notefreq`, `gate`, *and* `vel` *in. The pattern drives declared* `params`, *and the voice wraps out with its envelope, gain, and orbit effects.*

2.3.1 What the engine feeds the patch

Three values are written into every running patch, readable by name:

- `notefreq`: the event's frequency, from `note/freq` and everything that modulates them. It is updated *every sample*, so glides and vibrato from the sequencing side pass straight through.
- `gate`: 1 while the note is held, 0 once the voice enters its release.
- `vel`: the event's velocity, 0 to 1.

A patch that hardcodes `110 saw` plays 110 Hz whatever you sequence. Swap the constant for `notefreq` and the patch follows the notes you write:

```
{
  notefreq saw
  600 lpf
  out
} synth
< a1 c2 e2 > note
.
```

`gate` lets a patch shape its own articulation on top: `notefreq sine gate *` mutes the body the instant the note releases, while the voice's envelope fades what remains.

2.3.2 Params you declare

The engine only feeds pitch, gate, and velocity, plus `bps`, the transport tempo in beats per second, and `bus`, the engine's control

bus lanes (*Core words* (4.1)). For everything else a patch declares its own parameters: `param name default` names a control the pattern can set per note, and a later bare `name` reads it. The default is part of the declaration, not optional, so `param cutoff 400` is the shortest form there is. Until an event writes one, the control holds that default:

```
{
  param cutoff 400
  notefreq saw
  cutoff lpf
  0.3 *
  out
} synth
< a1 c2 e2 > note
.
```

The same word sets it from outside the braces, value first. The name is a string (a bareword works when no Cagire word shadows it):

```
{
  param cutoff 400
  notefreq saw
  cutoff lpf
  0.3 *
  out
} synth
< a1 c2 e2 > note
3000 "cutoff" param .
```

Every note starts from the declared defaults. Delete the `3000 "cutoff" param` and the next notes are dull again. The script states what it wants on every event, so what you read is what sounds. A sourceless update of a held voice is the one exception: it writes only what it names, so nudging a drone never resets its other params. Patch params also take modulation. Anything that builds a motion drives a patch param exactly as it drives `lpf`, per sample:

```
{
  param cutoff 400
  notefreq saw
  cutoff lpf
  0.3 *
  out
} synth
< a1 c2 e2 > note
200 4000 2 lfo "cutoff" param
.
```

A patch's `cutoff` is its own name, not the voice's filter: patch params live in their own namespace, so declare whatever reads well. A patch may declare up to sixteen, each exactly once (*Limits* (4.10)).

2.3.3 What wraps the output

The signal a patch sends to `out` runs through the voice's normal chain: the envelope words (`attack`, `release`, …) shape it, `gain` scales it, and the orbit's effects (`verb`, `delay`, …) apply. A patch is a source, not a bypass. Keep patch amplitudes modest (`0.2 *` is a good habit); the engine also gives sources a little headroom and silences non-finite samples rather than letting them poison the mix. A patch has to reach `out`, and `out` is also where it declares its width. One `out` is a mono voice; a two-channel list is stereo. That is the only way a patch sets its channel count:

```
{
  [ 110 saw 111 saw ]
  out
} synth
.
```

2.3.4 Retriggering a held voice

Retriggering a *held* voice does not dip `gate`, so envelopes inside the patch do not re-fire; the voice's own envelope re-articulates regardless. If a patch-internal envelope is the sound, give each note its own voice instead of retriggering one. What the engine does not feed a patch, and what it refuses to install, is listed in *Limits* (4.10).

2.4 Inside and outside the braces

Cagire hosts two languages. Both are Forth-like, so if you know how to write Cagire code, you already know how to write Arf code. Each language keeps its own dictionary. The sequencing language runs the whole script, naming sounds and setting their parameters. Arf's only role is to let you write your own digital signal processing code. A large number of words appear in both dictionaries with different meanings, and a few appear in only one. The braces are what decide: a word read outside `{ }` is a sequencing word, and the very same spelling read inside is an Arf word. Nothing but the position tells them apart. Because so many words overlap, you have to keep the two dictionaries clearly separated in your head. The rest of this page lays the shared spellings side by side to help you do that.

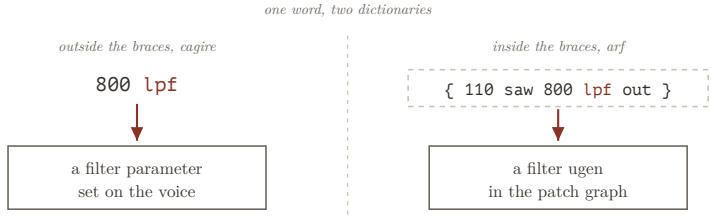


Figure 101: The braces choose the dictionary: outside them lpf sets a filter param on the voice, inside it is a filter ugen in the graph.

The clearest case is a word that is real in both dictionaries and does something genuinely different in each:

```
;; Cagire: set the voice's filter param
"kick" snd
800 lpf
.

;; Arf: build a filtered saw from scratch
{
  110 saw
  800 lpf
  out
} synth
.
```

The first script never leaves the sequencing language. It plays a `kick` sample and reads `800 lpf` as a lowpass cutoff set on that voice. The second hands everything between the braces to Arf. There `lpf` is no longer a parameter but a filter node wired into the graph, filtering a `saw` the patch builds for itself. The same three letters do two different jobs, and only the braces tell Cagire which one you meant.

2.4.1 The shared spellings

`lpf` is not a special case, it is the pattern. More than sixty spellings live in both dictionaries, and they fall into three groups. Most mean the same thing on both sides, so you can forget about them. A second group names genuinely different machinery, which surprises you once. The third group looks identical and is not. That is the group to learn.

Start with the plainly different ones. Outside the braces each of these sets a level or a time on a voice or an orbit. Inside, the same spelling is the machine itself, with inputs of its own:

Spelling	Outside the braces	Inside the braces
`delay`	The orbit's delay send	A delay line, `( in time -- sig )`
`verb`	The orbit's reverb send	A whole reverb, `( in mix room damp -- sig )`

spelling	outside the braces	inside the braces
`comb`	The orbit's comb send	A feedback comb, `( in time fb -- sig )`
`fb`	The orbit's feedback send	Declares a feedback latch's width
`comp`	The sidechain duck amount	A compressor, `( in thresh ratio attack release -- sig )`
`crush fold wrap distort`	Lo-fi amounts on the voice	Shapers, each taking a signal and its bounds
`decay`	The envelope's decay time	An exponential decay, `( in time -- sig )`
`adsr`	Sets the four envelope times at once	An envelope generator reading a gate
`noise`	The tom's stick-noise amount	White noise, the source
`gate`	Sets how long the note lasts	Reads whether the note is still held

None of those will fool you for long. The patch word always eats a signal along with its numbers, and the sequencing word takes the number alone. `0.4 verb` is a send level. `in 0.5 0.8 0.4 verb` is a room.

The near-misses are where you get caught. These five spell the same, do nearly the same job, and disagree about something you cannot see from the call site:

spelling	outside the braces	inside the braces
`range`	Maps `0`..`1` onto `[min, max]`	Maps `-1`..`1` onto `[lo, hi]`
`coin`	A 50/50 boolean, taking nothing	Passes a trigger with probability `p`, `( trig prob -- sig )`

SPELLING	OUTSIDE THE BRACES	INSIDE THE BRACES
`slew`	Glides to a target over a duration	Bounds how fast a signal may change, `( in up down -- sig )`
`times`	Runs a quotation `n` times as the script executes	Stamps a quotation into the graph `n` times, once, at compile
`select`	Runs the `n`th of several quotations	Keeps the `n`th channel of a list

`range` is the one to watch, because it fails quietly rather than loudly. Hand it a bare oscillator inside the braces and it behaves, because oscillators swing both ways. Hand it a `phasor`, a `perc`, or anything else that only ever goes positive, and you silently lose the bottom half of your span. Put `bi` in front of it, or use `linlin` and name both ends yourself.

Everything else agrees. The arithmetic (`+`, `*`, `min`, `mtof`, `floor`), the stack shuffles (`dup`, `swap`, `over`, `rot`), and the comparisons mean on both sides what they mean anywhere. `!` rewards a second look. Outside the braces it copies a value up the stack `n` times. Inside, it copies a signal across `n` channels, which is one idea in two mediums. And `rand`, `exprand`, and `logrand` differ only in when they roll, once per event outside and once per note inside (*Noise & sources* (4.3)).

Some grammar is deliberately shared, meaning the same thing on both sides rather than colliding by accident. A definition, `: name … ;`, an argument head block, `{ a b }` right after the name, and `:>` naming an intermediate all read identically inside and outside the braces (see *Defining words* (3.11.1) and *Names, locals & reuse* (3.5)). `param` goes further. Its two halves, the `param name default` declaration inside a patch and the `value name param` write outside it, form one feature rather than a coincidence of spelling. *The voice contract* (2.3) explains that pairing in full.

One principle keeps the shared words straight: declarations read forward, computations read backward. A declaration states its name before its body or value, so `: name`, `param name default`, and `buf name secs` all read left to right. A computation takes its operands

off the stack behind it, so a filter, an operator, or an oscillator reads right to left. This split holds in both languages.

The editor keeps you oriented while you write: brace regions render in their own color, autocompletion switches to the patch vocabulary the moment the cursor slips inside, and the F1 Dictionary lists every patch word under the Arf tab.

2.4.2 Comments inside and out

Comments follow the same rule. Outside the braces you have three comment forms: `;;` for the rest of a line, `;* ... *;` for a block, and a lone `;` that closes a `:` definition. Inside the braces Arf keeps only the first of them, `;;`, because it has no block comment of its own. Its lone `;` does one job only: closing an Arf `: ... ;` definition. Write a `;* *;` inside a patch and the compiler never sees a comment at all. It reads two unknown words and stops before a note sounds.

2.4.3 Patch words outside braces

This only works one way. A patch word typed outside `{ }` never reaches Arf, because the sequencing language does not know that the patch compiler exists. Short patch words like `out`, `saw`, and `sine` quietly turn into strings, the same fallback that lets a bare sample name work. Longer ones may trip the typo guard and hand you a suggestion for some unrelated Cagire word. Neither outcome is an error, so nothing warns you and the note simply comes out wrong or silent. If a patch behaves strangely, the first thing to check is that the braces close exactly where you think they do. Once you keep the two dictionaries apart, you can write patches without second-guessing every word. The last page of the Basics, *Patches as effects* (2.5), shows how a patch can process sound instead of making it.

2.5 Patches as effects

Everything so far has built a patch that makes a sound out of nothing, an oscillator or noise source that the voice simply plays. A

patch can just as easily work the other way round, taking incoming sound and reshaping it. One thing flips it into that role: reading audio input. A small family of words does that. `in` reads channel 0. `n input` reads one numbered channel on its own. `n inputs` gathers the first `n` channels into one wide signal, so `2 inputs` reads a stereo pair. Their signatures live in *Core words* (4.1). How many of them you use is also how wide the patch declares itself. A patch that reads only `in` asks for one channel and receives a downmix of whatever feeds it. `2 inputs` asks for two and receives the pair as it stands. The moment a patch uses any of them it becomes an *effect* and can no longer be played as a sound. The patch text alone decides the role, with no mode to declare and no flag to set.

An effect patch can live in two places, and in both it runs in series as an insert. The patch reads what arrives, and its output replaces it. What changes is how much it gets to touch. On an *orbit* it sits at the end of the shared chain and works on that whole bus, the built-in reverb and delay tails included (see *What are orbits?* (8.1)). Inside a *voice* it works on that one voice alone, just before the envelope (see *What are insert effects?* (7.1)). In short:

- use `dsp` when you want to create an orbit effect.
- use `fx` when you want to create an insert effect.

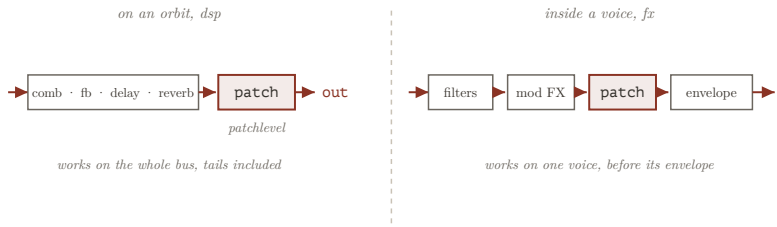


Figure 102: Where an effect patch lives: `dsp` *closes the orbit chain and works on the whole bus,* `fx` *sits inside one voice before its envelope. Both replace what they read.*

2.5.1 On the orbit: `dsp`

`dsp` hangs an Arf patch on the end of the current orbit's chain. This is where it parts company with the built-in orbit effects. Each of those adds a wet copy on top of a dry that always survives. The

patch *replaces* the bus with whatever it returns. That is what lets an orbit patch take things away. A filter here really darkens the bus, a saturator really drives it, a gate really cuts it:

```
"break" snd
{
  in
  400 lpf
  out
} dsp
.
```

The patch runs *after* the native chain (comb, feedback, delay, reverb), so it works on the finished orbit mix, tails and all. Closing that filter takes the reverb down with everything feeding it, which is the one thing a send can never do. Like every orbit setting, the patch is *sticky*. It stays on the orbit until you replace or clear it, across any number of events.

`patchlevel` (default 1) sets the dry/wet mix, from 0 for the untouched bus to 1 for the patch alone. It takes a modulation chain like any other orbit param. Sweeping it is how you fade an effect in without rewriting the patch:

```
"break" snd
{
  in
  0.02 0.9 comb
  out
} dsp
0 1.0 2 lfo patchlevel
.
```

Name a patch once with `defdsp`, then hand `dsp` that name instead of a block, and clear the slot with the reserved name `off`:

```
{
  in
  0.3 0.7 allpass
  out
} "wash" defdsp

;; use the dsp
"hat" snd
"wash" dsp
.

;; wash off the dsp
"hat" snd
"off" dsp
.
```

Redefining a `defdsp` name while it is playing swaps the running instance on the next event that names it. A name that is not installed, or one that names a sound-role patch, changes nothing. The orbit keeps what it already has.

2.5.2 Inside the voice: `fx`

`fx` inserts a patch into the voice's own chain, late: after the filters, lo-fi and mod FX words, just before the envelope applies. It replaces what it reads exactly as `dsp` does, but it has no `patchlevel` twin, so the balance is entirely yours to make. The dry is not lost. It is whatever `in` reads, so mix it back in yourself:

```
"break" snd
{
  in dup
  4000 lpf
  0.7 * swap
  0.3 * +
  out
} fx
.
```

On a mono voice `in` reads the voice signal and both input channels carry it. On a stereo voice `2 inputs` reads the pair and `in`

reads their downmix. Cagire sums a stereo patch on a mono voice back to mono, so the insert never changes the voice's width. `deffx` names an insert patch the same way `defdsp` does. `"off" fx` clears the slot on the next event, and the slot survives retriggers of a held voice. Cagire drops an event whose `fx` names a missing patch, and drops one whose voice pool is exhausted, matching how `"<name>"` sounds behave. That is the one place the two homes differ under failure. `dsp` on a name it cannot resolve leaves the orbit exactly as it was. `fx` on a name it cannot resolve costs you the note.

2.5.3 The effect words at a glance

The words that install and clear an effect in either home, gathered in one place:

word	what it does
`dsp`	Insert a patch at the end of this orbit's chain, replacing the bus: inline `{ in ... out }`, a bare `defdsp` name, or `"off"`
`defdsp`	Install an effect patch under a name for `dsp`
`patchlevel` / `plevel`	Dry/wet mix for the orbit patch, 0 dry to 1 patch only (default 1, takes ModChains)
`fx`	Insert a patch into this voice's chain, before the envelope; its output replaces the voice signal
`deffx`	Install an effect patch under a name for `fx`
`"off"`	Reserved name: `"off" dsp` and `"off" fx` clear the slot

2.5.4 What effects cannot do

An effect patch takes no per-note controls. Reading `in` alongside `gate`, `notefreq`, `vel`, or a `param` of its own is an install error. Two values it may still read are `bps`, the transport tempo in beats per second, so an effect can lock its own motion to the clock, and `bus`, an engine control bus lane, so a fader routed with `cc>bus` can reach an effect on an orbit (*Control buses* (3.18.6)). To shape an effect from the outside, use `patchlevel` and the voice's envelope in the sequencing language. To shape it from the inside, use generators

like `follow`, `dust`, and the oscillators. An effect may generate as freely as it processes.

One consequence of a self-generating effect is worth keeping in mind. An orbit that has been silent for a second goes to sleep, and the patch stops running with it until the bus carries signal again. A patch that has to sound at every moment belongs in a voice, not on an orbit. The Language tour covers the vocabulary an effect can draw from, beginning with *Oscillators & noise* (3.1). It comes back to this page's two words with worked recipes in *Effects in practice* (3.8). *Core words* (4.1) lists every structural word with its signature.

Chapter 3

Language tour

3.1 Oscillators & noise

Almost every patch begins with a source, a word that makes sound out of nothing. A source is the one thing a patch cannot do without. Every stage downstream has something to work on once a source has put a signal on the stack. Two families supply almost everything you will build. The periodic sources repeat a fixed waveform at a frequency you hand them, and that steady repetition is what the ear hears as pitch. They are `sine`, `saw`, `tri`, `pulse`, `varsaw`, and `blip`. The noises are the other family, every frequency at once and no pitch at all, until a filter carves one out of them. A few sources sit apart from both families. They make a stream of impulses or a jittering value rather than a tone, and you will use them as triggers and control signals to drive the rest of a patch.

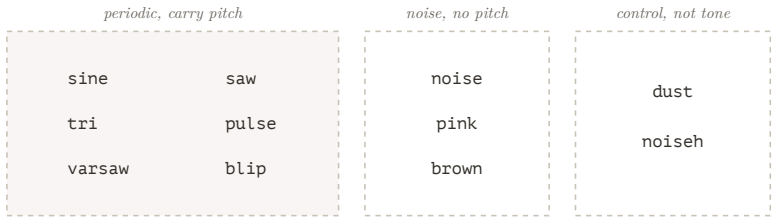


Figure 103: The three source groups: periodic sources carry pitch, noises carry none, and a few make control signals rather than tone.

3.1.1 A pure tone

`sine` is the simplest source there is: one frequency in Hz, one partial, and nothing else. It is loud at full scale, so you multiply it down before `out`, and what comes back is a clean test tone:

```
{
  440 sine
  0.2 * out
} synth .
```

That little patch is a whole voice in miniature. `440` sets the frequency. `sine` turns it into a tone. `0.2 *` trims it down from full scale, and `out` marks the result as the sound this voice plays. Swap the constant for `notefreq` and the source plays what you sequence instead of a fixed pitch (*The voice contract* (2.3)):

```
{
  notefreq sine
  0.2 * out
} synth
< a3 c4 e4 > note
.
```

Now the same patch tracks `< a3 c4 e4 >`, taking its frequency from the sequencer note by note rather than from a number baked into the text.

3.1.2 The bright shapes

`saw` and its siblings carry every harmonic at once. They are the raw material of subtractive synthesis, made to be filtered down. All of them are band-limited, so aliasing stays well down and the tone stays clean even when you sweep the pitch up. Point `saw` at `notefreq` and you have the bright starting point most of what follows is built on:

```
{
  notefreq saw
  0.2 * out
} synth
< a1 c2 e2 > note
.
```

`tri` is the mellow one of the set, soft where the saw is buzzy. `pulse` takes a second input, its `width`, and slowly modulating that width is the classic single-oscillator shimmer:

```
{
  110
  0.2 sine 0.4 *
  0.5 +
  pulse
  0.2 * out
} synth .
```

Here `0.2 sine 0.4 * 0.5 +` builds a slow signal that moves around a half-open pulse, and the tone thickens and thins as the width changes. `varsaw` morphs from saw to triangle as you move a width control. `blip` builds a tone from a chosen number of harmonics, so one harmonic gives you a sine and many give you a bright buzz. Signatures for all six are in *Oscillators* (4.2).

3.1.3 Noise without pitch

`noise` is every frequency at once. On its own it is plain hiss, but it becomes the starting point of snares and cymbals once a filter shapes it. `pink` is its better-balanced cousin, with equal power in every octave rather than the rising tilt of plain noise, so it sits more gently under a mix:

```
{
  pink
  0.3 * out
} synth .
```

On its own it is still just hiss, but that even balance is what makes pink noise the usual starting point for wind, waves, and breath.

3.1.4 Noise as a control

Not every noise is meant to be heard directly. `dust` fires random impulses at a density you set, a stream of triggers rather than a texture. Feed those impulses to an envelope and a single held voice plays its own irregular rhythm, with no sequencer involved:

```
{
  12 dust
  0.05 perc
  440 sine *
  0.3 * out
} synth .
```

Each impulse from `12 dust` fires the `perc` envelope. That envelope opens and shuts around the `440 sine`, so the one voice ticks along at random roughly twelve times a second. `noiseh` holds a random value and jumps to a fresh one at a rate you give in Hz. It is the sample-and-hold that scatters a pitch or a cutoff around at random:

```
{
  8 noiseh
  400 * 600 +
  sine
  0.2 * out
} synth .
```

Here `8 noiseh` picks a new number eight times a second, and mapping it onto a frequency span makes the `sine` hop between random pitches. The full non-periodic set (`brown`, `clipnoise`, `dust2`, `noisei`, `noiseq`) is in *Noise & sources* (4.3).

3.2 Subtractive shaping

The raw sources of the last page (*Oscillators & noise* (3.1)) are loud and shapeless: a `saw` carrying every harmonic, `noise` carrying every frequency at once. None of them is an instrument yet, only bright material. Subtractive synthesis is how you make an instrument out of that surplus: you *remove* material instead of adding it. The tool that does the removing is the filter, and every filter in a patch reads the same way. It takes the signal first, then a cutoff frequency in Hz, and decides which side of that cutoff survives. A `saw` is the perfect thing to feed one, dense enough that there is always something worth removing. This page runs it through the

low-pass, high-pass, and band-pass, then raises their resonance until the filter produces a tone of its own.

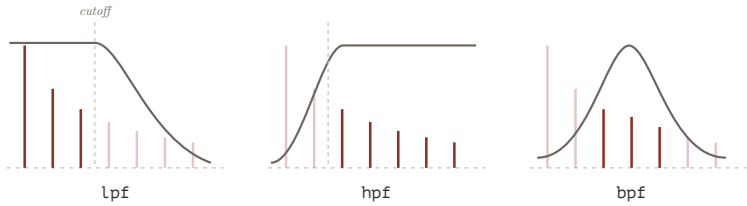


Figure 104: A saw enters with a full stack of harmonics; the low-pass, high-pass, and band-pass each carve a different slice out of it.

3.2.1 Low-pass & high-pass

`lpf` is a one-pole low-pass. It keeps what sits below the cutoff and softens everything above. Feed a `saw` through it at `800` Hz and the buzz loses its top:

```
{
  110 saw
  800 lpf 0.3 *
  out
} synth
.
```

This is the simplest filtering move there is. Drop `800` toward `200` and the saw goes from buzzy to dull. Re-send the script to hear each step. `hpf` is the mirror image, keeping what sits *above* the cutoff and dropping the rest, so the same source thins from below instead:

```
{
  110 saw
  400 hpf 0.3 *
  out
} synth
.
```

That one is the same synth with the filter flipped. Everything under `400` Hz is gone, leaving the bright edge of the saw. Raise the cutoff further and the tone keeps thinning toward a hiss.

3.2.2 Resonant filters

The one-pole filters are gentle. The resonant family adds a third input, a `0..1` resonance that emphasises the cutoff. `lpf2` takes `in cutoff res`, and at `0.8` a peak blooms right at `600` Hz:

```
{
  110 saw
  600 0.8 lpf2
  0.3 * out
} synth
.
```

`moog` has the same shape but four cascaded poles with feedback, the classic ladder, and it comes out darker and more vocal at the same settings:

```
{
  110 saw
  800 0.6 moog
  0.3 * out
} synth
.
```

That one already leans on its own resonance. Keep it in mind, because we will push it further at the end of this page.

3.2.3 Carving noise

Noise contains every frequency at once, so a band-pass turns it into something almost pitched. `bpf` keeps only a band around the cutoff, and the resonance sets how narrow that band is:

```
{
  noise 1000 0.8 bpf
  0.4 * out
} synth
.
```

Push the resonance higher and the band tightens toward a breathy whistle at `1000` Hz.

3.2.4 Sweeping the cutoff

The classic move is to modulate the cutoff with a slow sine. A sine is bipolar, `-1..1`, and `range` maps it straight onto a Hz span. Here the sweep signal comes first, so `swap` reorders the stack into the `in cutoff` order `lpf` wants:

```
{
  2 sine 200 800 range
  110 saw swap
  lpf 0.3 *
  out
} synth
.
```

Compute the source first and you need no `swap`. A `1` Hz sine sweeps a resonant band across the saw's whole spectrum:

```
{
  110 saw
  1 sine 0 1800 range
  0.8 bpf
  0.3 * out
} synth
.
```

When the modulator is not bipolar, `linlin` does the same mapping from any input span: `-1 1 200 800 linlin` is `200 800 range` spelled out.

3.2.5 The filter as oscillator

Push `moog`'s resonance toward `1` and it starts to ring on its own. At `0.99` it outputs a sine at the cutoff frequency almost regardless of the input. Set the cutoff to `notefreq` and the filter itself becomes the voice's oscillator, tuned by whatever you sequence:

```
{
  110 saw 0.2 *
  notefreq 0.99 moog
  0.3 * out
} synth
< a2 c3 e3 > note
.
```

Every filter (the notch, the raw `svf` taps, allpasses, shelves, resonators) is in *Patch filters* (4.4). Filtering takes material away. The next page, *Saturation & dynamics* (3.3), is about putting some back.

3.3 Saturation & dynamics

Filtering takes material away. This page does the opposite two things. It adds harmonics that were never in the source, and it rides the level of what is already there. Saturation is the adding half. It comes from the simplest fact in the language: a signal driven past what a curve can represent gets bent rather than passed on. Dynamics is the riding half, where the patch measures its own loudness and does something about it. The two belong together because they are what stands between a filtered oscillator and something that sounds finished. A raw `saw` through an `lpf` is correct and lifeless. A little drive and a little level control is most of the difference.

3.3.1 Driving a signal until it bends

`tanh` is the one word you need. It takes a signal and squashes it toward `-1` and `1` along a smooth curve. Quiet material passes through almost untouched, and loud material has its peaks rounded off. Nothing happens until you push something into it, which is why saturation is always two words: a gain, then the curve. Multiply a saw up by three and the peaks flatten into warmth rather than clipping into a square:

```
{
  110 saw 3 *
  tanh
  0.3 * out
} synth
.
```

Change the `3` and you sweep the whole character of the sound. At `1` it is barely doing anything, and at `3` it is thick. At `20` the saw is close to a square wave and the tone has stopped being a saw at all. That drive-then-curve pair is common enough that `distort` packages it, taking the amount as its own input so you can modulate it like anything else:

```
{
  notefreq saw
  0.5 sine uni 0.9 * distort
  0.3 * out
} synth
a1 note
.
```

Here a slow sine walks the drive up and down, so the note breathes between clean and gritty without its pitch or filter moving at all. That is worth knowing as a technique. Timbre can come from the drive alone.

3.3.2 Throwing information away

Where saturation adds harmonics smoothly, the lo-fi pair adds them by destroying detail, and the two kinds of detail give two very different sounds. `crush` throws away amplitude resolution, quantizing the signal to a number of bits. A smooth curve becomes a staircase, and the error you hear is a gritty buzz that tracks the signal:

```
{
  4 impulse [ 60 63 67 70 ] seq mtof saw
  3 crush
  600 lpf
  0.2 * out
} synth
.
```

Three bits is drastic on purpose. Anything above eight is a subtle grain, and the interesting range is low. `decim` throws away time resolution instead. It holds each sample for a while before taking a new one, which folds high frequencies back down as aliasing:

```
{
  notefreq saw
  4000 decim
  0.3 * out
} synth
a2 note
.
```

The tone keeps its pitch but grows a metallic shadow above it, and the lower you set the rate the further down that shadow reaches. Both words usually want to sit after the filter rather than before it. Anything you put downstream of them removes the grit you just paid for, which is occasionally the effect you are after and usually not.

3.3.3 A patch that listens to itself

`follow` is the one word here that measures rather than changes. It traces the loudness of whatever you hand it, rising at one rate and falling at another. What comes back is not audio but a control signal, the shape of the sound's own envelope. Aim that at a filter and the patch responds to its own dynamics, opening when it is struck and closing as it dies:

```
{
  4 impulse 0.05 perc 220 saw * :> src
  src 0.01 0.15 follow 0 1 300 4000 linlin :> cut
  src cut lpf
  0.3 * out
} synth
.
```

Read the first three lines in turn. Line one builds a plucked saw and names it. Line two measures that same signal and maps its loudness onto a cutoff range. Line three filters the signal with the number line two produced. The short attack lets the filter snap open on each hit. The longer release lets it close slowly. Lengthen the release and the sweeps drag behind the notes, which is a sound in itself.

`comp` is the finished version of that idea, a real compressor rather than a follower you wire up yourself. It watches the level, and whenever the signal exceeds the threshold it pulls the gain down by the ratio. Here a loud noise hit sits over a quiet saw, and the compressor stops the hit from swamping it:

```
{
  8 impulse 0.03 perc noise * 0.6 *
  110 saw 0.15 * +
  0.25 4 0.005 0.08 comp
  0.5 * out
} synth
.
```

The four numbers are threshold, ratio, attack, and release. Lower the threshold and the compressor catches more of the signal. Raise the ratio and it squeezes what it catches harder. Shorten the attack and it catches the hit sooner, at the cost of the transient. Cagire reads every input per sample, so the threshold can be a signal too, which is how you build ducking inside a single patch.

3.3.4 Where to put them

Order matters more here than anywhere else in the tour, because these words are not linear and so they do not commute. Saturating

before a filter means the filter removes the harmonics saturation just made. That is usually what you want: drive gives you material, and the filter shapes it. Saturating after a filter keeps every harmonic you generated, which is brighter and harsher. Compression before saturation controls how hard the drive is hit. Compression after it evens out what the drive produced. There is no correct order, only two different instruments, so swap two lines and listen rather than deciding on paper.

Every word on this page is listed with its signature in *Shaping, dynamics & pan* (4.5), alongside the stereo words the next page takes up.

3.4 Modulation & envelopes

Every number in a patch can be replaced by a signal. That is modulation: one signal steers another, and a parameter that sat at a fixed value starts to move. Loudness, pitch, a filter cutoff, the depth of another modulator, all of them take it.

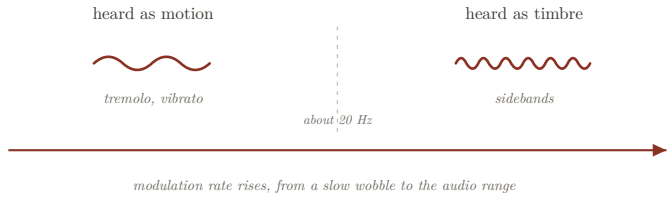


Figure 105: One modulator, two regimes: heard as motion below roughly twenty cycles a second, fused into sidebands and timbre above it.

3.4.1 A slow sine on loudness

Loudness is the easiest destination to hear. Multiply a steady tone by a slow sine and its level rises and falls in a smooth pulse, which is tremolo. A bare `sine` swings from `-1` to `1`, and loudness has no meaning below zero, so `uni` folds the modulator up into the `0..1` range before it reaches the multiply:

```
{
  notefreq sine
  5 sine uni *
  0.3 * out
} synth
a2 note
.
```

The `5` is the wobble rate in Hz, low enough that you feel it as movement and do not hear a pitch of its own.

3.4.2 Faster, until it becomes timbre

Now push the same modulator faster. Past a certain rate the ear stops tracking the swings and the two tones fuse into one timbre. Multiplying two bipolar signals directly, with no `uni`, is ring modulation, a hard metallic clang. *Amplitude & Ring Modulation* (4.3) covers the sidebands it produces:

```
{
  notefreq sine
  200 sine *
  0.3 * out
} synth
a3 note
.
```

Put `uni` back on the modulator and the same setup becomes amplitude modulation. The original tone stays present and is not cancelled through zero.

3.4.3 Into the frequency input

The other place to aim a modulator is the frequency itself, added into the pitch before the oscillator reads it. A slow shallow wobble there is vibrato. Here the pitch drifts ±6 Hz around its note, five times a second:

```
{
  notefreq
  5 sine 6 * +
  saw
  900 lpf
  0.2 * out
} synth
a2 note
.
```

That adds a `5` Hz sine, scaled to a ±6 Hz swing, straight onto `notefreq` before `saw` reads it, so the pitch never settles. Push the same modulator into the audio range and the wobble fuses into timbre. This is frequency modulation (see *Frequency Modulation (FM)* (4.6)). Derive the modulator's rate from `notefreq` and the colour holds steady as the tune moves, because the ratio between carrier and modulator then stays constant on every note:

```
{
  notefreq
  notefreq 2 * sine 100 * +
  sine
  0.3 * out
} synth
< a2 c3 e3 > note
.
```

The modulator runs at twice `notefreq` with a depth of `100`, so each note in the `<a2 c3 e3>` phrase keeps one bell-like colour as the pitch climbs.

3.4.4 Envelopes from the gate

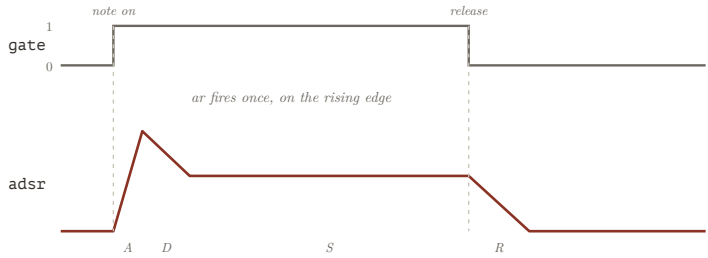


Figure 106: The gate signal becomes a shape: adsr traces its held attack, decay, sustain, and release; ar fires once on the rising edge.

Modulators run on a clock of their own. When a patch needs to move in time with the note that started it, the engine gives you `gate`. It writes `gate` into every running patch one sample at a time, holding it at `1.0` for as long as the note is held and dropping it to `0.0` the instant the voice releases (see *The voice contract* (2.3)). Feed it to `adsr` and the patch gets its own contour in step with the sequencer, rising on the note-on and falling away on the note-off:

```
{
  gate 0.05 0.2 0.6 0.4 adsr
  notefreq saw 1200 lpf *
  0.3 * out
} synth
a2 note
.
```

`ar` wants a trigger and not a held level, and the rising edge of `gate` is exactly that. It fires one attack-and-release shape at every note-on then ignores the rest of the note. An envelope reshapes anything you multiply it into, not only amplitude. Here one drives a filter, sweeping the cutoff open and shut on every note:

```
{
  notefreq saw
  gate 0.01 0.2 ar 3000 * 200 +
  lpf
  0.3 * out
} synth
< a1 a1 c2 e2 > note
.
```

That envelope scales up by `3000` and lifts by `200`, so the `lpf` cutoff climbs from `200` Hz to a bright peak and closes again across each step of the `<a1 a1 c2 e2>` phrase.

3.4.5 Triggers inside the patch

A patch need not wait on the sequencer for its rhythm. It can keep a pulse of its own. `impulse` fires one-sample spikes at a steady rate in Hz. `perc` catches each spike and stretches it into a short percussive decay, so one held voice can carry a whole rhythm inside itself:

```
{
  8 impulse 0.08 perc
  notefreq sine *
  0.3 * out
} synth
a2 note
.
```

That pulses eight times a second, and each spike becomes a `0.08`-second `perc` blip ringing a `notefreq` sine, all under one sustained `a2`.

3.4.6 The patch envelope & the voice envelope

A patch does not always need an envelope of its own, because the voice it plays in already has one. You set that outer envelope from the sequencing language, and it wraps whatever the patch sends to `out`. The plain loudness contour is covered already. Put an envelope inside the braces when the shape *is* the sound and not its volume: a filter sweep, the depth of an FM modulator, a rhythm of internal pulses.

One caveat comes from *The voice contract* (2.3). Retriggering a held voice does not dip `gate` back to zero, so a patch envelope driven by the gate does not re-fire. The voice's own envelope re-articulates regardless. *Envelopes & triggers* (4.7) lists every envelope and trigger word with its full signature.

3.5 Names, locals & reuse

Every patch so far has been one long chain, read from left to right and then forgotten. That serves a single line of signal well. But sometimes you want to hear one sound from two places, or apply the same shaping to two different sources. Then you need a way to name a part of the patch and point back at it. Arf has a small family of naming words for exactly this. They differ on a single question: does the name stand for a recipe you paste again, or for a running node you share? A `:` definition is a recipe, expanded as a fresh copy at every use. A `~`-prefixed definition is a value, computed once so that every later use reads the same node. `as name` binds a live signal mid-chain, so a word further down can reach back and reuse it. The figure lines the three up side by side, and the rest of this page takes them one at a time.

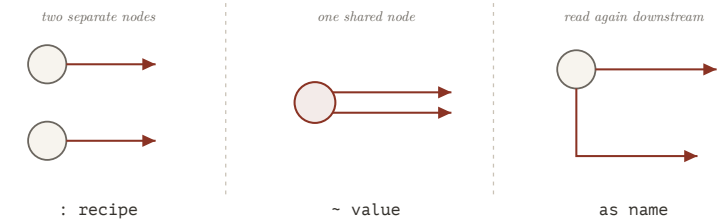


Figure 107: Three ways to reuse a signal: a colon recipe pastes a fresh copy, a tilde value runs once as one shared node, and as name binds a live node mid-chain.

3.5.1 A recipe, pasted fresh

`: name … ;` defines a word whose body is a recipe. Nothing runs when you define it. Arf remembers the tokens and pastes them inline

wherever the name appears later, the leftmost path in the figure. Here `rez` is a resonant low-pass you can add after any source:

```
{
  : rez 700 0.8 lpf2 ;
  110 saw rez
  0.2 * out
} synth
.
```

That reads as one filter, but the value of a recipe is reuse. Point the same `rez` at two sources and each use expands a fresh copy. You get two independent filters built from one definition, edited in a single place:

```
{
  : rez 700 0.8 lpf2 ;
  55 saw rez
  222 saw rez
  +
  0.15 * out
} synth
.
```

This is the figure's fork on the left: one recipe, two separate boxes downstream that share nothing at runtime.

3.5.2 One node, shared

A ~-prefixed definition is not a recipe but a value. Its body runs once, and every later use reads the same node, the middle path in the figure. That difference matters whenever one signal should steer several things at the same time. Here one slow `sine` bends the pitch and opens the filter together, so both move in lockstep:

```
{
  : ~wob 0.5 sine ;
  220 ~wob 20 * + saw
  900 ~wob 500 * + lpf
  0.2 * out
} synth
.
```

Written as a plain `: wob 0.5 sine ;`, each use would paste its own fresh oscillator, two separate sines drifting apart. `~wob` is one signal computed once and heard from two places. That is what keeps the pitch bend and the filter sweep locked to a single motion.

3.5.3 Binding a live node

A definition is not the only way to name something. `as name` binds whatever is on top of the stack right where you write it. A later bare `name` reuses that same node without any definition at all, the right-hand path in the figure. It is the quickest way to fork one live signal into two chains. Here one saw feeds two filter paths that rejoin at the sum:

```
{
  110 saw as src
  800 lpf
  src 3000 hpf 0.2 * +
  0.3 * out
} synth
.
```

Arf builds the saw once and binds it to `src`. The low-pass filters it, then `src` feeds the very same node to a high-pass, and the two bands add back together. The oscillator runs a single time and you hear it through two different filters.

3.5.4 One sample late

A trailing quote, `name'`, reads a bound name one sample late, the previous sample of whatever `as name` binds. The value it returns is already a sample old, so the read is allowed to sit *before* the binding. That is how a signal can feed itself. This single quote is the basis of every echo and resonator loop, and *Channels & feedback* (3.6) covers it in depth.

3.5.5 Locals in a definition

A definition can take stack arguments like any other word. Writing `{ a b }` at the start of the body names them, left to right matching bottom to top. Inside the body you read the inputs by name instead of juggling `dup` and `swap`. These are Arf's own braces,

and they nest cleanly inside the outer capture. Here `pluck` takes a trigger and a frequency:

```
{
  : pluck { hit f }  f saw  hit 0.005 0.3 ar * ;
  4 impulse trig 330 pluck
  0.2 * out
} synth
.
```

`pluck` reads a trigger and a frequency, strikes a saw with a short attack-release, and works for any pair you hand it. Locals add no node of their own to the graph. They only give the arguments a name, so a longer definition stays readable without a run of stack shuffles. The same brace block names a word's arguments in the sequencing language too, with the same call-order meaning, so a definition reads the same on both sides of the braces (see *Defining words* (3.11.1)).

To name a value partway through a graph, without a full argument list, `:>` binds the top of the stack to the next name. It is the one-name companion to `{ … }`, and where `as` leaves its signal in place, `:>` pops it:

```
{
  110 saw :> src
  src 800 lpf
  src 3000 hpf
  +
  0.2 * out
} synth
.
```

`:> src` names the saw and takes it off the stack. The two `src` reads feed one shared node into a low-pass and a high-pass, summed back together. It is the popping cousin of the `as src` version above. Every word on this page lives in *Core words* (4.1), with a signature and a compile-checked example each.

3.6 Channels & feedback

A signal inside a patch is not always a single stream. It can carry several channels at once, and every word you write applies to all of them together. A filter on a two-channel signal is really two filters working in step. When a word receives inputs of different widths, the result is as wide as the widest of them. The narrower inputs are reused in turn, which is why a single number scales every channel of a list at once. Most of the tour has stayed on a single channel, because a single channel is all a mono sound needs. This page adds the two ideas that give a patch width and depth. Channel lists set how many streams run in parallel and give a stereo `out` its left and right. One-sample feedback lets a stream read its own past and echo into itself.

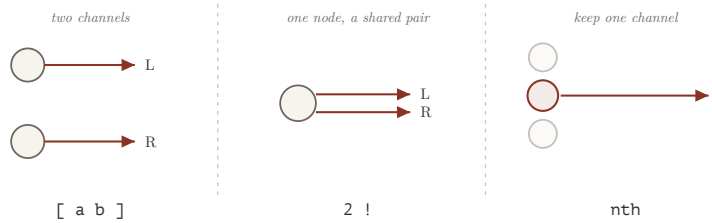


Figure 108: A bracket list builds parallel channels for a two-channel out; 2 ! replicates a mono node into a shared pair, and nth keeps one channel.

3.6.1 A stereo signal

`[ … ]` builds a channel list, and the number of entries you put between the brackets is the width of the signal that comes out. Two entries make a stereo pair, and a two-channel `out` is the only way a patch ever declares itself stereo (*The voice contract* (2.3)). Everything upstream is simply one channel or many until `out` reads the width off the list. Detune the two sides by a single hertz and they drift slowly in and out of phase. The tone widens and narrows on its own and never sits quite still:

```
{
  [ 110 saw 111 saw ]
  0.2 * out
} synth
.
```

That one builds a saw at `110` on the left and a saw at `111` on the right. The one-hertz gap between them does the whole job. The two waves slip a cycle apart and back roughly once a second, which the ear reads as movement. Notice that `0.2 *` scales both channels in a single word, because a word laid on a list touches every channel in it. Widen the gap and the motion speeds up. Make the two sides equal and the width collapses back to a point dead centre.

3.6.2 Replicating and picking channels

Two words move between widths, in opposite directions. `n !` takes a single node and replicates it into `n` channels that all share the one computation. You get the same tone in stereo without paying for it twice and without any detune between the sides:

```
{
  notefreq saw
  600 lpf
  0.2 * 2 !
  out
} synth a1 note
.
```

Here the saw is built and filtered once. `2 !` fans that finished node out into two channels, and `out` sends them left and right as an identical pair. It is a mono sound sitting dead centre, computed once. `nth` goes the other way. It reaches into a channel list and keeps a single channel of it, chosen when the patch is parsed rather than while it runs. That matters most for a word that hands back several channels on one pass. `svf` is a state-variable filter with four outputs at once (lowpass, bandpass, highpass, and notch), and a bare `out` after it would keep only the first. `nth` is how you say which branch you actually want:

```
{
  [ 110 saw 800 0.7 svf ]
  0 nth
  0.3 * out
} synth
.
```

`0 nth` keeps output zero, the lowpass, and drops the other three before they cost anything downstream. Wrapping `svf` in `[ … ]` is what turns its four outputs into a list `nth` can index. Ask for `1` instead and you hear the bandpass out of the very same filter.

3.6.3 Stacking structure

`( … )` records a quotation, a stretch of patch saved but not yet run. `n times` stamps that quotation into the chain `n` times in a row. It is how you repeat a stage without typing it out. The repetition compounds, because each copy takes the output of the copy before it. Four allpass filters in series is the classic diffusion chain. It smears a single click into something that starts to sound like a room:

```
{
  8 impulse
  ( 0.03 0.5 allpass ) 4 times
  0.3 * out
} synth
.
```

`8 impulse` fires a steady stream of clicks, and each one passes through the four allpasses laid down by `4 times`. Every filter adds a little more delay and dispersion, until the sharp edge of the impulse is blurred into a soft, grainy tail.

3.6.4 A single echo

`delay` is the word that lets a patch hold onto sound. It replays whatever you feed it a fixed number of seconds later. Its output is the delayed copy alone, with none of the dry signal mixed in:

```
{
  220 sine
  0.2 *
  0.3 delay
  out
} synth
.
```

Here the `220 sine` reaches `out` only through the delay, so the tone starts `0.3` seconds after the note fires. To hear a real echo, the original and its repeat together, keep the dry signal with `dup` and add the delayed copy on top of it. *Delays & time* (4.6) shows that recipe. Every delay claims a real block of memory to store its samples in, and the total a single note may hold is capped. Know that before you stack a dozen of them (*Limits* (4.10)).

3.6.5 Closing the loop

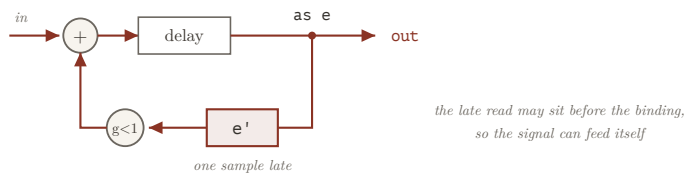


Figure 109: as e binds the live signal and e prime reads it one sample late, so the loop can feed itself without a cycle.

A single `delay` plays its input back once, because nothing feeds the delayed signal into the delay again. To hear an echo repeat and slowly die away, you have to mix the delayed signal back into the delay's own input. That is a loop: the output becomes part of the next input. The problem is that a signal cannot depend on itself in the same instant. The value would have to exist before it was computed. Arf's answer is the one-sample tick. `as name` binds the live signal at the point you write it. `name'`, the same name with a trailing quote, reads what that node held one sample ago. A tick read returns a value that is already a sample old, so the read is

allowed to appear *before* the binding it refers to. The compiler reads one step back in time instead of asking for a value that does not exist yet. That single sample of delay is what lets the loop close without a cycle:

```
{
  4 impulse 0.05 perc 440 sine * 0.2 *
  e' 0.7 * +
  0.25 delay as e
  out
} synth
.
```

Read it as a round trip. `4 impulse 0.05 perc 440 sine *` is the fresh sound entering the loop, a short blip made from a `440` sine. `e' 0.7 *` is last round's output, read one sample late and turned down. The two are summed, delayed by `0.25` seconds, and the result is bound as `e` so the next pass can read it in turn. The `0.7` is the loop gain, and it is the number to watch. Each trip around the loop multiplies the signal by it. Anything under `1` shrinks the echoes toward silence. Anything at or above `1` grows them without bound until the output overloads. Keep it below one and the tail fades the way an echo should.

3.6.6 Colouring the loop

The loop does not have to return exactly what went into it. Anything you place between the tick read and the binding runs on every trip, so a filter inside the loop colours each repeat a little more than the last. Put a lowpass in the path and the echoes lose their highs one repeat at a time, the classic dub delay:

```
{
  2 impulse 0.02 perc 220 sine * 0.1 *
  d' 0.75 * +
  1200 lpf
  0.4 delay as d
  out
} synth
.
```

The feedback gain `0.75` sets how many repeats you get before the tail dies. The `1200 lpf` inside the loop sets how fast each pass darkens. Together the two numbers shape a delay that fades and dims at once. The filter sits inside the loop rather than after it, so its effect stacks. The fifth echo has been through the lowpass five times and is muffled far more than the first.

A feedback name carries a single channel unless you say otherwise. `fb name n` widens the latch to `n` channels, so both `as name` and `name'` bind and read that many streams together. Declare the loop two channels wide and it can carry a stereo echo, each side feeding back into itself:

```
{
  fb e 2
  [ 220 330 ] sine
  0.1 *
  e' 0.75 * +
  0.3 delay as e
  out
} synth
.
```

`fb e 2` sets `e` up as a two-channel latch before the loop is built. The `[ 220 330 ]` seeds the two sides with different pitches. From there the left and right echoes decay along their own separate paths.

3.6.7 A comb on noise

`comb` packages that same delay-and-feedback loop into one word (`in time fb`). It delays its input by `time` seconds and feeds back a fraction `fb` of the result, exactly the loop you just wired by hand. At the delay times you have been using, tenths and hundredths of a second, you hear the repeats as separate echoes. Shrink `time` down to a few milliseconds and they arrive so fast that the ear hears a *pitch* instead, tuned to the length of the delay. A burst of noise through a five-millisecond comb is the classic recipe for a plucked string:

```
{
  4 impulse 0.02 perc noise *
  0.2 *
  0.005 0.9 comb
  0.3 * out
} synth
.
```

Each impulse fires a short noise burst, and `0.005 0.9 comb` sends it round a five-millisecond loop with a high feedback of `0.9`. What returns is not a run of echoes but a ringing tone. The comb has turned the delay time into a fundamental frequency, and the noise into a pluck. Every delay word (`delay`, `comb`, `allpass`, and the full reverb `verb`) carries its signature in *Delays & time* (4.6). Width so far has meant how many channels a patch carries. Aiming those channels at a point in the stereo field is the next page, *Placing sound in stereo* (3.7).

3.7 Placing sound in stereo

Channels & feedback (3.6) showed how a patch gets two channels: a list of two entries reaching `out`, and everything upstream widening to match. That gives a patch width, but width is not the same as placement. A two-entry list puts one signal hard left and the other hard right, and nothing in between. This page is about the in between. It covers aiming one sound at a point in the field and spreading several sounds across it. It also covers which of those jobs belongs to the patch and which belongs to the sequencing language.

3.7.1 One sound, one position

`pan2` takes a mono signal and a position and gives back two channels, a left and a right. The level is shared between them, so the sound keeps its loudness wherever you put it. The position runs from `-1` at the far left through `0` in the middle to `1` at the far right. It is a multi-output word, so it leaves *two* values on the stack rather than one. A bare `out` after it would take only the first. Gathering the pair in a channel list is what makes it a stereo voice:

```
{
  [ notefreq saw 0.2 * 0.6 pan2 ]
  out
} synth
a2 note
.
```

The brackets are doing real work there. Without them Cagire computes the right channel and then drops it, and you hear a quiet mono voice. With them, `out` reads a two-channel list and the voice is stereo, placed at `0.6`, a little right of centre.

Cagire reads every input per sample, so the position does not have to be a number. Feed it a slow oscillator and the sound moves:

```
{
  [ 220 saw 0.2 * 0.3 sine pan2 ]
  out
} synth
.
```

That is the classic auto-pan, one sweep across the field every three-and-a-bit seconds. Speed the sine up past a few hertz and the movement stops reading as position and starts reading as tremolo. *Modulation & envelopes* (3.4) crosses the same threshold with amplitude.

A `rand` in the position gives each note its own place instead, which is how you widen a repeated figure without ever moving anything:

```
{
  [ notefreq saw 0.2 * -1 1 rand pan2 ]
  out
} synth
< c3 e3 g3 c4 >
note
.
```

Each note draws once and holds, so the phrase scatters across the field while every individual note stays perfectly still.

3.7.2 Many sounds, spread out

When there are several signals rather than one, placing them individually gets tedious fast. `splay` does the whole job at once. Hand it a channel list and it ranks the channels evenly across the field, the first hard left, the last hard right, everything else spaced between. It then sums the result down to a left and a right. It compensates the level as the list grows, so a wide splay does not come out louder than a narrow one. Four partials of one tone, fanned out:

```
{
  [ [ 220 330 440 550 ] sine 0.15 * splay ]
  out
} synth
.
```

Read the brackets from the inside out. The inner list builds four frequencies. `sine` spreads across all four to make four oscillators. `0.15 *` scales all four in one word. `splay` fans them into a left and a right, and the outer list gathers that pair for `out`. This is the cheapest way to make one note sound large, and it works just as well on four detuned saws, four filtered noise bands, or four copies of a definition.

`splay` is also the natural partner of the `!` and `nth` words from *Channels & feedback* (3.6): build wide, splay down, send out.

3.7.3 Two rules that catch people

Both of these words hand back more than one channel, and words like that take a single channel in. Cagire refuses a channel list fed to `pan2` at install rather than guessing at it. Pan first and widen afterwards, never the other way round. If you have a wide signal that needs placing, `mix` it down first or splay it instead.

The second rule is about how widths meet. When a word receives inputs of different widths, the result is as wide as the widest of them and the narrower inputs are reused in turn. That is why `0.15 *` scales four channels with one number. It is also why a two-channel signal filtered by a three-channel cutoff comes out three channels wide rather than raising an error. Mostly this does what you want

silently. When a patch comes out wider than you expected, this rule is why.

3.7.4 What the patch does not decide

A patch declares one channel or two, and there it stops. Sending a voice to a third speaker, spreading it across a ring, or deciding where an orbit sits in a larger room are all sequencing-side concerns. The words in *Panning* (3.5) and the routing in *Multichannel output* (9.1) handle them. The patch's job is to hand the engine a well-made stereo image. What the engine does with it afterwards is not the patch's business.

Signatures for both words are in *Shaping, dynamics & pan* (4.5). That completes the source-building half of the tour. *Effects in practice* (3.8) turns the same vocabulary around, to process sound rather than make it.

3.8 Effects in practice

Patches as effects (2.5) established the rule. Read audio input and the patch becomes an effect, installed on an orbit with `dsp` or inside a voice with `fx`. That page is about the mechanism. This one is about what to build with it, because an effect patch is a different craft from a source patch. It has no note to follow, no gate to open, and no velocity to read. What it has instead is a signal arriving from somewhere else and one clock, `bps`, that ties it to the transport. Working inside that constraint is most of the skill.

3.8.1 One channel or two

Everything starts with how much you read. A patch that uses only `in` asks for one channel and receives a downmix. That is the right choice whenever the effect is going to collapse the image anyway. A patch that uses `2 inputs` asks for the pair and keeps the stereo it was given. The difference is audible on anything with width, so decide it deliberately rather than by habit:

```
"break" snd
{
  2 inputs
  bps phasor bi 300 4000 range
  lpf
  out
} dsp
.
```

That is a tempo-locked filter sweep across a break, and it is worth reading twice. `2 inputs` gives two channels. `bps phasor` runs one ramp per beat at whatever tempo the transport is at. `bi` turns that ramp bipolar so `range` can use all of it. The single cutoff value that comes out reaches both channels at once, because a narrower input spreads across a wider one. Change the tempo and the sweep follows without a number in the patch changing.

3.8.2 Effects that make their own motion

An effect cannot read a note, but nothing stops it generating. Oscillators, `dust`, `impulse`, and `follow` all work exactly as they do in a source. An effect can therefore have a life of its own rather than sitting still until something plays through it. A tremolo is the smallest example, and it needs nothing but a multiply:

```
saw snd
a2 note
2 s
{
  2 inputs
  6 sine uni 0.7 * 0.3 +
  *
  out
} dsp
.
```

`follow` is the more interesting generator, because it turns the incoming signal itself into control. This one opens a filter in proportion to how loud the input is, which makes the effect louder-is-brighter without any sequencing at all:

```
"break" snd
{
  in :> dry
  dry 0.005 0.2 follow 0 1 400 6000 linlin
  dry swap lpf
  out
} dsp
.
```

The `swap` is there because the cutoff comes before the signal it filters, the same reordering *Subtractive shaping* (3.2) runs into. Compute the dry signal first, name it with `:>`, and you can use it twice in whichever order the words want.

3.8.3 Both homes replace, so the dry is your job

`dsp` and `fx` differ in what they get to touch, not in how they behave. Both are inserts. Whatever the patch returns is what comes out, and the signal that went in is gone unless you kept it. A patch that returns only its processed signal is therefore fully wet in either home. That is what you want for a filter or a gate, and rarely what you want for a crusher. That is why effect patches so often mix by hand:

```
"break" snd
{
  in :> dry
  dry 3 crush 2000 lpf 0.6 *
  dry 0.4 * +
  out
} fx
.
```

Sixty percent crushed, forty percent clean, summed back together. Change the two numbers and you have a wet/dry control. Delete the second line and the effect is fully wet. Installed with `dsp` the same patch works the same way, with one extra lever. `patchlevel` fades between the untouched bus and the patch from outside the braces. A blend you want to perform belongs there, and a blend that is part of the effect's character belongs in the patch text.

3.8.4 Naming an effect you will reuse

An effect you like is worth a name, and `defdsp` and `deffx` give it one the same way `defsynth` does for instruments. A prelude is the natural home, so every pattern in the bank can reach the same effect:

```
{
  2 inputs
  0.03 0.6 allpass
  0.05 0.5 allpass
  0.3 *
  out
} "wash" defdsp
```

From there `"wash" dsp` puts it on an orbit and `"off" dsp` takes it away, and editing the definition swaps the running effect on the next event that names it. That is the same live-editing loop instruments get. An effect is just a patch, and patches are meant to be changed while you listen.

3.8.5 Two things to remember

An orbit that has gone quiet goes to sleep after about a second, and a sleeping orbit stops running its patch. For a filter or a delay that is exactly right, since there is nothing to process. For a patch whose whole job is to generate, it means the sound stops when the bus does. If an effect has to run continuously, it belongs inside a voice, where the note keeps it alive.

The other is that an effect gets no per-note controls at all. Reading `gate`, `notefreq`, `vel`, or declaring a `param` alongside `in` is refused at install, not ignored at runtime, so you find out immediately. Shape an effect from the outside with `patchlevel` and the voice's own envelope, and from the inside with the generators above.

The full effect vocabulary is the same one sources use, minus the per-note reads. *Core words* (4.1) has the structure, and the reference pages from *Oscillators* (4.2) onward have the rest. The tour closes on *Putting it together* (3.9), which builds three finished instruments out of everything up to here.

3.9 Putting it together

Every page so far has isolated one idea and shown it working on its own. That is how you learn a vocabulary, but it is not how you build an instrument. A real patch is four or five of those ideas layered until the result stops sounding like a demonstration. This page builds three instruments end to end, a kick, a bass, and a pad, and names the page each move came from. It introduces nothing new. If you have read the tour, you already know every word here. What is left is the habit of combining them.

3.9.1 A kick drum

A kick is a pitch envelope and an amplitude envelope aimed at a sine, and the pitch one is what makes it a kick rather than a thud. `xline` falls exponentially, the curve a drum head actually follows. A drop from `400` Hz to `50` in eighty milliseconds gives the click and the body in a single word. This is the recipe *Envelopes & triggers* (4.7) uses to introduce `xline`, written out here with its bindings made explicit. What matters on this page is what you add to it afterwards:

```
{
  gate trig :> hit
  hit 400 50 0.08 xline sine
  hit 0.25 perc *
  0.8 *
  out
} synth
.
```

The first line is the piece worth studying. `gate` is `1` while the note is held, so `trig` turns its rising edge into a single impulse at the note-on. `:> hit` names that impulse so both envelopes can fire from it (*Modulation & envelopes* (3.4), *Names, locals & reuse* (3.5)). Everything after that is two envelopes and a multiply. Shorten the `0.08` for a tighter kick, lengthen the `0.25` for a longer tail, and raise the `400` for more click.

Adding a little saturation gives it weight without raising the level, which is the *Saturation & dynamics* (3.3) move:

```
{
  gate trig :> hit
  hit 400 50 0.08 xline sine
  hit 0.25 perc *
  2 * tanh
  0.5 *
  out
} synth
.
```

The `2 *` drives it and `tanh` catches the peaks, so the kick reads louder through a mix while its actual peak is lower than before.

3.9.2 A bass

A bass wants to track the notes you write, so it starts from `notefreq` rather than a constant (*The voice contract* (2.3)). Two detuned saws give it body, a resonant filter gives it character, and an envelope on the cutoff gives it movement:

```
{
  notefreq saw
  notefreq 1.006 * saw
  + 0.4 *
  gate 0.005 0.12 ar 2500 * 200 + 0.6 moog
  0.5 *
  out
} synth
< a1 a1 c2 g1 >
note
.
```

The two saws are a fifth of a percent apart, which is close enough that you hear thickness rather than two pitches (*Oscillators & noise* (3.1)). The envelope line does the classic thing. `ar` fires on the gate's edge, scales up to `2500`, and lifts by `200`, so the `moog` cutoff snaps from `200` Hz to nearly `2700` and closes again on every note (*Subtractive shaping* (3.2)).

Making the filter movement follow the playing rather than a fixed shape is a one-line change. Use the follower instead of the envelope:

```
{
  notefreq saw notefreq 1.006 * saw + 0.4 * :> src
  src 0.005 0.1 follow 0 1 200 3000 linlin :> cut
  src cut 0.6 moog
  0.5 *
  out
} synth
< a1 a1 c2 g1 >
note
.
```

Now the cutoff follows how loud the note actually is, so velocity and the voice's own envelope both reach the filter without any explicit wiring.

3.9.3 A pad

A pad is where width and slow motion matter more than punch. Start from the stereo idea in *Channels & feedback* (3.6): a list of two entries, detuned against each other, so the two sides drift in and out of phase:

```
{
  [ notefreq notefreq 1.004 * ] saw
  600 lpf
  0.15 *
  out
} synth
< c3 e3 g3 >
note
2 s
.
```

That is already usable, and it is four moves. A slow shared modulator makes it move, and this is the one place where the difference between a recipe and a value matters. Write the oscillator with a ~ prefix and one oscillator drives both the pitch drift and the filter, so they move together rather than wandering apart (*Names, locals & reuse* (3.5)):

```
{
  : ~drift 0.15 sine ;
  [ notefreq ~drift 3 * + notefreq 1.004 * ~drift 2 *
- ] saw
  900 ~drift 400 * + lpf
  0.15 *
  out
} synth
< c3 e3 g3 >
note
2 s
.
```

One `0.15` Hz sine now bends the two sides in opposite directions and opens the filter at the same time. The result reads as a single slow breath rather than three unrelated wobbles. Spreading it wider is the *Placing sound in stereo* (3.7) move, four partials splayed across the field instead of two channels:

```
{
  [
    [ notefreq notefreq 1.004 * notefreq 2 * notefreq
3.01 * ] saw
    1200 lpf
    0.08 *
    splay
  ]
  out
} synth
< c3 e3 g3 >
note
2 s
.
```

3.9.4 Giving them names

Three instruments built inline are three instruments that vanish with the step that holds them. Move each into a `defsynth` in a bank prelude. The whole bank can then play them by name, and you can keep editing them while patterns run (*Named patches* (2.2)):

```
{ gate trig :> hit  hit 400 50 0.08 xline sine  hit
0.25 perc *  2 * tanh 0.5 * out } "kik" defsynth
{ notefreq saw notefreq 1.006 * saw + 0.4 *  gate
0.005 0.12 ar 2500 * 200 + 0.6 moog  0.5 * out }
"bas" defsynth
```

```
"kik" snd .
```

```
"bas" snd
< a1 a1 c2 g1 > note
.
```

Note the short names. `kick` and `bass` are already sample folders in most sets. A patch loses to a built-in source but beats a sample folder, so naming a patch `kick` would quietly take over a name you were using (*Named patches* (2.2)).

3.9.5 Where to go next

Everything on this page is the tour's vocabulary rearranged. What is left is the reference, which lists every word with its signature and a compile-checked example, starting at *Core words* (4.1). Two habits are worth carrying out of the tour, and both are smaller than any of the techniques. Name an intermediate the moment you need it twice. Change one number at a time while the pattern is playing. Arf is a small language, and you learn almost all of it by turning a value and listening to what happens.

Chapter 4

Reference

4.1 Core words

The structural vocabulary of a patch: everything that is not an oscillator, filter, or number. These words route signals to the output, rearrange the stack, group channels, define and name parts, and read what the engine feeds in. *Names, locals & reuse* (3.5) and *Channels & feedback* (3.6) teach the ideas. This page lists the words. A few of them read the tokens that follow rather than values from the stack. Their signatures therefore show those tokens literally in front of the `( -- )`: `as name`, `:> name`, `fb name n`, `buf name secs`, and `param name default`.

4.1.1 Output

WORD	SIGNATURE	WHAT IT DOES
`out`	`( sig -- )`	Add the signal to the output (repeated `out`s sum).

Every patch ends here. There is no default routing. A patch that never reaches `out` is a compile error. One `out` of a single signal is mono, and an `out` of a two-channel list is stereo (*The voice contract* (2.3)). Every example below ends on it.

4.1.2 Stack

Arf reads a patch left to right, and values land on a stack until a word consumes them. These words shuffle the stack when the order is wrong. They are the same words as the sequencing language, working on Arf's own stack.

WORD	SIGNATURE	WHAT IT DOES
`dup`	`( a -- a a )`	Duplicate the top of the stack.

word	signature	what it does
`swap`	`( a b -- b a )`	Swap the top two stack items.
`drop`	`( a -- )`	Discard the top of the stack.
`over`	`( a b -- a b a )`	Copy the second item to the top (shared nodes, like `dup`).
`rot`	`( a b c -- b c a )`	Rotate the third item to the top.
`nip`	`( a b -- b )`	Discard the second item.
`tuck`	`( a b -- b a b )`	Copy the top item beneath the second.
`2dup`	`( a b -- a b a b )`	Duplicate the top two items as a pair.

`swap` is the one you reach for most, whenever a modulator comes before the signal it steers (*Subtractive shaping* (3.2)).

4.1.3 Channels & lists

word	signature	what it does
`[`	`( -- )`	Begin a channel list.
`]`	`( -- chans )`	End a channel list (concatenate everything since `[`).
`!`	`( sig n -- sig×n )`	Replicate a signal across `n` channels (`n` is a constant).
`nth`	`( chans n -- chan )`	Pick channel `n` of the list beneath it at parse time (a free selection).

A list of two is a stereo signal. `!` fans one signal into several that share a node. `nth` keeps just one branch of a multi-output word.

```
{ [ 110 220 ] saw mix 600 lpf 0.2 * out } synth .
```

An ordinary word spreads across a list without being asked, so `600 lpf` written after a two-channel signal is two filters running in step (*Placing sound in stereo* (3.7)). The words that hand *back* several channels are the exception. `svf` and `pan2` take a single channel in, and Cagire refuses a list fed to either at install rather than guessing at it. Reduce with `mix` or `nth` first, fan back out afterwards. A list may hold sixty-four channels.

4.1.4 Quotations

word	signature	what it does
`( … )`	`( -- )`	Record a quotation: the tokens stay unevaluated until a combinator runs them.
`times`	`( n -- … )`	Evaluate the preceding `( … )` quotation `n` times in sequence (`n` a constant 1..=1024).

A quotation holds tokens without running them, and `times` unrolls them. The idiom is a diffusion chain, one allpass typed once and built four times:

```
{ 8 impulse ( 0.03 0.5 allpass ) 4 times 0.3 * out }
synth .
```

4.1.5 Definitions & binding

word	signature	what it does
`: … ;`	`: name … ; ( -- )`	Define a word; its body expands inline wherever `name` is later used.
`~`	`~name ( -- sig )`	A `~`-prefixed definition is a value: its body runs once, every use shares the node.
`{ }`	`{ a b c } ( x y z -- )`	Bind the top signals to per-definition locals, read by name without stack juggling.
`as`	`as name ( sig -- sig )`	Bind the top signal to `name`; a later bare `name` reuses the same node.
`:>`	`:> name ( sig -- )`	Bind the top signal to a per-expansion local named by the next token, popping it; the one-name form of `{ … }`.

`:` pastes a recipe fresh at each use. `~` computes a value once and shares it. `as` names a live signal mid-chain. `{ }` names a definition's arguments. `:>` names one intermediate and takes it off the stack. *Names, locals & reuse* (3.5) works through all five with an example each.

4.1.6 Feedback, time & memory

word	signature	what it does
`'`	`name' ( -- sig )`	Read `name` one sample late; closes a feedback loop (mono).
`fb`	`fb name n ( -- )`	Declare a feedback latch `name` of width `n`, so `name'/as name` carry n channels.
`now`	`( -- sig )`	This instance's own sample clock: 0 when it starts, one step per sample.
`sr`	`( -- sig )`	The engine's sample rate, folded to a constant when the patch is compiled.
`buf`	`buf name secs ( -- )`	Size a named buffer to `secs` seconds.

A trailing quote reads a bound name one sample late, which is how a signal feeds itself without a graph cycle (*Channels & feedback* (3.6)). `buf` sizes the memory the record/play pair uses (*Buffers & sampling* (4.8)).

`now` counts samples for one running patch instance, not for the engine. It starts at 0 when the instance starts. For a voice source that is the note-on, and for an orbit effect it is the moment you set the patch. Two notes therefore agree on rate and never on phase. Read `now` as elapsed time inside this note, not as a wall clock the whole session shares.

`sr` is what turns that count into seconds. The rate is fixed for a patch's life, so the compiler folds it into a constant and `now sr /` costs nothing at runtime. The same figure spelled `now 48000 /` would run at the wrong speed on a 44.1 kHz device. Here a two-second sweep opens the filter across the note:

```
{ 110 saw  now sr / 0.5 * 0 1 clip  0 1 200 3000
linlin  0.7 moog 0.2 * out } synth .
```

4.1.7 The voice inputs

word	signature	what it does
`notefreq`	`( -- sig )`	The event's frequency in Hz, driven per sample; vibrato and glide come through.
`gate`	`( -- sig )`	1 while the note is held, 0 in release.
`vel`	`( -- sig )`	The note velocity, 0..1.
`bps`	`( -- sig )`	The transport tempo in beats per second (120 BPM reads as 2).
`bus`	`( lane -- sig )`	Engine control bus lane `lane`, a value from 0 to 1 that the host writes; the lane must be a constant.
`param`	`param name default ( -- )`	Declare a named control the pattern sets per note; a bare `name` reads it, the default holds until an event writes it.

The first three are what doux writes into every patch. Read them to follow the sequencer (*The voice contract* (2.3)). `bps` is the fourth. It and `bus` are the two values an effect patch may read as well as a source (*Limits* (4.10)). Feed it to `phasor` and you have exactly one ramp per beat, whatever the tempo does afterwards:

```
{ 110 saw  bps phasor bi 200 2000 range  lpf 0.2 *
out } synth .
```

Multiples follow from the same value: `bps 4 *` is one ramp per sixteenth, `bps 0.25 *` one per bar in four.

`bus` reads one of the engine's 128 control bus lanes, numbered 0 to 127. A lane holds a value from 0 to 1 that the sequencing side writes with `bus!` or routes from a controller with `cc>bus` and `osc>bus` (*Control buses* (3.18.6)). The lane must be a constant, so `3 bus` reads lane 3 and a computed lane is a compile error. The engine latches the lanes once per block, so the signal moves in steps

of one block. Like `bps`, a lane belongs to the engine and not to a note, so an effect patch may read it:

```
{ 110 saw  3 bus 0 1 200 4000 linlin lpf  0.2 * out }
synth .
```

`param` adds controls of your own. Declare `param cutoff 400` inside the braces, then set it from the pattern with the same word, value first. Up to sixteen per patch, each name declared once. *The voice contract* (2.3) covers what they carry, when they reset, and how they take modulation.

```
{ param cutoff 400 notefreq saw cutoff lpf 0.3 *
out } synth a2 note 2000 "cutoff" param .
```

4.1.8 Audio input

WORD	SIGNATURE	WHAT IT DOES
`in`	`( -- sig )`	Read audio input channel 0 (a mono patch is fed the downmix).
`input`	`( n -- sig )`	Read audio input channel `n` (a constant; `0 input` is `in`).
`inputs`	`( n -- sig×n )`	Read the first `n` input channels as one n-wide signal (`2 inputs` for stereo).

Reading any of these makes the patch an *effect*. It processes sound with `dsp` or `fx` instead of being played as one (*Patches as effects* (2.5)). A patch that reads only `in` declares one input channel and receives the downmix of whatever feeds it. `2 inputs` declares two and receives the pair untouched.

```
"break" snd { in 2000 lpf out } fx .
```

4.2 Oscillators

The periodic sources. Each takes a frequency in Hz (a constant, `notefreq`, or any signal) and leaves one signal on the stack. Cagire reads every input per sample, so you can modulate anything here, frequency included. All of these are safe at any pitch. The geomet-

ric shapes are band-limited (polyBLEP/polyBLAMP), and `blip` clamps its harmonics under Nyquist, so nothing aliases as you sweep upward. *Oscillators & noise* (3.1) plays them in context. This page lists them.

word	signature	what it does
`sine`	`( freq -- sig )`	Sine oscillator: a pure tone. Frequency in Hz.
`sinefb`	`( freq fb -- sig )`	Feedback sine: the previous sample bends the phase, adding harmonics as `fb` rises (0 is a pure sine).
`saw`	`( freq -- sig )`	Sawtooth oscillator, band-limited (polyBLEP). Frequency in Hz.
`pulse`	`( freq width -- sig )`	Pulse oscillator, band-limited (polyBLEP), variable `width` duty.
`tri`	`( freq -- sig )`	Triangle oscillator, band-limited (polyBLAMP). Frequency in Hz.
`varsaw`	`( freq width -- sig )`	Variable-slope saw-to-triangle, band-limited (polyBLAMP). `width` sets the peak position.
`blip`	`( freq nharm -- sig )`	Band-limited impulse train: `nharm` equal cosine harmonics of `freq`, clamped under Nyquist.

4.2.1 sine

A pure tone: one partial, nothing for a filter to remove. Below audio rate it doubles as an LFO. Here a 5 Hz sine wobbles the frequency of another for vibrato.

```
{ 440  5 sine 6 * +  sine 0.2 * out } synth .
```

4.2.2 sinefb

A sine that feeds its own last sample back into its phase. At `fb` 0 it is an ordinary `sine`. Raise `fb` and the feedback folds in harmonics, brightening the tone from pure through a reedy, brassy growl and

on toward noise near the top of its range. That is a full timbral sweep from a single control, so modulating `fb` is the point of the word. Here a slow sine drives it from clean to gritty and back:

```
{ 220  0.5 sine 0.5 * 0.5 +  sinefb 0.2 * out }
synth .
```

Keep `fb` in 0..pi. The word clamps it there, and all of the useful motion lives in that span.

4.2.3 saw, tri and varsaw

`saw` carries every harmonic, falling at 6 dB per octave. It is the subtractive starting point, and its brightness is what filters are for (*Patch filters* (4.4)). `tri` is the mellow shape, a triangle whose harmonics fall away quickly, so it sits under a mix without ever taking it over. `varsaw` morphs between the two as you move its `width`, and modulating that width slides the tone smoothly from bright to soft.

4.2.4 pulse

`width` is the duty cycle: 0.5 is a hollow square (odd harmonics only), and pushing it toward either edge thins the tone toward a nasal buzz. Modulating the width with a slow sine is classic pulse-width modulation, chorus-like movement from a single oscillator. The same modulator drives `varsaw`'s width just as well. Keep the width term inside 0..1. This one swings between 0.1 and 0.9.

```
{ 110  0.2 sine 0.4 * 0.5 +  pulse 0.2 * out }
synth .
```

4.2.5 blip

`nharm` cosine harmonics of `freq`, all at equal amplitude. 1 is a sine, and large counts approach a click train. Sweeping the count opens the spectrum like a filter, except the change is stepped, one harmonic at a time. Here the count moves between 2 and 42.

```
{ 220  2 sine 20 * 22 +  blip 0.3 * out } synth .
```

For pitch conversions such as `mtof`, see *Patch math* (4.9).

4.3 Noise & sources

The non-periodic sources. The colored noises take no input and leave one signal. The rate-driven ones take a density in events per second or a frequency in Hz. Cagire reads that rate per sample like everything else in a patch, so you can modulate the rate itself. The random-value words at the bottom draw once and hold, per note or per trigger. *Oscillators & noise* (3.1) plays them in context. This page lists them.

word	signature	what it does
`noise`	`( -- sig )`	White-noise source: full-scale, in [−1, 1).
`pink`	`( -- sig )`	Pink noise: equal power per octave (−3 dB/oct).
`brown`	`( -- sig )`	Brown noise: a bounded random walk (−6 dB/oct).
`clipnoise`	`( -- sig )`	Clipped white noise: randomly +1 or −1.
`dust`	`( density -- sig )`	Random impulses, unipolar [0, 1), `density` events per second.
`dust2`	`( density -- sig )`	Random impulses, bipolar [−1, 1), `density` events per second.
`noiseh`	`( freq -- sig )`	Stepped noise: holds a random value, jumping at `freq` Hz (sample-and-hold).
`noisei`	`( freq -- sig )`	Ramped noise: linearly interpolates between random values at `freq` Hz.
`noiseq`	`( freq -- sig )`	Smooth noise: quadratically interpolates between random values at `freq` Hz.
`rand`	`( lo hi -- sig )`	Random constant: a uniform draw in [lo, hi), held for the life of the note.

word	signature	what it does
`exprand`	`( lo hi -- sig )`	Random constant, biased toward `lo`; args must be positive.
`logrand`	`( lo hi -- sig )`	Random constant, biased toward `hi`; args must be positive.
`trand`	`( trig lo hi -- sig )`	Triggered random: holds a uniform draw, redrawn on each rising edge.
`texprand`	`( trig lo hi -- sig )`	Triggered random, biased toward `lo`; args must be positive.

The four noises differ only in spectral tilt. `noise` is flat and so sounds top-heavy. `pink` has equal power per *octave* and matches how you hear. `brown` tilts darker again into a rumble, and `clipnoise` is the harshest of the set. Reach for `pink` whenever noise is a texture in itself, and for `noise` when a filter is going to carve it into something else (*Patch filters* (4.4)). `dust` is not a texture at all but a stream of random single-sample triggers. `dust2` is its bipolar twin, audible as crackle in its own right at higher densities. *Envelopes & triggers* (4.7) covers what a trigger can drive.

4.3.1 noiseh, noisei and noiseq

Random values at a controllable rate: the random *modulators*. `noiseh` holds each value flat until the next, the classic sample-and-hold. `noisei` glides between them along a straight line. `noiseq` eases in and out of each along a smooth curve. `noiseq` is the softest of the three, and the one to reach for when a modulation should feel hand-drawn. Run any of them at audio rate and the modulator turns back into a rough noise source of its own.

Paired with `range` (*Patch math* (4.9)) these give the random-modulation recipes worth memorizing. `noiseh` + `range` is a stepped wander, what `jit` does outside the braces. `noisei` + `range` is a smooth one, a `drift` oscillation. `brown` + `range` is a slow drunken drift, `drunk`. Stepped, between 200 and 4000 Hz, twice a second:

```
{ 110 saw  2 noiseh 200 4000 range  lpf 0.3 * out }
synth .
```

4.3.2 rand, exprand and logrand

Where the noises move constantly, `rand` draws once per note and holds. Every note the pattern fires gets its own value and keeps it for the note's whole life. It is how a patch gets per-note character, a slightly different detune, cutoff, or decay on every hit, with nothing sequenced:

```
{ notefreq 0.99 1.01 rand * saw  400 2000 rand lpf
gate 0.01 0.3 ar *  0.3 * out } synth <c3 e3 g3>
note .
```

`exprand` draws the same way but biased toward `lo`, and `logrand` toward `hi`. Both need positive bounds. Prefer them whenever the value is a frequency or a time. A uniform draw over 200..4000 Hz lands in the top octave most of the time. An exponential one spreads evenly across the octaves.

```
{ 200 4000 exprand sine 0.2 * out } synth .
```

These are the same words the sequencing language has (*Inside and outside the braces* (2.4)). Outside the braces, `rand` rolls when the event fires. Inside, it rolls when the note starts.

4.3.3 trand and texprand

`trand` is a triggered `rand`. It holds its draw until a rising trigger edge, then draws again. Feed it any trigger (an `impulse`, a `clock`, the `gate` edge) and the value becomes a stepped random sequence in the range you asked for. It is a `noiseh` with explicit bounds and your own clock.

```
{ 8 impulse 200 800 trand sine 0.2 * out } synth .
```

`texprand` is its exponentially spread twin, the natural choice for a random decay or cutoff per hit:

```
{ 8 impulse dup 200 3200 texprand sine swap 0.05 perc
* 0.3 * out } synth .
```

For a random *choice* from a fixed list rather than a range, and for passing triggers through by probability, see `tchoose` and `coin` in *Envelopes & triggers* (4.7).

4.4 Patch filters

Filters carve the spectrum a source hands them. Every one takes its input signal first, parameters after (`sig cutoff lpf`, never the other way round). Cutoffs are in Hz, `res` runs 0 to 1, and like everything in a patch each parameter is read per sample, so a filter sweep is just an LFO on the cutoff. *Subtractive shaping* (3.2) plays them in context; this page lists them.

4.4.1 One-pole & utility

The gentle workhorses, one pole each, alongside a few utilities for slew and DC. None of these resonate; they tilt or ease a signal rather than carve it.

WORD	SIGNATURE	WHAT IT DOES
`lpf`	`( in cutoff -- sig )`	One-pole low-pass: attenuates above the cutoff.
`hpf`	`( in cutoff -- sig )`	One-pole high-pass, the residual of the low-pass; attenuates below the cutoff.
`apf`	`( in cutoff -- sig )`	First-order allpass: flat magnitude, frequency-dependent phase (phasers, diffusion).
`lag`	`( in time -- sig )`	One-pole slew: eases a signal toward its target over `time` seconds (portamento; 0 = passthrough).
`slew`	`( in up down -- sig )`	Slew limiter: the output chases the input, bounded to `up`/`down` units per second.
`dcblock`	`( in -- sig )`	DC blocker: removes the constant offset, leaving the audio band untouched.

4.4.2 Resonant two-pole

Add a `res` input and the two-pole designs grow a peak at the cutoff. The first four each keep a single response, while `svf` leaves all four taps at once for you to choose from with `nth`.

word	signature	what it does
`lpf2`	`( in cutoff res -- sig )`	Resonant two-pole low-pass (state-variable).
`hpf2`	`( in cutoff res -- sig )`	Resonant two-pole high-pass (state-variable).
`bpf`	`( in cutoff res -- sig )`	Resonant two-pole band-pass (state-variable).
`notch`	`( in cutoff res -- sig )`	Resonant two-pole band-reject / notch (state-variable).
`svf`	`( in cutoff res -- lp bp hp notch )`	State-variable core: leaves all four taps at once (low, band, high, notch); pick one with `nth`.

4.4.3 Character

Filters with a voice of their own, from the Moog ladder's squelch to the ringing resonators and the RBJ shelves and bells.

word	signature	what it does
`moog`	`( in cutoff res -- sig )`	Moog-style ladder low-pass: four cascaded poles with resonant feedback; self-oscillates as `res` nears 1.
`ringz`	`( in freq decay -- sig )`	Ringing resonator: strikes a damped sinusoid at `freq`, ringing out over `decay` seconds.
`modal`	`( in freq decay structure bright -- sig )`	Modal resonator: eight tuned modes rung by whatever excites them; `structure` morphs string to bar to bell, `bright` sets how long the upper modes last.
`reson`	`( in freq q -- sig )`	Resonant band-pass: a constant 0 dB peak at

word	signature	what it does
		`freq`, sharpness set by `q` (RBJ biquad).
`peak`	`( in freq gain q -- sig )`	Peaking EQ: boosts or cuts a band by `gain` dB at `freq`, width set by `q` (RBJ biquad).
`lowshelf`	`( in freq gain -- sig )`	Low shelf: lifts or dips everything below `freq` by `gain` dB (RBJ biquad, fixed slope).
`highshelf`	`( in freq gain -- sig )`	High shelf: lifts or dips everything above `freq` by `gain` dB (RBJ biquad, fixed slope).

4.4.4 lpf, lpf2 and moog

`lpf` is one pole, 6 dB per octave: a gentle slope rather than a wall, and the default first move on a saw.

```
{ 110 saw 800 lpf 0.3 * out } synth .
```

The two-pole `lpf2` adds `res`, a peak at the cutoff that turns a sweep into a vowel, and `moog` cascades four poles with feedback around the ladder: steeper, darker, and the classic squelch when the cutoff moves.

```
{ 110 saw  2 sine 200 2000 range  0.8 lpf2 0.3 *
out } synth .
```

One caution for every `res` input on this page: near `1` the two-pole designs ring hard, and `moog` self-oscillates outright. Drop the output gain before you explore up there.

4.4.5 bpf

Keeps only the band around the cutoff. On noise it is a pitch-maker: raise `res` to narrow the band and hiss becomes a breathy tone at the cutoff frequency.

```
{ noise 1000 0.8 bpf 0.4 * out } synth .
```

4.4.6 ringz and lag

`ringz` is a resonant body more than a filter: each impulse strikes a damped sinusoid at `freq` that rings out over `decay` seconds. Feed it clicks for mallets, short noise bursts for bowed textures.

```
{ 4 impulse 880 0.3 ringz 0.5 * out } synth .
```

`lag` eases any signal toward its target over `time` seconds, a low-pass for control values. Put it on `notefreq` and note steps become glides: portamento handled by the patch itself.

```
{ notefreq 0.08 lag saw 700 lpf 0.3 * out } synth <a1
c2 e2 a2> note .
```

4.5 Shaping, dynamics & pan

Once a source is filtered, these words shape what is left: saturating it, degrading it, riding its level, or placing it across the stereo field. Like every patch word they read their parameters per sample, so any amount here can itself be modulated. *Saturation & dynamics* (3.3) and *Placing sound in stereo* (3.7) play them in context; this page lists them.

4.5.1 Saturation

WORD	SIGNATURE	WHAT IT DOES
`tanh`	`( in -- sig )`	Hyperbolic-tangent soft saturation.
`distort`	`( in amount -- sig )`	`tanh` saturation driven by `amount` (0 = gentle).

Drive a signal past unity and `tanh` rounds the peaks instead of clipping them square: warmth, then grit as you push harder. `distort` is the same curve with a drive control up front, so the amount can be modulated like anything else.

```
{ 110 saw 0.6 distort 0.3 * out } synth .
```

4.5.2 Degradation

word	signature	what it does
`crush`	`( in bits -- sig )`	Bit crusher: quantizes to `bits` of amplitude resolution.
`decim`	`( in rate -- sig )`	Sample-rate decimator: holds the input, re-sampling `rate` times per second.

`crush` throws away amplitude resolution and `decim` throws away time resolution: the lo-fi pair, quantization noise from one, aliasing from the other. Both usually want to sit after the filter rather than before it.

```
{ 440 sine 4 crush 0.2 * out } synth .
```

4.5.3 Dynamics

word	signature	what it does
`follow`	`( in attack release -- sig )`	Amplitude follower: tracks |in| with separate attack/release smoothing.
`comp`	`( in thresh ratio attack release -- sig )`	Compressor: reduces gain by `ratio` while the level exceeds `thresh`.

`follow` traces a signal's loudness envelope and returns control, not audio; aim it at a filter cutoff and the patch responds to its own dynamics. `comp` is the finished compressor, evening out a signal that swings too wide.

```
{ 440 sine 0.8 *  0.3 4 0.01 0.1 comp  0.5 * out }
synth .
```

4.5.4 Stereo

word	signature	what it does
`pan2`	`( in pos -- l r )`	Equal-power panner: mono in, a left and a right channel out; `pos` runs −1 (left) to +1 (right).

word	signature	what it does
`splay`	`( chans -- l r )`	Spread a channel-list across the stereo field, equal-power left to right, level-compensated.

Both leave *two* values on the stack rather than one, so gather the pair in a channel list or a bare `out` keeps only the left (*Placing sound in stereo* (3.7)).

```
{ [ 440 sine 0.2 *  0.5 sine  pan2 ] out } synth .
```

`splay` is the many-to-two version: hand it a whole channel list and it fans the channels evenly across the field, the first hard left, the last hard right, the rest ranked between, then sums them back down to one pair. The level is compensated as the channel count grows, so a wide splay does not blow past a single voice's headroom.

```
{ [ [ 220 330 440 ] sine 0.2 * splay ] out } synth .
```

Spreading a voice across more than two speakers is a sequencing-side concern; the patch declares stereo and the engine's *Panning* (3.5) and *Multichannel output* (9.1) take it from there.

4.6 Delays & time

Delay-based effects hold the signal and replay it: an echo, a resonance, or a whole room, depending on how long the line is and how much of it feeds back. Each of these words claims real per-note memory, and the total is capped (*Limits* (4.10)). *Channels & feedback* (3.6) plays them in context. This page lists them.

4.6.1 The words

word	signature	what it does
`delay`	`( in time -- sig )`	Delay line: echoes the input `time` seconds later.
`comb`	`( in time fb -- sig )`	Feedback comb: an interpolated delay feeding itself back by `fb`.

word	signature	what it does
`allpass`	`( in time fb -- sig )`	Schroeder allpass: flat magnitude, dense `time`-spaced echoes.
`verb`	`( in mix room damp -- sig )`	Freeverb-style mono reverb: combs into series allpasses. `mix` blends dry→wet.

4.6.2 delay

A plain delay line. It outputs the signal `time` seconds late, the wet tap only, not mixed with the dry input. For the classic dry-plus-echo you split the signal and sum the delayed copy back yourself. One delay is a single repeat. Feeding the output back with `as/'` is what turns it into a train (*Channels & feedback* (3.6)).

```
{ 4 impulse 0.05 perc  440 sine *  dup  0.3 delay +
0.3 * out } synth .
```

4.6.3 comb

`comb` packages that feedback loop into one word. `fb` sets the tail length. At echo-scale times you hear distinct repeats. Shrink the time under ~20 ms and they fuse into a pitch, the Karplus-Strong string:

```
{ 8 impulse 0.05 0.7 comb 0.3 * out } synth .
```

4.6.4 allpass

Flat in magnitude. It colours phase, not tone. One is nearly inaudible. Chained, they smear a click into diffuse density, which is why a stack of them is the back half of a reverb. Build the chain with the `( … ) times` idiom (*Core words* (4.1)) rather than typing the same line four times.

4.6.5 verb

A complete reverb: eight damped feedback combs into four series allpasses. `mix` blends dry to wet, `room` sets the size, `damp` rolls off the tail's highs. Feed it something percussive to hear the space:

```
{ 2 impulse 0.05 perc 440 sine * 0.3 *  0.4 0.9 0.5
verb  0.5 * out } synth .
```

The voice already carries an orbit reverb and delay that wrap the patch (*The voice contract* (2.3)). Reach for these when the space must live *inside* the instrument, feeding back through a filter, tuned short as a resonator, or diffusing one layer before the rest.

4.7 Envelopes & triggers

An envelope shapes a signal over time. A trigger is the one-sample spike that starts it. Between them they give a patch its own sense of time, an attack, a decay, a rhythm, independent of the sequencer or in step with it. *Modulation & envelopes* (3.4) puts them in musical context. This page lists them.

4.7.1 Triggers & clocks

word	signature	what it does
`impulse`	`( rate -- sig )`	Clock: fires a one-sample impulse at `rate` Hz.
`clock`	`( rate -- sig )`	Clock locked to the global sample clock: fires at `rate` Hz with no drift.
`trig`	`( in -- sig )`	Rising-edge detector: fires when the input crosses from ≤0 to >0.
`coin`	`( trig prob -- sig )`	Probability gate: passes each trigger with probability `prob`, swallows it otherwise.
`phasor`	`( freq -- sig )`	Ramp oscillator: a 0→1 phase ramp at `freq` Hz; the precise phase source.

`impulse` is the workhorse pulse. `clock` is its drift-free cousin, so related rates stay sample-locked to each other. `trig` makes a trigger out of any signal's rising edge: the gate's edge, an LFO crossing zero, a comparison going true.

```
{ 2 sine trig 0.1 perc 440 sine * 0.3 * out } synth .
```

`coin` flips a weighted coin on every trigger that reaches it and passes only the winners, swallowing the rest, so a steady clock comes out the other side as a generative rhythm:

```
{ 8 clock 0.6 coin 0.04 perc noise * 8000 hpf 0.3 *
out } synth .
```

4.7.2 Envelopes

WORD	SIGNATURE	WHAT IT DOES
`perc`	`( trig time -- sig )`	Percussive envelope: linear decay 1→0 over `time`, reset by a trigger.
`ar`	`( trig attack release -- sig )`	Attack-release: rises then falls; a retrigger rises from the current level.
`adsr`	`( gate attack decay sustain release -- sig )`	Gated ADSR: sustains while the gate is held, releases when it drops.
`decay`	`( in time -- sig )`	Exponential decay: each input impulse rings down 60 dB over `time`.
`line`	`( trig start end dur -- sig )`	Linear line from `start` to `end` over `dur`, (re)started by a trigger; holds `end`.
`xline`	`( trig start end dur -- sig )`	Exponential line from `start` to `end` (same sign, nonzero); the pitch-drop curve.
`linseg`	`( trig [ … ] -- sig )`	Multi-segment envelope: ramps through a `[ l0 t1 l1 … ]` list per trigger.

`perc` is the quickest way to make a hit: a trigger in, a falling shape out, ready to multiply a tone. The two trigger envelopes differ on retrigger, and that is the difference worth remembering. `ar` rises from wherever it currently is, and `perc` restarts from 1. For a note that holds and releases with the key, `adsr` reads `gate` directly (*The voice contract* (2.3)).

`line` and `xline` travel between two levels you name, and they work as well on a pitch or a cutoff as on loudness. Until the first

trigger they rest at `start`, and after the ramp they hold `end`. `xline` takes the exponential path, the right curve for a falling pitch. One of them driving a sine is the kick-drum recipe (*Putting it together* (3.9)).

`linseg` is the general case. Write an arbitrary breakpoint shape as a list that starts with a level and then alternates time and level, `[ l0 t1 l1 t2 l2 … ]`. That makes the list an odd length, at least three long, and Cagire refuses any other shape at install. Each time is the seconds spent travelling to the level after it, and the last level holds until the next trigger. Three segments give a swell, a dip, and a long fade, aimed here at a filter rather than at loudness:

```
{ notefreq saw  4 impulse [ 200 0.05 4000 0.1 800 0.6
300 ] linseg  lpf  0.25 * out } synth a1 note .
```

To bend any of these linear shapes without breakpoints, `curve` in *Patch math* (4.9) re-curves a 0..1 signal.

4.7.3 Hold & step

WORD	SIGNATURE	WHAT IT DOES
`seq`	`( trig [ … ] -- sig )`	Step sequencer: holds a value from the list, advancing one step per trigger.
`tchoose`	`( trig [ … ] -- sig )`	Random choice: holds a uniformly picked value from the list, repicking per trigger.
`latch`	`( in trig -- sig )`	Sample & hold: captures the input on each rising edge and holds it.

`seq` walks a list of values, one step per trigger: a melody or a rhythm of parameters held inside the patch. Paired with `mtof` (*Patch math* (4.9)) it is a self-contained arpeggio. Swap `tchoose` in and the order goes away, because it picks any value from the list on every trigger:

```
{ 4 impulse [ 60 63 67 70 ] seq mtof saw 600 lpf 0.2
* out } synth .
```

`latch` freezes whatever is at its input the instant a trigger arrives. It is the same sample-and-hold as `noiseh`, but on a signal you choose. For random values in a *range* rather than from a list, `rand` and `trand` are in *Noise & sources* (4.3).

4.8 Buffers & sampling

A buffer is a strip of per-note memory a patch can write into and read back: the basis of tape-style looping and short delays you address by name. Three words cooperate: `buf` sizes the memory, `record` writes to it, `play` reads from it.

4.8.1 The words

WORD	SIGNATURE	WHAT IT DOES
`buf`	`buf name secs ( -- )`	Size a named buffer to `secs` seconds (the next two tokens are name and length).
`record`	`record name ( in -- sig )`	Record the input into buffer `name` (looping), passing the signal through.
`play`	`play name ( -- sig )`	Play back buffer `name`, looping at the play head.

Each of the three names its buffer with the *next* token, not a value on the stack: `buf a 2`, `record a`, `play a` all refer to buffer `a`. A buffer holds one channel, so `record` wants a mono signal and refuses a channel list; `mix` one down first if what you are tapping is wide. `record` returns its input untouched, so you can keep using the live signal after tapping it into the buffer, or `drop` it:

```
{ buf a 2  noise 0.2 * record a  drop  play a  out }
synth .
```

Here two seconds of noise are written into `a` while, on the same pass, `a` is played back and sent out: a looping sampler in three words.

4.8.2 What a buffer is not

A buffer belongs to the note, not the session. Each triggered note gets its own fresh instance of the patch, and so its own empty

buffers, so a buffer cannot carry audio from one note to the next, and there is no shared, named sample store here (loading files is the sequencing side's job, through the ordinary sample sources). Buffer memory counts against the per-instance cap of 4 MiB, shared with every delay line in the patch; size buffers to what the note needs (*Limits* (4.10)).

4.9 Patch math

Numbers and signals are the same thing in a patch, so arithmetic *is* signal processing: add two oscillators and they mix, multiply them and they ring-modulate. Beyond the four basic operators, this page collects the rounding, mapping, and comparison words, the glue that reshapes one signal into the range another one needs. *Modulation & envelopes* (3.4) puts the mapping words to work; this page lists them.

4.9.1 Writing numbers

A number in a patch is anything Arf can read as one, which is a little more than you might expect. Decimals and scientific notation work (`1.5e-2`), an underscore may sit anywhere inside a numeral to group digits (`44_100` reads as forty-four thousand one hundred), and a single slash between two numerals is a ratio the compiler folds on the spot, so `2/3` is one token worth `0.667` and `1/8` is a useful way to write an eighth. Two slashes, or a word on either side, is not a number: `1/2/3` and `1/x` fall through to word lookup and come back as unknown words.

The spacing is what tells the two apart, and it matters. `2/3` is a single value pushed on the stack, while `2 / 3` is three tokens: push `2`, divide (consuming whatever is under it), push `3`. Keep ratios tight and division spaced.

4.9.2 Arithmetic

WORD	SIGNATURE	WHAT IT DOES
`+`	`( a b -- sig )`	Add two signals.

word	signature	what it does
`-`	`( a b -- sig )`	Subtract the second signal from the first.
`*`	`( a b -- sig )`	Multiply two signals (gain, ring/amplitude modulation).
`/`	`( a b -- sig )`	Divide the first by the second (IEEE: ÷0 is ±inf).
`mix`	`( in... -- sig )`	Sum a channel-list down to one signal.
`select`	`( index [ … ] -- sig )`	Pick one signal from a channel-list by index (floored, clamped).

`*` is the busiest word in the language: a constant is gain, a slow signal is tremolo, an audio-rate signal is ring modulation. `mix` and `select` work on channel lists, shown alongside `[ … ]` in *Core words* (4.1).

4.9.3 Rounding & wrapping

word	signature	what it does
`floor`	`( x -- sig )`	Round toward −∞.
`ceil`	`( x -- sig )`	Round toward +∞.
`round`	`( x -- sig )`	Round to nearest integer (ties to even).
`trunc`	`( x -- sig )`	Round toward zero (drop the fraction).
`mod`	`( a b -- sig )`	Euclidean remainder a mod b (never negative for b>0).
`wrap`	`( x lo hi -- sig )`	Wrap x into the range [lo, hi).
`fold`	`( x lo hi -- sig )`	Reflect x into [lo, hi] (wavefolder).

`fold` is the musical one: reflecting a driven signal back on itself past the limits adds harmonics without ever clipping, the wavefolder's fizz.

```
{ 220 sine 3 *  -0.4 0.4 fold  0.5 * out } synth .
```

4.9.4 Sign, magnitude & bounds

word	signature	what it does
`abs`	`( x -- sig )`	Absolute value (magnitude).
`neg`	`( x -- sig )`	Negate a signal (flip its sign).
`sign`	`( x -- sig )`	Sign of a signal: −1, 0, or +1.
`recip`	`( x -- sig )`	Reciprocal 1/x (IEEE: 1/0 is ±inf).
`min`	`( a b -- sig )`	The smaller of two signals.
`max`	`( a b -- sig )`	The larger of two signals.
`clip`	`( x lo hi -- sig )`	Clamp a signal into [lo, hi].

`min`, `max`, and `clip` double as hard shapers, the square-edged counterpart to `fold` above: `clip` squares a driven sine off into buzz, `0 max` half-wave rectifies. Keep the result bounded before `out`.

4.9.5 Transcendental

word	signature	what it does
`sqrt`	`( x -- sig )`	Square root (IEEE: √negative is NaN).
`pow`	`( x y -- sig )`	Raise x to the power y.
`exp`	`( x -- sig )`	Exponential e^x.
`log log2 log10`	`( x -- sig )`	Natural, base-2, base-10 logarithm.
`sin cos tan`	`( x -- sig )`	Trigonometric functions of x in radians.
`atan`	`( x -- sig )`	Arctangent (bounded ±π/2).

These operate on a signal's *value*, not its phase: `sin` is the arithmetic function (radians in), distinct from the `sine` oscillator. `exp` on a falling ramp bends a linear decay into a natural one, though `curve` below does the same job with one control.

4.9.6 Range mapping

word	signature	what it does
`uni`	`( x -- sig )`	Bipolar −1..1 to unipolar 0..1.

word	signature	what it does
`bi`	`( x -- sig )`	Unipolar 0..1 to bipolar −1..1.
`range`	`( x lo hi -- sig )`	Map a bipolar −1..1 signal onto [lo, hi].
`linlin`	`( x ilo ihi olo ohi -- sig )`	Linearly map x from [ilo,ihi] to [olo,ohi].
`linexp`	`( x ilo ihi olo ohi -- sig )`	Map x to an exponential [olo,ohi]; musical for pitch and cutoff.
`lerp`	`( a b t -- sig )`	Linear interpolate a→b by t.
`xfade`	`( a b t -- sig )`	Equal-power crossfade a→b by t (0..1).
`curve`	`( x amount -- sig )`	Bend a 0..1 signal: `amount` 0 is linear, positive starts slow, negative starts fast.

The everyday problem is a modulator that lands in the wrong range. A bare oscillator swings −1..1, and `range` drops it straight onto a span you name at both ends:

```
{ 220 sine  1 sine 0 0.4 range  * out } synth .
```

`uni` folds a modulator positive first when the target cannot go negative, and `bi` is its inverse. `linlin` names both ends of the input span instead of assuming −1..1, and `linexp` maps the same way but exponentially, the right curve for frequency, where equal steps should sound equal.

`curve` reshapes the ramps themselves: feed it anything that runs 0..1 and one number bends the path, positive easing in like an exponential, negative snapping early like a logarithm.

```
{ 4 impulse  0.3 perc  3 curve 440 sine *  0.3 *
out } synth .
```

4.9.7 Pitch & level

word	signature	what it does
`mtof`	`( m -- sig )`	MIDI note number to frequency in Hz (69 = 440).

word	signature	what it does
`ftom`	`( hz -- sig )`	Frequency in Hz to MIDI note number (440 = 69).
`dbamp`	`( db -- sig )`	Decibels to linear amplitude (0 dB = 1).
`ampdb`	`( amp -- sig )`	Linear amplitude to decibels (1 = 0 dB).

`mtof` turns note numbers into pitch, so a `seq` of integers becomes a tune in equal temperament (*Envelopes & triggers* (4.7)); `dbamp` lets you set levels in the units a mixer uses.

```
{ 60 mtof sine 0.2 * out } synth .
```

4.9.8 Comparison

word	signature	what it does
`< >`	`( a b -- sig )`	1 if the test holds, else 0.
`<= >=`	`( a b -- sig )`	1 if the test holds, else 0.
`== !=`	`( a b -- sig )`	1 if equal / not equal, else 0.

Each returns `1` or `0`, a gate signal. Multiply a tone by one and it sounds only where the test is true. A phasor compared against a threshold is a square wave whose width you dial with the number:

```
{ 1 phasor 0.5 < 440 sine *  0.2 * out } synth .
```

4.10 Limits

The patch language runs inside the doux engine, and the engine decides what a patch is allowed to be. This page gathers what a patch cannot do. It covers the roles it cannot mix, the shapes it cannot take, the resources it cannot exceed, and the words from standalone Arf that the embedded engine does not carry. Cagire checks every rule below at install, and the message names the word or the number that broke it.

4.10.1 Rules

RULE	DETAIL
A patch is a source or an effect, never both	Reading `in`, `input`, or `inputs` makes it an effect: it installs only with `dsp` or `fx`, and never plays as a sound (*Patches as effects* (2.5)).
An effect reads no per-note controls	`gate`, `notefreq`, `vel`, and a `param` of its own are all refused alongside `in`. Only `bps`, the transport tempo, and `bus`, an engine control bus lane, are allowed.
Every patch reaches `out`	There is no default routing, so a patch that never reaches `out` is a compile error, on purpose.
`out` is mono or stereo, nothing else	One signal or a two-channel list. A three-channel list is refused, because a doux voice is mono or stereo (*The voice contract* (2.3)). Spreading wider is the sequencing side's job, through *Panning* (3.5) and *Multichannel output* (9.1).
Audio input is at most stereo	`3 inputs` is rejected: the buses are two channels wide.
A patch does not set its own polyphony	doux owns polyphony, and every triggered note already gets its own fresh instance, so there is no `voices` word.
`off` is a reserved name	`"off" dsp` and `"off" fx` are how you clear a slot (the full naming rules are in *Named patches* (2.2)).
Multi-output words take one channel in	`svf` and `pan2` are the two, and a channel list fed to either is rejected rather than guessed at (*Core words* (4.1)).
A definition cannot call itself	Bodies are pasted inline, directly or through another definition, so recursion has nothing to bottom out on. The expander stops at sixty-four levels deep.

rule	detail
A `~value` leaves exactly one signal	The whole point of the tilde is that every use reads the same single node (*Names, locals & reuse* (3.5)).

4.10.2 Resource caps

Six caps apply, all sized generously.

cap	limit
Instances	The Voices setting, sixty-four by default. A trigger past it is dropped of one while the others ring (*Inline synths* (2.1)).
Delay and buffer memory per instance	4 MiB, which is sixteen delay lines at their default length.
Graph complexity	4096, where a bare sine weighs about a dozen.
Channel list width	64 channels.
`( … )` `n times` repeats	1024.
`param` declarations per patch	16.

4.10.3 Absent words

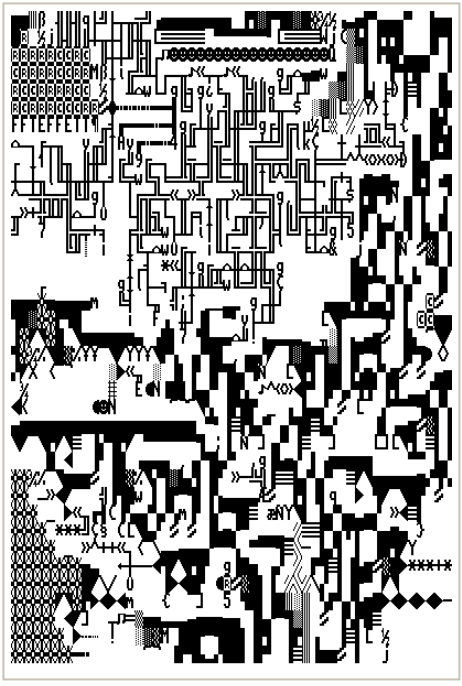
Standalone Arf has session directives such as `stop` and `fade`, along with its own slot management. The embedded engine has no session, so those words do not exist here. In Cagire a pattern is the session. The rule is general. If a word is not listed in the F1 Dictionary under the Arf tab, the compiler does not know it, and a patch that names it fails to compile like any other unknown word.

4.10.4 Where a refusal shows up

None of this fails silently. A patch that will not compile or will not install reports where you evaluated the script that carried it. That is the step's own error line if it came from a step, or the prelude's status message if it came from a prelude (*Error handling* (2.10)). The message repeats the engine's own words, so `` `param` needs a name and a constant default `` arrives as written.

A failure is never cached, either. Cagire remembers successful installs, so it skips an unchanged patch on re-evaluation. It retries a broken one every time the script runs. That is what keeps the error in front of you until you fix it, rather than showing it one time only.

Appendices



Appendix A

Is Cagire a Forth?

Yes and no. Cagire's language is built on Forth's foundations: a data stack, postfix notation, and words you define yourself. If you have written Forth before, the core of the language carries over unchanged. But it is a Forth bent toward making music, and a few things differ from the classic language. The larger additions like native numbers and ratios, quotations, variables, the sequencer context, probability, polyphony each have their own page in *The language* (III). This is the short list of surprises. I am definitely not a Forth specialist nor a computer scientist. Programming languages draw my interest for sure. However, I am only interested in programming languages as long as they allow me to write and think about music.

A.1 Comments

Classic Forth writes comments in parentheses, `( like this )`. In Cagire parentheses build quotations, blocks of deferred code, so comments use a double semicolon instead, running to the end of the line:

```
sine snd . ;; a plain sine
```

For a comment that runs across several lines, Cagire stays in the semicolon family: `;*` opens and `*;` closes, and everything in between is ignored.

```
;* this comment
   spans two lines *;
sine snd .
```

A.2 Unknown words

Classic Forth aborts the moment it meets a word it does not know. Cagire pushes it onto the stack as a string instead:

```
sine print
```

This prints `sine`. The name you have typed since the first page of this manual was never a defined word, only a string that `snd` reads as the name of a source. Sample names work the same way, which is why they read bare. The one exception is a longer bareword sitting one letter away from a real word: Cagire assumes a typo and suggests the word you meant instead of quietly pushing a string. The rule has its home in *Words & comments* (2.3).

A.3 Programs for sound generation

This is the part with no Forth equivalent. A classic Forth program prints text or performs various side effects that help you to build a program. A Cagire program builds a sound and the entire language is built around this. The word `snd` and the parameter words accumulate in a sound register, and nothing reaches the audio engine until you emit with `..` That last word is the sharpest clash for a Forth eye: Forth's `.` pops a number and prints it, Cagire's `.` performs. Printing is `print`. The register is the subject of *The sound register* (2.5).

A.4 Familiar names do different jobs

Two more classic names survive with new meanings. `loop` is not the tail of a counted loop, it fits a sample across a number of steps, and `begin` does not open one either, it sets where playback starts inside a sample. Both live in *Looping samples* (5.3). The rest of the loop vocabulary (`do`, `until`, `while`, `repeat`) is simply gone: counted repetition is a quotation handed to a word, `( ... ) 3 times`, and the family is covered in *Iteration* (3.10.3).

A.5 Conditionals come in two forms

Cagire keeps the classic `if ... else ... then`. However, it also offers quotation-based conditionals that read more naturally in a postfix line. `?` runs a quotation when a condition is true, `!?` when it is false, and `ifelse` chooses between two:

```
( c5 note ) ( c3 note ) 1 ifelse saw snd .
```

The condition `1` picks the first quotation, so this plays `c5`. These are covered in *Combinators* (3.10) and *Control flow* (3.9).

A.6 There is no memory

Forth is a language about memory: `@` and `!` move cells around, `allot` claims space, `create` and `does>` build words out of it. Cagire has none of that. There are no addresses, no cells, and nothing to allot. What survives is the pair of sigils, glued to a name: `!freq` stores a value into a named box and `@freq` reads it back:

```
440 !freq @freq print
```

This prints `440`. There is no `variable` declaration, and no `constant` either: a word that pushes a value is just `: answer 42 ;`. The boxes are covered in *Variables* (3.7).

A.7 There is no return stack

`>r`, `r>` and `r@` do not exist, so the classic trick of tucking a value away on the return stack has no direct spelling. Quotations take over the job: `dip` sets the top value aside, runs a block on what remains, and puts it back.

```
1 2 5 ( + ) dip print print
```

Where Forth writes `>r + r>`, Cagire writes `( + ) dip`: the `5` steps aside, `1` and `2` become `3`, and the `5` returns on top, so this prints `5` then `3`. Its sibling `keep` runs a block on the top value and hands you the original back as well. Both are covered in *Quotations & combinators* (2.7).

A.8 Built-ins cannot be redefined

In many Forths you can redefine anything, including the core words. Cagire looks up built-in words before your own, so redefining one is quietly pointless: the definition exists, but lookups never get that far:

```
: dup 99 ; 5 dup print
```

The built-in `dup` keeps its old job. This prints `5`, not `99`. Your own words you can redefine freely, and you can even redefine a number.

```
: 2 4 ; 2 print
```

After this, `2` pushes `4`. That is the reverse of classic Forth, where nothing is protected. Definitions and redefinition are covered in *The dictionary* (2.6).

A.9 Named arguments use a head block

Classic Forth grew named arguments twice: `LOCALS|`, notorious for binding names in reverse order, and the newer `{: a b :}` header that reads in call order. Cagire keeps the modern shape with lighter braces: `: name { a b } … ;` names the arguments right after the word's name, bound so the first name takes the value you pushed first.

```
: bell { bright }  sine snd  bright fmfb . ;
0.9 bell
```

A.10 So, is it a Forth?

It is a Forth in spirit: the stack, postfix order, definitions, the same economy of syntax. It is not a standard Forth, and it is not trying to be: no memory, no return stack, and familiar words retuned toward sound. In temperament it sits closer to Joy or Factor, a small concatenative language shaped for live musical performance.

Appendix B

General MIDI presets

Cagire gives many General MIDI programs a named source word so you do not have to remember program numbers. Each word starts with `gm` and plays like any other source, so `gmpiano` is the grand piano and `gmflute` is the flute. Every program is reachable by number as well, from `gm0` to `gm127`, whether or not it carries a name. These words sound from a standing start, since Cagire ships with a soundfont of its own, and they play whatever is loaded: Cagire keeps a single soundfont active at a time and hands the built-in one over to any `.sf2` file it finds in a sample folder. For playing soundfonts in full see *Soundfonts* (2.9), and for getting the `.sf2` file loaded see *Loading soundfonts* (5.4). This appendix lists all 128 programs.

B.1 Preset reference

Each row is one General MIDI program. The instrument is the sound the standard puts at that number, and so what a soundfont built to the standard plays there. The word is what you write in a script: a name where Cagire has one, and the plain `gm<number>` form otherwise. The aliases play the same voice as the main word. The family is the rough timbral group Cagire files the program under. The drum kit is the exception to all of this. It sits on a bank of its own rather than on a program number, so it is reached with `gmdrums` (also `gmdrum` and `gmpercussion`) and is left out of the list below.

Program	Instrument	Word	Aliases	Family
0	Acoustic Grand Piano	`gmpiano`	`gmgrandpiano`	Piano

Program	Instrument	Wordnet	Aliases	Family
1	Bright Acoustic Piano	gm1	gmbrightpiano	Piano
2	Electric Grand Piano	gm2		Piano
3	Honky-tonk Piano	gm3		Piano
4	Electric Piano 1	gm4	gmelectricpiano, gmrhodes	Piano
5	Electric Piano 2	gm5		Piano
6	Harpsichord	gm6	gmharpsichord	Piano
7	Clavinet	gm7	gmclavinet	Piano
8	Celesta	gm8	gmcelesta	Chromatic Percussion
9	Glockenspiel	gm9	gmglockenspiel	Chromatic Percussion
10	Music Box	gm10	gmmusicbox	Chromatic Percussion
11	Vibraphone	gm11	gmvibraphone	Chromatic Percussion
12	Marimba	gm12	gmmarimba	Chromatic Percussion
13	Xylophone	gm13	gmxylophone	Chromatic Percussion
14	Tubular Bells	gm14	gmtubularbells	Chromatic Percussion
15	Dulcimer	gm15		Chromatic Percussion
16	Drawbar Organ	gm16	gmorgan	Organ
17	Percussive Organ	gm17		Organ
18	Rock Organ	gm18		Organ

Program	Instrument	Aliases	Family
19	Church Organ	gmchurchorgan	Organ
20	Reed Organ	gm20	Organ
21	Accordion	gmaccordion	Organ
22	Harmonica	gmharmonica	Organ
23	Tango Accordion	gm23	Organ
24	Acoustic Guitar (nylon)	gmguitar	gmnylon, gmnylonguitar	Guitar
25	Acoustic Guitar (steel)	gmsteelguitar	gmsteel	Guitar
26	Electric Guitar (jazz)	gmjazzguitar	Guitar
27	Electric Guitar (clean)	gmcleangt	gmclean	Guitar
28	Electric Guitar (muted)	gm28	Guitar
29	Overdrive Guitar	gmoverdrive	gmoverdriven	Guitar
30	Distortion Guitar	gmdistgt	gmdistortionguitar	Guitar
31	Guitar Harmonics	gm31	Guitar
32	Acoustic Bass	gm32	Bass
33	Electric Bass (finger)	gmbass	gmfingerbass	Bass
34	Electric Bass (pick)	gmpickbass	Bass
35	Fretless Bass	gmfretless	Bass

Program	Instrument	Word	Aliases	Family
36	Slap Bass 1	gmslapbass	gmslap	Bass
37	Slap Bass 2	gm37		Bass
38	Synth Bass 1	gmsynthbass		Bass
39	Synth Bass 2	gm39		Bass
40	Violin	gmviolin		Strings
41	Viola	gmviola		Strings
42	Cello	gmcello		Strings
43	Contrabass	gmcontrabass		Strings
44	Tremolo Strings	gm44		Strings
45	Pizzicato Strings	gmpizzicato	gmpizz	Strings
46	Orchestral Harp	gmharp		Strings
47	Timpani	gmtimpani		Strings
48	String Ensemble 1	gmstrings	gmensemble	Ensemble
49	String Ensemble 2	gmslowstrings		Ensemble
50	Synth Strings 1	gm50		Ensemble
51	Synth Strings 2	gm51		Ensemble
52	Choir Aahs	gmchoir		Ensemble
53	Voice Oohs	gm53		Ensemble

Program	Instrument	Aliases	Family
54	Synth Voice	gm54	Ensemble
55	Orchestra Hit	gm55	Ensemble
56	Trumpet	gmtrumpet	Brass
57	Trombone	gmtrombone	Brass
58	Tuba	gmtuba	Brass
59	Muted Trumpet	gm59	Brass
60	French Horn	gmfrenchhorn	Brass
61	Brass Section	gmbrass	Brass
62	Synth Brass 1	gm62	Brass
63	Synth Brass 2	gm63	Brass
64	Soprano Sax	gmsopranosax	Reed
65	Alto Sax	gmaltosax	gmalto	Reed
66	Tenor Sax	gmtenorsax	gmtenor	Reed
67	Baritone Sax	gmbarisax	gmbari	Reed
68	Oboe	gmoboe	Reed
69	English Horn	gm69	Reed
70	Bassoon	gmbassoon	Reed
71	Clarinet	gmclarinet	Reed
72	Piccolo	gmpiccolo	Pipe
73	Flute	gmflute	Pipe
74	Recorder	gmrecorder	Pipe
75	Pan Flute	gmpanflute	gmpan	Pipe

Program	Instrument	Aliases	Family
76	Blown Bottle	`gm76`	Pipe
77	Shakuhachi	`gm77`	Pipe
78	Whistle	`gm78`	Pipe
79	Ocarina	`gmwhistle`	Pipe
80	Lead 1 (square)	`gmocarina`	Synth
81	Lead 2 (sawtooth)	`gmlead`	`gmsquarelead`	Synth
82	Lead 3 (calliope)	`gmsawlead`	`gmsawsynth`	Synth
83	Lead 4 (chiff)	`gm83`	Synth
84	Lead 5 (charang)	`gm84`	Synth
85	Lead 6 (voice)	`gm85`	Synth
86	Lead 7 (fifths)	`gm86`	Synth
87	Lead 8 (bass and lead)	`gm87`	Synth
88	Pad 1 (new age)	`gm88`	Synth
89	Pad 2 (warm)	`gmpad`	`gmnewage`	Synth
90	Pad 3 (polysynth)	`gmwarmpad`	`gmwarm`	Synth

Program	Instrument	Aliases	Family
91	Pad 4 (choir)	gmpolysynth	Synth
92	Pad 5 (bowed)	gm92	Synth
93	Pad 6 (metallic)	gm93	Synth
94	Pad 7 (halo)	gm94	Synth
95	Pad 8 (sweep)	gm95	Synth
96	FX 1 (rain)	gm96	SFX
97	FX 2 (soundtrack)	gm97	SFX
98	FX 3 (crystal)	gm98	SFX
99	FX 4 (atmosphere)	gm99	SFX
100	FX 5 (brightness)	gm100	SFX
101	FX 6 (goblins)	gm101	SFX
102	FX 7 (echoes)	gm102	SFX
103	FX 8 (sci-fi)	gm103	SFX
104	Sitar	gmsitar	World

Program	Instrument	Words	Aliases	Family
105	Banjo	gm banjo		World
106	Shamisen	gm 106		World
107	Koto	gm 107		World
108	Kalimba	gm kalimba		World
109	Bagpipe	gm 109		World
110	Fiddle	gm 110		World
111	Shanai	gm 111		World
112	Tinkle Bell	gm 112		Percussion
113	Agogo	gm 113		Percussion
114	Steel Drums	gm steeldrum		Percussion
115	Woodblock	gm 115		Percussion
116	Taiko Drum	gm 116		Percussion
117	Melodic Tom	gm 117		Percussion
118	Synth Drum	gm 118		Percussion
119	Reverse Cymbal	gm 119		Percussion
120	Guitar Fret Noise	gm 120		Other
121	Breath Noise	gm 121		Other
122	Seashore	gm 122		Other
123	Bird Tweet	gm 123		Other
124	Telephone Ring	gm 124		Other
125	Helicopter	gm 125		Other
126	Applause	gm 126		Other
127	Gunshot	gm 127		Other

Seven names, from `gmwhistle` to `gmpolysynth`, currently select the program one above the instrument they are named for. That is a known mismatch in the engine's preset table, and the rows above are what those words do today, not what they should do. Reach for the plain `gm<number>` form when you want an exact program from that stretch.

Appendix C

Troubleshooting

When something breaks mid-set, you had better be ready to fix it. A silent Cagire is a frustrating Cagire, so this page goes symptom by symptom, in the order you are likely to meet them. Almost everything here happens in the Engine view (F6), so start there when sound misbehaves. Look outside Cagire as well. An audio device, a driver, a firewall or a system setting is behind a good share of the symptoms below, and each section says when that is the likely cause.

C.1 macOS says Cagire is damaged and cannot be opened

Cagire is not damaged. The macOS builds carry my signature rather than an Apple one, and macOS refuses to open a downloaded application that Apple has not signed, so the message names the wrong problem. Clear the quarantine flag from a Terminal, then open Cagire the usual way.

```
xattr -dr com.apple.quarantine /Applications/
Cagire.app
```

Point the same command at the file you unzipped if you took the terminal build rather than the app bundle.

C.2 Windows says it protected your PC

Cagire is not dangerous. Windows raises this panel for any program that does not carry a signing certificate it recognises, and Cagire does not carry one. The only button on show says Don't run, which makes the panel look like a dead end. Click More info, then Run anyway. Windows stops asking once it has seen the file.

C.3 Cagire does not play any sound

Open the Audio tab of the Engine view and press `t` to play a short test tone. If you hear the tone, the device path is fine and the silence is musical. Playback may be stopped, which the transport buttons at the top left will tell you at a glance (`F9` starts it again from any view), the pattern muted or another one soloed (`M` in the Sequencer or Patterns view clears every mute, `X` every solo), or the steps themselves quiet because of a programming error. If you do not hear the tone, the wrong output device is selected. Pick the right one in the Output column with `↑`/`↓` and `Enter`, then press `R` to restart the stream. One more thing to check before blaming the device: if **Patching** reads *Manual*, Cagire has opened its ports and deliberately wired nothing, so silence is the correct behaviour until you connect them in your patchbay. Set it back to *Auto* if that was not what you meant. See *Audio output* (6.2).

C.4 The Engine view shows an error

The stream failed to open, so the engine is not running at all. The red ERROR block at the top of the metrics spells out what went wrong, usually a device that vanished or refused its settings. Pick another output device in the Audio tab and press `R` to try again. Until the stream is back, `{ }` patches refuse to install too, with *no audio stream running*. See *Audio output* (6.2).

C.5 Crackles and dropouts

The buffer is too small for your machine. In the Audio tab, raise Buffer a notch. It doubles each step, and the tab shows the latency each size works out to. Press `R` to restart the engine with the new settings. If your driver reads *Host managed*, the size is decided outside Cagire, in your system's audio settings.

C.6 The Load meter runs red

Crackles that survive a bigger buffer mean the machine cannot compute the sound in time. Load in the Metrics column shows how much of each audio deadline the engine spends, yellow from half, red near the limit, and Voices right below counts what you are burning. Raise DSP Block in the Audio tab, lower the Voices ceiling, or thin out the heaviest patches. See *Audio output* (6.2) and *Limits* (4.10).

C.7 The audio device disappeared

An interface you unplug mid-session does not come back on its own. Press `D` in the Audio tab to rescan the device list, select the device again, and press `R` to bring the stream back.

C.8 My step is playing an old version

The step's script has an error. Cagire does not silence a failing step. It keeps playing the last version that worked and marks the step with a small symbol. Put the cursor on the step and the footer prints the message; fix the line, and the next pass plays the new version. See *Error handling* (2.10).

C.9 A sample name plays nothing

Cagire names samples after the folders they live in. A silent `"hats" snd` usually means the library does not know that name. Open the **Sample** tab of the Engine view. The header counts the folders you have registered and the files found inside them. Add the missing folder with `A`, or check the spelling against the *Samples* (2.7) view, which is also where folders get renamed. A song someone sent you needs its samples registered under the same names on your machine too. See *Sample folders* (6.4) and *Exporting and sharing* (4.6).

C.10 A patch plays nothing or will not change

A named patch only exists once its definition has compiled, so `"bass" snd` plays silence if the definition never ran. And when an edit fails to compile, the step wears the same error symbol as any other error. The footer will try to explain why and the last version that compiled keeps sounding. The message names exactly what the patch language refused, a missing word, a role mixup, a cap you crossed. See *Named patches* (2.2) and *Limits* (4.10).

C.11 Link synchronization errors

Ableton Link finds its peers on the local network. Make sure both machines sit on the same network, and check that your firewall lets Cagire and the other program talk on it. Once a peer appears, the badge in the Clock tab reads *CONNECTED* and the Metrics column counts the peers. See *Synchronization* (6.3).

C.12 External gear receives nothing

Notes leave through one of four output slots. `dev` picks the slot (0 when unsaid) and `chan` the channel, and a slot that reads *(not connected)* in the MIDI tab swallows everything sent to it, silently. Point the Output row at your interface with ←/→ and make sure `dev` and `chan` match what the synthesizer listens to. An interface that vanished mid-session reads *(disconnected)* and wants the same treatment. See *MIDI (I/O)* (6.5).

C.13 A note keeps ringing on external gear

Stopping playback sends *All Notes Off* on every channel of all four MIDI outputs. `p` (`Panic`) in the Engine view does the same on demand. If a synthesizer still drones, it holds the note on its side. See *MIDI (I/O)* (6.5).

C.14 Live recording sounds wrong

If the input device runs at a different sample rate than the output, the Metrics column flags a *rate mismatch* in red. Cagire keeps working, but match the two rates in your driver or interface for a clean capture. See *Audio input* (6.1).

C.15 My interface has more inputs than Cagire opened

Cagire opens two input channels unless you tell it otherwise, because a driver asked for nothing in particular offers a stereo pair whatever the hardware carries. Raise **In Channels** in the Audio tab to the number your interface really has and press `R`. The Metrics column then reads the granted width back to you, and says *asked 8, got 4* in red if the device refused. A refusal usually means the wrong device is selected, since the entry named for the system default is stereo by definition. Pick the interface itself in the Input column rather than the default. See *Audio input* (6.1).

C.16 Nothing arrives over OSC

Destinations live on the OSC tab, one slot per `name@host:port`, and a slot still reading *(not connected)* has nowhere to send. For incoming messages, check the Listen row (`Space`); the Last line below it shows the most recent message received, the quickest proof that anything reaches Cagire at all. If the port cannot be opened, the status bar says *OSC input failed*. See *OSC (I/O)* (6.6).

C.17 The screen looks broken

Below fifty columns of screen space, the sequencer grid gives up. It will tell you so. Give the window more room. The Engine view drops its Metrics column when the terminal narrows, which is a layout choice, not a bug. Garbled borders mean your terminal font is missing the box-drawing glyphs Cagire draws with; pick a font

that has them, or use the desktop build, which bundles one. See *Desktop & Terminal* (1.5).

Appendix D

About the author



Figure 110: Raphaël Forment

My name is Raphaël Forment, though online you will more often find me as *BuboBubo*. I am a musician and a software developer. I live and work in France. I make music by writing code. I think of my

computer not as a tool for producing recordings but as a musical instrument, something to play live and in the moment. Most of what I do is live coding: writing and changing programs on stage while the music plays, so that every performance is an improvisation and difficult to repeat. I have played at algoraves, festivals, and art venues in France and abroad, from Lyon and Paris to Bucharest and Kyrgyzstan.

I build my own instruments. Cagire is one of them. Before it there was Sardine, a live coding environment in Python, along with Topos, Sova, and a handful of smaller tools. I am a serial builder of live coding environments. I believe that live coding is a field where experimentation is of prime importance, and I am not satisfied with the monoculture built around TidalCycles and Strudel even though I find these tools fantastic!

D.1 Other projects

All my projects are free and open source, because I care about open culture and about instruments made by artists, outside the commercial circuit. I helped found livecoding.fr to grow this practice in France. I am part of the TOPLAP collective, and I play with the Cookie Collective. My research follows the same thread. In 2025 I defended a doctoral thesis on the aesthetics and techniques of live coding at the ECLLA laboratory of Jean Monnet University in Saint-Étienne. I write and teach about creative programming in music: digital lutherie, algorithmic notation, sound synthesis, and the relationship between musicians and their source code. If you want to see more, my work lives at raphaelforment.fr.

Appendix E

Contributing

Cagire is free and open source software, released under the AGPL-3.0 license. I build it for myself and for my friends and I am glad when other people find it useful and want to help. There are many ways to do that, and most of them have nothing to do with writing code. If you find a bug, report it. If you have an idea for a word, a sound or a feature, suggest it. If a sentence in this documentation is wrong, unclear, or missing, say so or fix it. If you run Cagire on a machine or an operating system I do not have, tell me what worked and what did not. And if you make music with it, I would love to hear it. All of this is contributing. The source code lives at git.raphaelforment.fr/BuboBubo/cagire. Open an issue there to report a problem or propose a change, and describe it well enough that I can reproduce it or understand what you want.

E.1 Working on the code

Cagire is written in Rust and builds with the stable toolchain. You also need a few system libraries: `cmake`, an audio backend, and, on Linux, the usual desktop graphics packages. The build instructions in the repository list the exact packages for each platform. Once they are in place, build with `cargo build` and run with `cargo run`. The workflow is the ordinary one. Fork the repository, make a branch for your change, do the work, and open a pull request. Before you send it, run the same checks I do and fix anything they report: `cargo fmt --all -- --check`, `cargo clippy --workspace --all-targets -- -D warnings`, and `cargo test --workspace`.

E.2 Conduct and license

Be kind. This project follows the Contributor Covenant, and by taking part you agree to uphold it. By contributing, you also agree that your work will be released under the AGPL-3.0 license, the same one that covers the rest of Cagire.

Appendix F

Acknowledgments

Cagire is a work of love. I am happy to share it with you freely and openly. I would not have been able to build this software without help. For this reason, I want to take some time to thank a few selected folks that have been essential to the development of Cagire:

- Felix Roos (eddyflux): for the initial design of the Dough audio engine.
- Julien Maillard & Thomas Lanza: for the intensive beta-testing, for contributing demos and for helping to fix one thousand bugs.
- Johann Philippe & Juliette Monzat: for being the first musicians to take the Cagire language on stage, in front of a real audience, back when it had barely earned that kind of trust. Bravo to them.
- Yann Orlarey & Stéphane Letz: for Faust, behind several of Cagire's effects.
- S. Christian Collins: for GeneralUser GS, the General MIDI soundfont Cagire ships with, given freely to everyone for twenty-five years.
- Athenor CNCM: Cagire language has been imagined and developed in a train somewhere between Lyon and Saint-Nazaire, most likely somewhere around Bourges.

Generally speaking, I would like to thank the core team behind the Cookie Collective and people from the TOPLAP collective/community. Live coding is evolving, maybe not in the right direction, but it is still a fever dream that I want to keep alive! I would also love to give a round of applause to the Grrrnd Zero and to the Fuz hackerspace in Montreuil!

Glossary

Bank: A group of patterns inside a project. A bank can carry its own prelude and travels as a self-contained unit when you save or share it. *See also Pattern, Project, Prelude.*

Bareword: A word Cagire does not recognise, which becomes a string instead of an error. A sound command reads a bareword as a source name. *See also Source, Word.*

Chord: A named chord quality stored in the sound register. At emit time Cagire stacks the quality's intervals above the `note` root to voice it. *See also Scale, Sound register.*

Combinator: A word that takes a quotation off the stack and runs it, deciding whether, when, or how often. The other half of control flow. *See also Quotation.*

Cut group: A numbered choke slot where only one voice may ring at a time. `cut` puts a voice in a group, and a new voice there silences the last. *See also Polyphony, Voice.*

Dictionary: The whole collection of words Cagire can run, in two layers: the built-ins and the words you define. A name is looked up here, built-ins first. *See also Word.*

Doux: The synthesis engine. It takes the sound register, builds a voice from it, and plays it. *See also Voice.*

Emit: Feeding the sound register to the audio engine with `.`, which produces one audio event, a voice. *See also Voice, Sound register.*

Envelope: The contour that opens and closes a voice's level over time. When it reaches zero the voice frees itself. *See also Voice, Gate.*

Gate: The parameter that sets how long a note is held. `0 gate` holds a voice open forever, an infinite voice you manage by hand. *See also Envelope, Held voice.*

Held voice: A voice tagged with `voice` and given an open gate so it sustains instead of fading. You reach back to it by its tag from later steps. *See also Gate, Voice.*

Insert effect: An effect that lives inside one voice's own chain and processes only that voice, in contrast with a shared orbit effect. Filters, distortion, and the movement effects are inserts. *See also Orbit.*

Live model: A step's script is fixed but the moment moves: Cagire reruns it each time the playhead arrives, and context words let it answer differently each run. *See also Step, Pattern.*

Local: A name that belongs to one word and is gone when it returns. Two forms: a `{ a b }` head block right after a definition name names the word's arguments, bound in call order, and `:>` names an intermediate partway through a body. The same forms work inside a patch. *See also Variable, Scope.*

Modulation: Moving a parameter over time instead of holding it still. Modulation words build a compact string the engine reads, for LFOs, envelopes, slides, and randomness. *See also Envelope.*

Orbit: A shared send bus that sits outside the voices. Many voices feed one orbit's reverb, delay, comb, feedback, or compressor, sharing a single space. There are eight, `0` to `7`. *See also Insert effect, Voice.*

Pattern: A row of steps run in order and looped. Each step holds a short script the sequencer runs when the playhead reaches it. *See also Step, Bank.*

Polyphony: Several voices from one emit: a parameter word takes every loose value on the stack and makes one voice per value. *See also Voice, Cut group.*

Prelude: A script that runs once before anything plays, where you put the definitions and settings everything should see. The global prelude is project-wide; each bank can carry its own. *See also Scope, Bank.*

Project: The whole of your work in Cagire, nested in four layers: a project holds banks, a bank holds patterns, a pattern holds steps. *See also Bank.*

Quotation: A piece of code kept as a value, written between parentheses and not run when read. It waits on the stack until a combinator runs it. *See also Combinator, Stack.*

Sample: A recorded sound used as a source. You register a folder of samples and play one as you would any other source. *See also Source.*

Scale: A set of pitches built on a tuning. A scale word pushes a scale value; `deg` turns a root, a scale, and a degree into a frequency. *See also Tuning, Chord.*

Scope: The part of a program where a name means something. Cagire has one scope, shared by every pattern, every bank, and the prelude, lasting the whole session. *See also Variable, Prelude.*

snd: The short form of `sound`: it takes a source name and opens the sound register. Pair it with `.` to play what you have described. *See also Sound register, Emit.*

Sound register: A record that holds the description of one sound: a source and its named parameters (`note`, `gain`, `lpf`, and so on). It is separate from the stack; `snd` opens it and `.` reads it and emits. *See also Source, Emit.*

Soundfont: A bank of sampled instruments in an `.sf2` file, the General MIDI sound set. `gm` is its source and `n` chooses the instrument. *See also Source, Sample.*

Source: A sound generator named into the sound register, such as an oscillator, a drum model, or a sample. Cagire reads an unrecognised bareword like `sine` or `saw` as a source name. *See also Sound register, Bareword.*

Stack: The single pile of values every word reads from and writes to. You add and take from the top, writing the data first and the word that uses it after. *See also Quotation.*

Step: One cell of a pattern, holding a short script Cagire runs the moment the playhead reaches it. A step is on or off, separate from what it holds. *See also Pattern.*

Tuning: A division of the octave into pitches. `edo` makes an equal temperament, `tuning` builds one from cents, and a scale picks degrees out of it. *See also Scale.*

Variable: A name with a value you set aside and fetch later: `!name` and `,name` store it, `@name` reads it. One table, shared by the whole session. *See also Scope.*

Voice: A single self-contained instrument the engine builds from the sound register, bundling a source with its own envelope, filters, and effects. It sounds, then frees its slot. *See also Doux, Envelope.*

Word: A name Cagire knows how to run. Built-in words ship with the engine; you define your own with `:` and `;`. *See also Dictionary.*

Bibliography

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press.

Brodie, Leo. 1984. *Thinking Forth*. Prentice-Hall.

Brodie, Leo. 1987. *Starting Forth*. 2nd ed. Prentice-Hall.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3): 321–30.

Index of Concepts

B

Bank **499**

Bareword 112, 114, 128, 145, 154, 199, 289, 293, 294, 353, 358, 392, 475, **499**

C

Chord **499**

Combinator 104, 105, 111, 114, 122, 123, 124, 129, 132, 135, 142, 149, 159, 161, 162, 164, 173, 178, 444, 476, **499**

Cut group 278, 280, 282, **499**

D

Dictionary **499**

Doux 7, 45, 244, 245, 277, 280, 282, 316, 390, 446, 468, 469, **499**

E

Emit **499**

Envelope 68, 117, 118, 120, 132, 210, 211, 213, 216, 217, 218, 219, 220, 226, 227, 244, 245, 246, 254, 256, 257, 258, 259, 274, 275, 277, 278, 279, 280, 284, 285, 315, 316, 317, 318, 319, 320, 322, 323, 324, 325, 328, 346, 373, 375, 383, 391, 393, 396, 399, 401, 402, 406, 407, 413, 415, 418, 419, 420, 431, 436, 437, 438, 439, 451, 453, 457, 460, 461, 464, 468, **499**

G

Gate 9, 68, 73, 118, 177, 179, 180, 181, 182, 183, 184, 194, 211, 214, 226, 227, 274, 279, 282, 283, 304, 306, 307, 309, 311, 313, 316, 317, 391, 396, 400, 418, 420, 433, 435, 438, 460, 461, 468, **499**

H

Held voice 246, 278, 279, 282, 283, 288, 313, 392, 393, 402, 406, 419, 420, **500**

I

Insert effect 245, 246, 255, 284, 319, 320, 321, 323, 325, 327, 329, 333, 335, 337, 343, 345, 346, 348, 349, 351, 352, 385, 399, **500**

L

Live model 105, 124, 125, 154, 156, 170, 172, 174, 175, 180, 238, **500**

Local **500**

M

Modulation **500**

O

Orbit 11, 35, 53, 87, 94, 218, 230, 236, 237, 245, 246, 256, 258, 272, 276, 277, 282, 314, 319, 320, 325, 328, 351, 352, 353, 355, 356, 357, 359, 361, 362, 363, 364, 365, 366, 367, 369, 372, 376, 377, 378, 379, 383, 384, 385, 391, 393, 395, 396, 399, 400, 401, 402, 403, 433, 436, 445, 460, **500**

P

Pattern **500**

Polyphony 114, 117, 119, 132, 146, 148, 151, 154, 276, 280, 385, 469, 474, **500**

Prelude 24, 44, 55, 58, 72, 78, 79, 80, 81, 99, 101, 105, 122, 127, 129, 169, 230, 238, 239, 240, 388, 436, 440, 471, **500**

Project 5, 8, 9, 10, 12, 14, 17, 23, 30, 32, 33, 37, 41, 44, 62, 71, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82, 95, 129, 238, 240, 369, 383, 387, 390, 494, 496, **500**

Q

Quotation **501**

S

Sample **501**

Scale **501**

Scope 35, 36, 39, 45, 52, 53, 118, 121, 123, 127, 150, 152, 154, 164, 167, 238, 377, **501**

Snd **501**

Sound register **501**

Soundfont 90, 91, 118, 246, 274, 275, 276, 306, 307, 316, 478, 497, **501**

Source **501**

Stack **501**

Step **501**

T

Tuning **501**

V

Variable 19, 105, 109, 118, 119, 123, 127, 128, 129, 130, 132, 149, 150, 152, 153, 154, 164, 167, 238, 256, 425, 448, 454, 474, 476, **502**

Voice **502**

W

Word **502**

Index of Words

A

above 212, 214
abs 140, 151, 466
ad 317, 322
address 234, 235
adsr 317, 396, 418, 461
after 27, 108, 120, 133, 181, 182
all 118, 119, 144, 145, 146, 154, 357, 495
always 179, 180
am 291, 292, 319
amdepth 291, 292
amshape 292
anchor 202, 204
and 143, 471
apply 123, 124, 161, 162, 167
at 69, 70, 123, 152, 165, 167, 184, 185, 191, 192, 193, 218, 356
att 316
attack 121, 166, 275, 316, 318, 393, 396, 457, 461

B

bank 273
barlow 189, 190
barlowg 189, 190
barlowv 189, 190
beat 125, 155
beatsecs 212, 216, 221, 222, 337, 357
begin 254, 304, 305, 306, 307, 309, 310, 312, 475
bend 225, 231, 232, 233
bjork 73, 158, 180, 181, 183, 189
bjorkrot 181, 183
body 240, 259, 317, 318
bounce 151, 170, 171, 174
bpf 321, 322, 323, 409, 454
bpq 321, 322
bright 166, 251, 454
bs 212, 216, 221
bus 99, 101, 228, 229, 230, 234, 236, 391, 402, 446, 469

C

cattack 364
ccnum 231, 233
ccout 225, 231, 232, 233
ccval 229, 231, 234
ceil 115, 140, 465
chan 228, 233, 234, 443, 490
chance 158, 179, 180
choose 6, 112, 149, 151, 152, 175, 176, 177
chord 119, 148, 197, 200, 202, 255
chorus 319, 340, 341
chorusdelay 340, 341
chorusdepth 340, 341
chorustype 340, 341
clamp 139, 140

clear 118, 144, 145, 228, 230, 335
cn 202, 205
coarse 331, 332
coin 179, 396, 453, 460, 461
comb 351, 359, 360, 396, 429, 430, 458, 459
combdamp 359, 360, 361
combfeedback 359, 360, 361
combfreq 359, 360
comp 362, 363, 364, 365, 378, 396, 414, 457
compattack 364
comporbit 363, 364
compratio 363, 364
comprelease 364
compthresh 362, 364
corbit 363, 364
cos 140, 466
cratio 363, 364
crelease 364
crush 6, 331, 332, 396, 412, 457
cthresh 362, 364
ctype 340, 341
current 209, 211, 213, 217, 218, 222, 227
curve 208, 212, 214, 222, 462, 466, 467
cut 281, 282
cycle 68, 149, 151, 152, 158, 170, 171, 173, 174, 184

D

dasym 335, 336
dec 316
decay 7, 11, 275, 316, 317, 318, 396, 454, 456, 461
defdsp 400, 401, 402, 436
deffx 402, 436
defsynth 134, 238, 383, 387, 388, 389, 390, 436, 440
deg 199, 200, 201, 202, 206, 207
delay 117, 351, 356, 358, 393, 395, 426, 427, 430, 458
delayfeedback 357, 358
delaytime 69, 353, 356, 357, 358
delaytype 358
dens 309, 310
depth 131, 138
detune 287, 288
dev 229, 233, 234, 235, 490
dip 123, 124, 161, 162, 167, 476
distort 167, 168, 284, 320, 333, 336, 396, 412, 456
distortasym 335, 336
distortmode 334, 336
distortvol 333, 336
div 191, 192
dmode 334, 336
drop 110, 137, 443, 463
drumkit 259, 260, 261
drunk 209, 213, 216, 227, 451
dsp 230, 352, 384, 389, 399, 400, 401, 402, 433, 435, 436, 447, 469
dub 87, 236, 314
dup 108, 110, 119, 121, 137, 149, 161, 167, 197, 397, 422, 427, 442, 443, 477
dupn 137

E

each 124, 151, 164, 165
ead 210, 213, 318
eadr 210, 318
eadsr 210, 318
edo 114, 202, 205, 206
end 186, 187, 208, 219, 254, 304, 305, 306, 307, 309, 310, 312, 461, 462
enddub 237
endorec 237
endrec 87, 237, 314
env 210, 216, 318, 323
envdelay 315, 318
envdly 315
eqhi 326, 327
eqhifreq 326, 327
eqlo 326, 327
eqlofreq 326, 327
eqmid 326, 327
eqmidfreq 326, 327
eqmidq 327
euclid 188, 189, 190, 191
euclidrot 183, 188, 189
every 73, 126, 158, 181, 182
except 181, 182
excite 251
exp 141, 466
expmap 141, 144, 208, 209, 212, 213, 223, 228
exprand 177, 178, 397, 451, 452

F

fb 361, 362, 396, 429, 442, 445, 448, 449, 458, 459
fbc 361, 362
fbcross 361, 362
fbd 361, 362
fbdamp 361, 362
fbt 361, 362
fbtime 357, 361, 362
feedback 351, 361, 362
fill 41, 125, 155, 156
filter 124, 151, 164, 165
first 181, 182
flanger 319, 342, 343
flangerdepth 342, 343
flangerfeedback 342, 343
flangermode 342, 343
flmode 342, 343
floor 115, 140, 397, 465
fm 296, 297, 298
fm2 255, 296, 297, 298
fm2h 296, 298
fmfb 166, 297, 298
fmh 166, 296, 298
fmi 296, 298
fmloop 297, 298
fmode 338, 339
fmpivot 296, 297, 298
fmshape 296, 297, 298
fold 337, 338, 339, 396, 465, 466
foldmode 338, 339
forget 120, 130, 169
freq 153, 154, 225, 234, 235, 254, 261, 262, 263, 265, 269, 271, 272, 288, 290, 302, 303, 308, 311, 391, 448, 449, 450, 454, 455, 456, 460, 476
fsh 349, 350
fshift 319, 349, 350

ftom 205, 206, 468
fx 319, 384, 389, 399, 401, 402, 433, 435, 447, 469

G

gain 6, 115, 116, 117, 144, 148, 168, 245, 284, 285, 319, 320, 333, 337, 378, 393, 455
gate 69, 117, 118, 145, 230, 231, 278, 279, 280, 282, 283, 315, 316, 317, 383, 391, 393, 396, 402, 418, 420, 436, 437, 446, 452, 461, 469
gen 124, 148, 151, 164, 165, 185, 186, 187
glide 283, 287, 288
grain 255, 303, 307, 308, 310, 311, 312
grit 259, 262, 263, 264, 265, 266, 267, 268, 269, 270
gt 115, 142, 159, 165

H

haas 286, 319, 366, 371
harmonics 255, 256, 259
hld 316
hold 316, 318
hpf 321, 322, 323, 408, 453
hpq 321, 322

I

ifelse 124, 143, 163, 476
inchan 92, 274
index 126, 151, 155, 163, 172, 188, 189, 191, 205, 465
iter 105, 125, 126, 127, 155, 156, 158, 159, 160, 170, 172, 174, 175, 176, 180, 238

J

jit 209, 216, 259, 321, 329, 360, 451

K

keep 124, 161, 162, 167, 476
kf 324

L

later 212, 215, 216, 217, 221
lbpf 323
lbpq 323
levels 208, 212, 214, 228, 379
lfo 207, 213, 216, 217, 254, 259, 271, 284, 286, 290, 303, 311, 357, 367
lhpf 323
lhpq 323
linmap 141, 223
llpf 323
llpq 323
log 141, 466
logrand 177, 178, 397, 451, 452
loop 184, 216, 304, 306, 307, 310, 475
lpf 11, 28, 105, 107, 115, 117, 144, 207, 209, 210, 212, 217, 220, 228, 321, 322, 323, 383, 384, 386, 389, 392, 395, 408, 410, 411, 419, 429, 443, 453, 455
lpg 211, 322

lpq 321, 322, 323
lt 115, 142

M

map 123, 124, 148, 149, 151, 164, 165, 167
max 140, 141, 177, 207, 209, 210, 212, 219, 223, 396, 466
mclock 233
mcont 233
mdown 156
min 140, 141, 177, 203, 207, 209, 210, 212, 219, 223, 396, 397, 466
mirror 100, 250, 299, 300
mod 140, 144, 159, 160, 163, 465
modal 255, 328, 329, 454
modalbright 329, 330
modaldecay 328, 329
modalfreq 328, 329
modalstruct 328, 329, 330
mode 199, 202
morph 255, 256, 259
mstart 233
mstop 233
mtof 205, 206, 397, 449, 462, 467, 468
mx 156, 157
my 156

N

nand 143
neg 140, 466
never 179, 180
nip 110, 137, 443
noall 119, 144, 146
nor 143
not 143, 220
note 11, 104, 106, 107, 115, 116, 117, 144, 145, 146, 148, 149, 151, 152, 164, 165, 196, 197, 198, 200, 201, 202, 204, 205, 231, 235, 239, 254, 257, 271, 272, 275, 280, 288, 290, 302, 303, 308, 311, 388, 391
nsteps 125, 155

O

oct 198, 200
odub 236
often 180
on 180, 181, 183
once 73, 181, 182, 218
or 143, 216
orbit 236, 237, 351, 362, 366, 369, 379
orec 87, 236, 314, 378
osc 101, 142, 208, 209, 211, 212, 213, 216, 217, 221, 222, 227, 228, 229, 236, 446
oscval 234, 235, 236
over 110, 137, 397, 443
overdub 236, 237

P

pan 117, 235, 285, 286, 310, 319, 366, 367
param 119, 153, 154, 386, 392, 397, 402, 436, 442, 446, 447, 469, 470, 471
pat 114, 185, 193, 195, 196

patchlevel 400, 401, 402, 435, 436
pattern 125, 155, 193, 195
pbank 125, 155
pbjork 181, 183
pbjorkrot 181, 183
pbounce 151, 170
pcycle 151, 170, 173, 174
phase 125, 127, 155, 238
phaser 319, 344, 346
phasercenter 345, 346
phaserdepth 344, 346
phasersweep 345, 346
pick 304, 305, 306, 307, 312, 313
pinv 195
plevel 402
postgain 245, 284, 285, 319, 320, 378
pow 140, 466
prcycle 151, 170
pressure 225, 231, 232, 233
prev 195
print 108, 109, 128, 132, 133, 135, 138, 149, 150, 155, 156, 159, 166, 206, 234, 475
prob 179, 396, 460
prod 139, 151, 152
program 231, 233
prot 191, 195
psh 349, 350
pshift 319, 349, 350
pshiftwin 350
punch 259, 262, 263, 264, 265, 266, 267, 269, 270
purn 151, 175, 176
pw 248, 249
pwalk 151, 170, 171
pwin 350

R

rand 165, 166, 168, 177, 397, 431, 450, 452, 463
range 141, 157, 209, 211, 212, 213, 221, 223, 224, 228, 338, 396, 397, 410, 434, 451, 467
rarely 180
rcycle 151, 170
rec 87, 236, 237, 313, 314
recurse 120, 169
rel 316
release 118, 275, 278, 283, 315, 316, 318, 393, 396, 457, 461
repeat 212, 216, 220, 475
reset 280
rest 114, 115, 119, 144, 145, 148, 158, 163
retrig 212, 217, 221
rev 139, 151
ring 251
rise 208, 209, 211, 213, 214, 215, 216, 217, 227, 272
rm 292, 319
rmdepth 292
rmshape 292
rot 110, 137, 397, 443
round 115, 140, 465
rsort 139, 151
runs 105, 125, 126, 155, 156, 158, 170, 171, 174, 175, 176, 180, 184, 305

S

sample 212, 221, 222, 227, 229, 337
sbpf 322, 323
sbpq 322
scale 114, 152, 199, 200, 201, 202, 205, 207
scan 252, 253, 254, 307, 310, 311, 312, 313
seed 139, 171, 176, 177, 178, 224, 238
select 124, 143, 152, 163, 164, 397, 465
shape 211, 220, 259, 260, 262, 263, 264, 265, 266, 267, 268, 269, 270
shpf 322, 323
shpq 322
shuffle 139, 151
sign 140, 466
sin 140, 466
size 250, 299, 300
slew 209, 213, 216, 397, 453
slice 90, 304, 305, 306, 307, 312, 313
slide 11, 208, 213, 216, 284, 290, 294, 311, 327, 329, 332, 343, 345, 348, 349, 360, 363
slot 125, 155
slpf 322, 323
slpq 322, 323
smear 319, 346, 347, 348
smearfb 347, 348
smearfreq 347, 348
smooth 212, 214, 218, 227, 228
snap 199, 201
snd 6, 11, 60, 61, 84, 104, 106, 110, 115, 116, 117, 119, 121, 134, 144, 145, 146, 151, 154, 167, 168, 193, 237, 248, 252, 253, 254, 273, 275, 278, 314, 315, 316, 388, 390, 475, 489, 490
sometimes 6, 180
sort 139, 151
sound 106, 116, 118, 144
span 367, 369
speakers 151, 368, 369
speed 157, 254, 273, 302, 303, 304, 306, 308, 310, 311, 312
spk 368, 369
spray 309, 310
spread 286, 287, 288, 310
sqrt 140, 466
ss 212, 216, 221
step 56, 105, 125, 155, 172, 180, 186, 187, 188, 189, 238
stepdur 125, 155
stepsecs 208, 209, 212, 216, 221, 222
stretch 254, 273, 302, 303, 306, 307, 308, 309, 310, 311, 312
stut 69, 304, 306, 307
sub 293, 294
suboct 293, 294
subwave 293, 294
sum 108, 110, 138, 139, 150, 151, 152
superpan 367, 369, 371
superwidth 367, 369
sus 203, 316

sustain 256, 275, 316, 318, 328, 461
swap 110, 137, 167, 397, 410, 422, 435, 443
swidth 367, 369
swing 77, 191, 192
sync 294, 295
syncmode 294, 295
syncph 295
syncphase 295
synth 134, 234, 235, 383, 385, 386, 387, 388, 390

T

tan 141, 466
tempo 125, 155, 157
tense 258
tilt 319, 327
timbre 255, 256, 259
time 395, 396, 429, 453, 456, 458, 459, 461
times 124, 129, 154, 164, 165, 397, 426, 444, 459, 470, 475
tonguex 257
tonguey 257
trunc 140, 465
tuck 110, 137, 443
tuning 114, 152, 202, 205, 206, 207
tx 258
ty 258

U

uni 415, 416, 466, 467
unison 197
urn 68, 151, 175, 176

V

velocity 231, 284, 285, 319
velum 258
verb 11, 105, 115, 117, 146, 351, 353, 355, 386, 393, 395, 396, 430, 459
verbchorus 355, 356
verbchorusfreq 355, 356
verbdamp 354, 355
verbdecay 353, 354, 355
verbdiff 354, 355
verbhighcut 355
verbhighgain 355
verblowcut 355
verblowgain 355
verbpredelay 354, 355
verbprehigh 355
verbprelow 355
verbsize 354, 355
verbtype 353, 355
vib 257, 289, 290
vibmod 289, 290
vibshape 289, 290
vinyl 319, 335, 336
vinylnoise 335, 336
vinyltone 335, 336
vinyltype 335, 336
vinylwow 335, 336
voice 279, 280, 282, 313
vowel 330
vowtype 331
vsize 354, 355

W

wah 319, 324, 325
wahmanual 324, 325
wahpeak 324, 325

wahsens 324, 325
walk 68, 151, 158, 170, 171, 184, 216
warp 250, 299, 300
wave 248, 249, 250, 260, 262
wchoose 151, 175, 176
width 286, 319, 366, 371, 405, 448, 449
wrap 216, 338, 339, 396, 465
wt 144, 145, 252, 253, 254
wtlen 253, 254

X

xnor 143
xor 143

The Cagire Book
Set in New Computer Modern and PT Mono
Typeset with Typst
Generated from the Cagire documentation sources