azCore — User Manual

Version 1.1.0 · April 2026

Table of Contents

Part 1 — Introduction

What is azCore?

azCore is a professional live music performance control system. It is a single, self-contained application written in Rust that lets you describe your entire stage setup — MIDI routing, audio playback, cue communication, on-screen scores, and UI control surfaces — in a purpose-built language called **azSyntax**.

At runtime, azCore compiles your `.az` files into a deterministic execution engine that responds in real time to MIDI events, keyboard shortcuts, and touch input. You describe *what should happen* and *when*; azCore makes it happen reliably, every night, every show.

A typical azCore installation manages: - Multiple MIDI keyboards and controllers routing to multiple synthesizers and sound modules - Audio clips (stems, sound effects, click tracks) triggered on demand - HTML-based sheet music and chord charts rendered on a touch monitor - Color-coded cue labels displayed to band members and production crew - Programmable buttons (pads) that can be pressed via touch, MIDI, or keyboard

Everything is driven by a single plain-text project: one or more `.az` files, one INI configuration file, and your media. No proprietary binary formats, no cloud dependencies, no required internet connection.

Who is it for?

Professional live musicians who perform with backing tracks, MIDI rigs, and multiple instruments and need precise, repeatable control over their setup every night.

Musical directors and keyboard players who must manage complex routing — splitting keyboards, layering patches, switching programs between songs — and want that complexity expressed as readable, editable code rather than buried in device menus.

Live audio engineers and production technicians who need a reliable way to send cues to lighting, video, and monitor engineers from a single centralized system.

Bands and touring artists who have 40, 60, or more songs in their catalog and need to load the correct patches, routes, and signals for each song in under a second, without any manual intervention.

Studio technicians who want to prototype complex MIDI and audio setups in a language they can version-control, share with collaborators, and reproduce exactly on any machine.

Where it excels

Setlist-driven performance. Define your setlist and all songs up front. azCore pre-compiles everything at boot. Song switching is instantaneous — no pause, no loading spinner.

Complex MIDI rigs. When you have three keyboards feeding six synth modules, with different splits, transpositions, and velocity curves per song, and it all needs to change flawlessly mid-note, azCore's Smooth Patch technology handles it without stuck notes or audible artifacts.

Multi-personnel communication. The cue system lets you flash color-coded messages to your lighting director, monitor engineer, video operator, and band simultaneously from a single `cue` command in your `.az` file.

Hybrid setups. azCore handles MIDI, audio clips, score display, and device hotplug within a single coherent system. You do not need separate apps for each concern.

Reliability under pressure. Every component is designed for live use: the compiler catches errors at boot, not on stage. The Smooth Patch eliminates stuck notes. The panic button resets everything. The REPL lets you diagnose any problem while the show goes on.

Touchscreen-first UI. The graphical interface is designed to be operated on a touch monitor at arm's reach while performing. Buttons are large, colors are meaningful, and the layout locks during the show so accidental touches do not move panels.

Remote monitoring. The UI is served over WebSocket. You can run the control surface on an iPad, a dedicated rack monitor, or a second laptop over Wi-Fi — with zero configuration change.

System requirements

	Minimum	Recommended
OS	macOS 12+, Windows 10	macOS 14+, Windows 11
CPU	Any modern x86_64 or ARM64	Apple Silicon M-series
RAM	512 MB	4 GB+ (for large audio preload sets)
Storage	100 MB (binary + project)	SSD recommended for fast audio decode
MIDI	Any CoreMIDI-compatible device (macOS) or Windows MIDI device	
Audio	Any Core Audio device (macOS) or WASAPI-compatible device (Windows)	
Display	1024×768	1366×768 or higher, touch-capable
Network	Not required	Wi-Fi for remote UI access

Part 2 — Components and Concepts

The azSyntax language

azSyntax is the domain-specific language you use to describe your entire performance setup. It is a declarative-imperative hybrid: some constructs declare resources (a device, a route, a song), while others execute actions (send MIDI, play audio, navigate steps). The two kinds are clearly distinguished in the language.

Key design decisions: - **Plain text.** `.az` files are UTF-8 text. Use any editor. Put them in git. - **One file or many.** You can write everything in one file or split across hundreds. The compiler assembles them. - **Readable by non-programmers.** The syntax is intentionally close to natural language: `route.add r_piano`, `audio.play intro_pad`, `cue band_cues "Verse — everybody in"`. - **No runtime ambiguity.** All symbols are resolved at compile time. If a device name is misspelled or a song references a missing block, the error is caught at boot, not on stage. - **Profiles.** You can write conditional sections using `profile live { ... }` or `profile rehearsal { ... }` to maintain a single project file that behaves differently in production vs. rehearsal.

A minimal `.az` file looks like this:

```
; Declare a MIDI output device
midi.device piano {
    direction = output
    channel = 1
}

; Declare a song
song my_song {
    caption = "My First Song"

    step default {
        midi.send piano { pc,3  cc,7,100 }
        message "Ready" { duration = 2s }
    }
}
```

Everything else is an elaboration of these fundamentals.

The compiler pipeline

When azCore starts, it compiles your project through ten sequential phases before a single MIDI message is sent:

Phase	What happens
0–1	Source files are read and tokenized
2	Tokens are parsed into an abstract syntax tree (AST)
3	Profiles are evaluated; blocks are expanded inline
4	Semantic validation: type checking, scope resolution, unknown references
5–9	Song and setlist catalogs are built; all songs pre-compiled
10	The executor starts; global code runs; the UI and REPL start

If any ERROR-level diagnostic is produced in phases 1–9, compilation stops and azCore does not start. This means every error in your `.az` files is caught before the show, never during. Warnings are printed but do not stop compilation.

This guarantees that if azCore starts successfully, your entire project is structurally valid.

The console (REPL TUI)

Start azCore to open the interactive console:

```
cargo run
```

The console is a full-screen terminal UI built with ratatui. It provides: - A scrollable log window showing all system events, errors, and command output - A command prompt at the bottom with command history (Up/Down arrows) - Mouse wheel scrolling in the log - All system output (MIDI events, route changes, audio status) streams here in real time

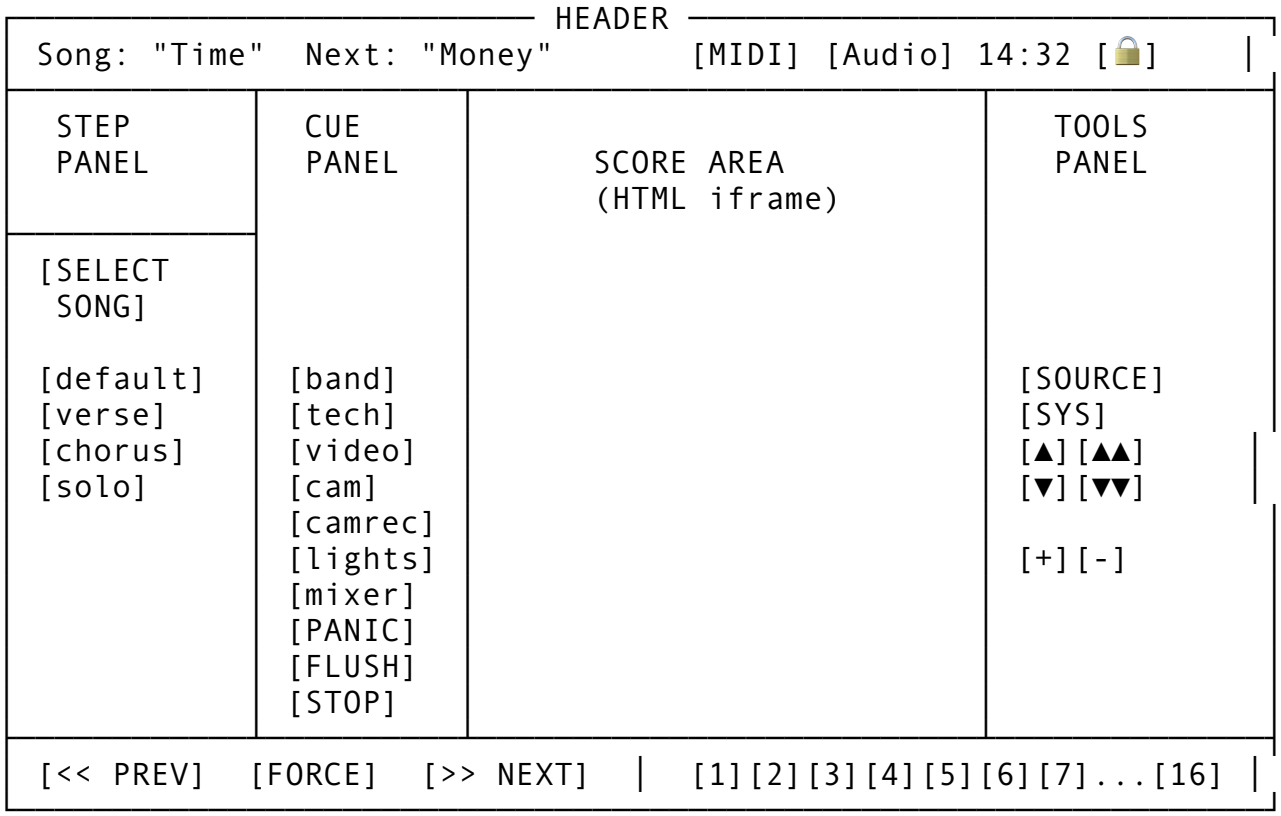
The console is the primary diagnostic and testing tool. During development of a new song, you will use it constantly. During a show, it runs invisibly in the background while the graphical UI is front and center.

Key console capabilities: - Load songs by name or open the song selector - Navigate steps, inspect routes and pads, check variables - Enable MIDI monitor to see all incoming and outgoing MIDI in real time - Run `--test` mode to auto-step through a song and verify it compiles correctly - Execute any azSyntax command interactively - Bind MIDI devices to physical ports

The graphical UI

The graphical UI is a web-based interface embedded in a native window (WebKit on macOS, WebView2 on Windows). It communicates with the azCore backend over a local WebSocket connection.

The UI is divided into panels:



HEADER — Shows current song name, upcoming song (from setlist), device status LEDs (MIDI IN/OUT activity, device bindings), current time, and resize lock indicator.

STEP PANEL — Lists the steps of the current song. Tap a step to jump to it. Includes the `SELECT SONG` button that opens the song/setlist selector.

CUE PANEL — Color-coded cue labels (band, tech, video, cam, cam_rec, lighting, mixer). Tap to send a cue from the `.az` file, or they light up automatically when the running song sends them. Also contains the emergency buttons: PANIC, FLUSH, STOP ALL.

SCORE AREA — The main center area. Normally displays the HTML score associated with the current step. Can also show the song selector (overlay) or the 4×12 board pad grid.

TOOLS PANEL — Score scroll and zoom controls. SOURCE button shows the raw `.az` source of the current song. SYS button opens the system overlay with quick access to CONNECTIONS (device bindings), ROUTE STATUS (live MIDI routes), SETTINGS (project file and profile), and system controls (restart, shutdown, lock layout).

PRESET BAR — The bottom row of 16 programmable buttons (`preset(1..16)`). Color reflects button type: blue for instant, red for toggle, pink for repeat. Expandable to the full 48-button board grid.

NAV BAR — Below the step panel: PREV, FORCE, and the large NEXT button.

The UI layout is fully resizable via drag handles between panels. In live use, set `ui_resize_lock = on` in the INI to freeze the layout and prevent accidental panel resizing.

System overlay panels

The **SYS** button opens an overlay with several system control panels accessible via buttons:

- **CONNECTIONS:** A unified view of the server URL, all connected client IPs, and your MIDI/audio device bindings (IN/OUT badges). Use this to verify that all devices are correctly bound before starting a performance.
- **ROUTE STATUS:** A live-updating display of all active MIDI routes, organized by input and output devices. Shows props (split, transpose, velocity, auto_hold) and template ID for each route. Auto-refreshes whenever routes are added, removed, or modified.
- **SETTINGS:** Select the project file (`.az` file) and performance profile that azCore will use. Changes take effect on the next restart.
- **Additional controls:** LOCK LAYOUT (prevent accidental panel resizing), RESTART (reload the current song), and SHUTDOWN (exit azCore).

MIDI routing

The MIDI router is the heart of azCore's live performance capability. It connects MIDI input devices to output devices with declarative rules.

Core concepts:

Logical devices. In `.az` files, you only ever reference logical device names (`piano`, `organ`, `bass_synth`). The physical binding — which USB port or MIDI interface port corresponds to `piano` — lives in `azCore.ini [midi_bindings]`. This separation means you can move to a different venue with different hardware by changing one line in the INI, not rewriting any songs.

Routes. A `route` declaration maps one or more input devices to one or more output devices, with optional filtering and transformation. Properties include: - `split = 36:96` — only forward notes in MIDI note range 36–96 - `transpose = 12` — transpose by 12 semitones (one octave up) - `velocity = 20:127` — clamp velocity to the range 20–127 - `auto_hold = on` — note-latch mode (notes stay on until pressed again)

Routes are declared but inactive. You activate them with `route.add`, deactivate with `route.remove`.

Smooth Patch. When you call `route.remove` on an active route, azCore does not immediately cut the connection. Instead, it enters a **RELEASE state**: the route continues forwarding `note_off` messages and sustain pedal releases for a configurable grace period (default 2 seconds). Only after all held notes have resolved (or the grace period expires) does the connection drop completely. This eliminates stuck notes when switching patches mid-note.

Exclusive routing. The `exclusive()` keyword on a `route.add` call automatically removes all routes sharing the specified input devices. This is the standard way to switch keyboard splits cleanly: one command adds the new route and removes any conflicting ones.

Router bypass. `midi.send` and `midi.automation` always bypass the router and go directly to their output device. The router only affects notes coming in from input devices.

Audio playback

azCore plays audio clips (`.wav` and `.mp3`) via the system audio stack (CPAL, with native CoreAudio planned).

Logical audio devices. Just like MIDI, audio output devices are declared with logical names and bound to physical OS audio devices in the INI. This lets you move between systems without changing `.az` files.

Clip declaration. An `audio.clip` block declares a playable audio asset: file path, output device, volume, optional trim points (`start_pos`, `end_pos`), and fade parameters.

Preload. For clips that must play instantly without any decode latency: - **Boot-time preload:** Add `preload = on` to the `audio.clip` block. The clip is decoded silently at boot, before the UI starts. When played, the audio starts immediately. - **On-demand preload:** Call `audio.preload clip_name` at runtime to decode a clip in a background thread. A progress indicator appears in the UI during decode. After decode, the clip plays instantly on every subsequent `audio.play`.

UI controls. When a clip is playing, the UI shows a visual progress ring with time remaining. Dedicated PAUSE, PLAY, and FADE buttons appear. The FADE button smoothly fades the clip out over a configurable duration (default 10 seconds).

Volume. Two-level volume control: each `audio.clip` has its own `volume` property (0–100), and each `audio.device` has a `default_volume` baseline. The effective volume is a combination of both.

Score display

The score system renders HTML files in a full-screen iframe in the center of the UI.

Flexibility. Using HTML for scores means anything expressible in a browser works: chord charts, tablature, SVG notation, zoomable PDFs converted to HTML, images with overlaid text, animated cue diagrams. If a browser can render it, the score system can show it.

Navigation. HTML anchor fragments (`marker`) allow jumping directly to a section within a long score document. You can define a score that covers an entire concert and use `score.show` with different markers per song.

Persistence. Score visibility is independent of song scope. When you switch songs, the score stays visible until you explicitly call `score.hide` or a new song shows a different score. This is intentional: your chord chart should stay on screen during a song even if you navigate between steps.

Off-page. When `score.hide` is called, the score area shows a configurable off-page display (logo, blank, or custom HTML).

Device management and hotplug

MIDI hotplug (macOS). macOS CoreMIDI does not allow a running process to detect ports added after it started. azCore works around this with a subprocess architecture: a background monitor process runs continuously, enumerating ports and comparing against the known set. When a new device appears, it spawns a proxy subprocess that connects to the port and forwards MIDI data over a Unix socket to the main process. This is transparent to `.az` files and users.

MIDI hotplug (Windows). Uses the native Windows MIDI API with periodic polling for device changes.

Audio hotplug. Audio device loss is detected via callback error monitoring and a configurable heartbeat interval. When an audio device disappears and reappears (e.g., USB audio interface reconnected), azCore automatically re-establishes the connection.

Console commands. At any time: `midi rescan` refreshes the MIDI device list, `audio rescan` refreshes the audio device list. `midi bind <id> <index>` and `audio bind <id> <name>` let you reassign logical devices to physical ports without editing the INI file.

License and activation

azCore uses offline license validation. No internet connection is required during normal operation.

License key format: `AZCO-XXXX-XXXX-XXXX-XXXX`

Activation. Enter your license key in the About modal (accessible from the header). The key contains a JWT payload signed with an RS256 private key; the public key is embedded in the binary. Verification happens entirely on-device.

Trial. On first run, azCore automatically starts a 30-day trial with full functionality. The trial status is shown in the About modal and as a persistent banner in the score area. When the trial expires, an overlay appears on startup; it can be dismissed for each session, but the banner remains.

Machine binding. License keys are bound to a machine ID derived from a hash of the platform hardware identifier (IOPlatformUUID on macOS, MachineGuid on Windows). Contact support to transfer a license to a new machine.

Part 3 — Language Reference

Data types and syntax conventions

Syntax basics

- **File extension:** `.az`
- **Comments:** ; starts a comment to end of line
- **Case sensitivity:** keywords are case-insensitive; identifiers are case-sensitive
- **Statement boundary:** newline — one statement per line, enforced by the parser
- **Inline exception:** a single child statement may appear on the same line as its parent's opening { (e.g., `wait 5s { route.set r { auto_hold=off } }`)
- **Block delimiters:** { and }
- **List separator:** ,

Data types

Type	Description	Examples
Integer	Decimal or hex (`h` prefix)	`42`, `-12`, `hA4`
String	Double-quoted, backslash escapes	`"Hello\nWorld"`, `"path/to/file.wav"`
Boolean	`on`/`off` or `true`/`false`	`on`, `off`, `true`, `false`
Duration	Compound `Nh Nm Ns Nms` (order fixed)	`1m30s`, `500ms`, `2h`, `5s`
MIDI note	`$` prefix, note name + octave	`$C3`, `$A#4`, `$Bb2`
Hex color	`#rrggbb` or `hRRGGBB`	`#ff0000`, `hFF00FF`

Duration note: Components must appear in order (hours → minutes → seconds → milliseconds). `30s500ms` is valid; `500ms30s` is not.

MIDI note convention: C3 = MIDI note 60 (Yamaha convention). This can be changed in the INI (`midi_map_c3`), but the default matches most hardware.

Variable interpolation

Variables can be interpolated in strings and MIDI message blocks:

```
var vol = 100
var colors = midi{pc,17  cc,7,90}

midi.send piano {cc,7,$vol}          ; scalar interpolation
midi.send organ {$colors}            ; midi fragment interpolation
message "Now: $song_name"            ; string interpolation
```

The special variable `$song_name` is always available and contains the caption of the current song.

MIDI message blocks (`midi{}`)

The `midi{}` literal appears in `midi.send`, variables, and trigger event patterns:

```
; Multiple messages in one block
midi.send piano { pc,3  cc,7,100  cc,11,127 }

; Store in variable, use later
var init_patch = midi{ pc,3  cc,7,100 }
midi.send piano { $init_patch }

; In trigger event pattern (device inside)
fire = midi{ device=icon  note_on,$C1,any }
```

Message types:

Syntax	MIDI message	Parameters
`pc,<val>`	Program Change	0–127
`cc,<ctrl>,<val>`	Control Change	ctrl: 0–127, val: 0–127
`note_on,<note>,<vel>`	Note On	note: 0–127 or `$Cx`, vel: 0–127 or `any`
`note_off,<note>,<vel>`	Note Off	note: 0–127 or `$Cx`, vel: 0–127 or `any`
`pitch_bend,<val>`	Pitch Bend	–8192 to +8191
`after_touch,<val>`	Channel Aftertouch	0–127

Keyboard shortcut blocks (`shortcut{}`)

```
press = shortcut{ key = "CTRL+F1" }
fire  = shortcut{ key = "PAGEDOWN" }
fire  = shortcut{ scan = 59 }        ; raw scan code
```

Supported keys: F1–F24, A–Z, 0–9, modifier combinations (`CTRL`, `SHIFT`, `ALT`, any combination), special keys: `ESCAPE`, `ENTER`, `SPACE`, `TAB`, `PAGEUP`, `PAGEDOWN`, `HOME`, `END`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `INSERT`, `DELETE`.

MIDI CC constants

azCore defines named constants for common MIDI CC numbers:

```
$BANK_MSB       ; CC 0
$MOD_WHEEL      ; CC 1
$BREATH         ; CC 2
$VOLUME         ; CC 7
$PAN            ; CC 10
$EXPRESSION     ; CC 11
$SUSTAIN        ; CC 64
$PORTAMENTO     ; CC 65
$SOSTENUTO      ; CC 66
$SOFT           ; CC 67
$RESONANCE      ; CC 71
$CUTOFF         ; CC 74
$REVERB_SEND    ; CC 91
$CHORUS_SEND    ; CC 93
$ALL_NOTES_OFF  ; CC 123
; ... 50+ constants total
```

midi.device

Declares a logical MIDI device.

Syntax

```
midi.device <id> {
    caption = "<display name>"
    direction = output | input
    channel = omni | same | <1..16>
    [force = on | off]
    [panic = on | off]
}
```

Properties

Property	Type	Default	Description
`caption`	string	—	Human-readable label shown in UI
`direction`	enum	—	`output`: send MIDI; `input`: receive MIDI
`channel`	enum/int	—	`omni`: all channels; `same`: pass through; `1`–`16`: remap to channel
`force`	boolean	`off`	When `on`, always send PC even if same value was sent before
`panic`	boolean	`on`	When `on`, device receives All Notes Off during panic

Notes

- A logical device with `direction = input` becomes available as a source for routes and triggers. You reference it by its `<id>`.
- A logical device with `direction = output` becomes a target for `midi.send`, routes, and automations.
- Physical binding (which hardware port corresponds to this logical device) is defined in `azCore.ini [midi_bindings]`, not in `.az` files.
- If a device is declared but not bound, commands targeting it are silently skipped (with a warning). This allows files to remain valid even when hardware is not connected.

Example

```
midi.device piano {
    caption = "Roland Piano"
    direction = output
    channel = 1
}
```

```
midi.device controller {
    caption = "Arturia Keylab"
    direction = input
    channel = omni
}
```

midi.send

Sends one or more MIDI messages immediately to one or more output devices, bypassing the router.

Syntax

```
; Device outside the block
midi.send <device_id> { <messages> }
midi.send <device_id>,<device_id2> { <messages> }

; Device inside the block
midi.send { device = <device_id>  <messages> }
```

Notes

- Multiple target devices can be specified as a comma-separated list.
- The `device =` form inside the block is equivalent; choose based on readability.
- `midi.send` ignores the router (always sends regardless of `router.enabled`).
- Channel remapping defined on the `midi.device` is applied.
- If PC deduplication is active (`force = off` on the device), Program Change messages to the same value are suppressed. Use `force = on` to always send.

Examples

```
; Send program change and volume
midi.send piano { pc,3  cc,7,100 }

; Send to multiple devices at once
midi.send piano,organ { cc,123,0 }    ; All Notes Off to both

; Variable interpolation
var setup = midi{ pc,17  cc,7,90 }
midi.send organ { $setup }

; Named CC constants
midi.send piano { cc,$EXPRESSION,127 }
```

Limits

- There is no limit on the number of messages per `midi.send` call.
- Sending to an unbound device produces a warning and is skipped.

---

midi.automation

Performs a linear sweep of a MIDI CC value from a starting value to a target value over a specified duration.

Syntax

```
; Declare and start
midi.automation <id> {
    output    = <device_id>
    controller = <0..127> | $<CC_CONSTANT>
    from      = <0..127>
    to        = <0..127>
    duration  = <duration>
}

; Stop (sends final value)
midi.automation.stop <id>

; Cancel (sends nothing)
midi.automation.cancel <id>
midi.automation.cancel all
```

Properties

Property	Type	Required	Description
`output`	device id	yes	Target MIDI output device
`controller`	int / constant	yes	MIDI CC number (0–127)
`from`	int	yes	Starting CC value
`to`	int	yes	Target CC value
`duration`	duration	yes	Total sweep duration

Notes

- The automation sends CC updates on every tick (configurable via `automation_tick_ms` in the INI, default 20ms).
- Automations have **ownership**: an automation declared inside a song body is automatically cancelled when that song is unloaded. An automation declared at global scope persists until explicitly stopped.
- Multiple automations can run simultaneously, even on the same CC number (last-write wins per tick).
- `midi.automation.stop` sends the final `to` value before stopping.
- `midi.automation.cancel` stops without sending the final value.

Example

```
; Fade expression in over 4 bars
midi.automation expr_swell {
    output     = strings
    controller = $EXPRESSION
```

```
    from       = 0
    to         = 127
    duration   = 8s
}

; Stop it 8 seconds later from a pad
pad preset(3) {
    caption = "Stop swell"
    type    = instant
    on_press { midi.automation.stop expr_swell }
}
```

---

midi.clip

Declares a MIDI file (`.mid`) for playback. Useful for click tracks, arpeggiators, and backing MIDI sequences.

Syntax

```
midi.clip <id> {
    media    = "<file.mid>"
    output   = <device_id>
    [volume  = <0..100>]
    [start_pos = <duration>]
    [end_pos   = <duration>]
}

midi.play  <id>
midi.pause <id>
midi.stop  <id>
```

Notes

- File path is relative to `project_root/media_dir` (from the INI).
- The `output` device must be an `output` direction MIDI device.
- MIDI clips bypass the router and are sent directly to the output device, like `midi.send` and `midi.automation`. Route filtering does not apply.
- Playback position is tracked and shown in the UI.

---

route

Declares a MIDI routing rule connecting input device(s) to output device(s), with optional split, transpose, velocity remapping, and latch.

Declaration syntax

```
route <id> {
    <inputs> <outputs>
    [split    = <low>:<high>]
    [transpose = <semitones>]
    [velocity  = <min>:<max>]
    [auto_hold = on | off]
    [send { <messages> }]
}
```

Activation / management syntax

```
; Declare only (inactive)
route r_piano { keys1 piano }

; Declare and activate immediately (inline form)
route.add r_piano { keys1 piano  split = 36:96 }

; Activate existing route
route.add r_piano

; Activate with exclusive cleanup
route.add r_split { keys1 piano  exclusive(keys1, keys2) }

; Modify active route
route.set r_piano { transpose = 12 }

; Deactivate (Smooth Patch applied)
route.remove r_piano

; Delete from catalog
route.delete r_piano
```

Route body

The first line after { defines the connections: `<input_devices> <output_devices>`.

- Input and output devices are space-separated lists of logical device IDs.
- The `send` sub-block sends initialization messages when the route is activated.

Properties

Property	Type	Description
`split`	`<low>:<high>`	

Property	Type	Description
		Only forward notes with MIDI numbers in [low, high] (inclusive)
`transpose`	integer	Shift note number by N semitones (can be negative)
`velocity`	`<min>:<max>`	Clamp incoming velocity to [min, max] range
`auto_hold`	boolean	When `on`, notes stay held until pressed a second time (latch)

auto_hold

When `auto_hold = on`, the route implements a note latch mechanism. The performer plays a chord, releases all keys, and the chord sustains indefinitely without holding keys or using the sustain pedal. A new NoteOn terminates the current chord and begins composing a new one.

States:

- **IDLE:** No notes in play. Normal NoteOff forwarding.
- **COMPOSITION:** At least one NoteOn received. NoteOff messages are trapped (not forwarded). The chord sustains while the performer holds or releases keys.
- **LATCHED:** All NoteOffs for all pressed notes have arrived and been trapped. No physical keys held. Chord sustains indefinitely.

Transitions:

- IDLE → COMPOSITION: first NoteOn received.
- COMPOSITION → LATCHED: all held notes have a corresponding trapped NoteOff.
- LATCHED → COMPOSITION: new NoteOn arrives — trapped NoteOffs are flushed (old chord ends), new note begins.
- Any → IDLE: `auto_hold = off`, `router.panic`, or route removal.

Re-press during COMPOSITION: if a note that already has a trapped NoteOff is pressed again, the trapped NoteOff is forwarded first (ending the old instance of that note), then the new NoteOn begins normally.

CC64 blocked: the sustain pedal (CC 64) is discarded on auto_hold routes. A console warning is emitted. `sustain_on` is never set on an auto_hold instance.

route.set locked: while `auto_hold = true` is active, all `route.set` property changes are rejected except `auto_hold = off`. This prevents stuck notes from mid-note transform mismatches.

Output exclusivity: an auto_hold route requires exclusive ownership of its output. No other route (ACTIVE or RELEASE) may share the same output simultaneously. Exclusivity is enforced at `route.add` time; conflicting instances are not activated and a warning is emitted.

Stopping auto_hold: `route.set <id> {auto_hold = off}` flushes all trapped NoteOffs to the output and resumes normal forwarding. No CC 123 is sent. `route.remove` also terminates auto_hold — trapped NoteOffs are drained before the Smooth Patch RELEASE cycle begins, so no notes are left stuck.

```
route r_pad { keys2 synth  auto_hold = on }

; Disable at runtime — flushes trapped NoteOffs
route.set r_pad { auto_hold = off }
```

`exclusive()` keyword

```
route.add r_new { keys1 synth  exclusive(keys1) }
```

When `exclusive(keys1)` is specified, azCore removes all currently active routes that use `keys1` as input before adding `r_new`. This is the standard mechanism for switching keyboard assignments cleanly.

Smooth Patch details

When `route.remove` is called: 1. Route enters **RELEASE** state — only `note_off` and `cc,64,0` (sustain off) are forwarded. 2. azCore tracks all held notes on the route's output(s). 3. When all held notes clear, or after the grace period (`grace_time_ms`), the route exits RELEASE and is fully deactivated. 4. If the output becomes idle, azCore sends CC 123 (All Notes Off) and CC 64=0 as a safety reset.

Examples

```
; Simple keyboard-to-piano route
route r_piano { keys1 piano }

; Split: low keys to bass, high to piano
route r_low  { keys1 bass   split = 36:59 }
route r_high { keys1 piano  split = 60:96 }

; Layer: both synths play simultaneously
route r_layer { keys1 piano,strings }

; Transposing route
route r_up   { keys1 organ  transpose = 12 }

; Activate with exclusive: switch from piano to organ
route.add r_organ { keys1 organ  exclusive(keys1) }

; Modify transpose on the fly
route.set r_organ { transpose = -12 }
```

Limits

- A route can have multiple inputs and multiple outputs.
- `split`, `transpose`, and `velocity` are per-route, not per output.
- Routes persist across song changes; they are not automatically removed when a song unloads. Manage route lifecycle explicitly.
- `route.set` modifies only active instances — the declared template is unchanged. Calling `route.add` on an already-active route restores all instance properties to the template's declared values (reverting any `route.set` changes) and re-executes the `send` block.

---

audio.device

Declares a logical audio output device.

Syntax

```
audio.device <id> {
    caption        = "<display name>"
    [default_volume = <0..100>]
}
```

Properties

Property	Type	Default	Description
`caption`	string	—	Human-readable label
`default_volume`	int	100	Baseline volume for all clips on this device

Notes

- Physical binding is in `azCore.ini [audio_bindings]`.
- The effective volume of a clip is proportional to both `default_volume` and the clip's own `volume`.
- If the bound device is not found, clips on it are silently skipped with a warning.

---

audio.clip

Declares a playable audio asset.

Syntax

```
audio.clip <id> {
    media    = "<file.wav|mp3>"
    output   = <audio_device_id>
    [volume    = <0..100>]
    [start_pos = <duration>]
    [end_pos   = <duration>]
    [fade_in   = <duration>]
    [fade_out  = <duration>]
    [preload   = on | off]
}
```

Playback commands

```
audio.play    <id>
audio.pause   <id>
audio.stop    <id>
audio.preload <id>          ; on-demand background decode
audio.clip.set <id> { volume = <0..100> }
```

Properties

Property	Type	Default	Description
`media`	string	—	File path relative to `media_dir`
`output`	device id	—	Target audio device
`volume`	int	100	Playback volume 0–100
`start_pos`	duration	0	Start position within file
`end_pos`	duration	(end)	End position within file
`fade_in`	duration	0	Linear fade-in from silence
`fade_out`	duration	0	Linear fade-out to silence before `end_pos`
`preload`	boolean	off	Decode at boot for instant playback

Preload behavior

`preload = on`**:** The clip is decoded silently during boot, before the UI starts. When `audio.play` is called, playback begins immediately with no decode latency.

`audio.preload <id>`**:** Queues the clip for background decode. A progress indicator appears in the UI. Once decode completes, the clip is Ready and subsequent plays are instant.

No preload: The first `audio.play` triggers decode, which may introduce a small delay for large files. Subsequent plays are instant (decoded samples are cached).

Examples

```
audio.device main_out {
    caption        = "Main PA"
    default_volume = 80
}

audio.clip thunder {
    media    = "thunder.wav"
    output   = main_out
    volume   = 70
    fade_in  = 500ms
    fade_out = 1s
    preload  = on
}

audio.clip intro_pad {
    media     = "strings_pad.wav"
    output    = main_out
    volume    = 60
    start_pos = 2s
    end_pos   = 30s
    fade_in   = 2s
    fade_out  = 3s
}
```

Limits

- `.wav` (PCM) and `.mp3` are supported.
- Simultaneous playback of multiple clips on the same device is supported.
- The UI FADE button fades out the clip over `clip_default_fadeout_ms` (INI, default 10s).
- `audio.stop all` stops all currently playing clips.

---

pad

Declares a programmable button on the UI.

Syntax

```
pad <slot> {
    caption  = "<label>"
    type     = instant | toggle | repeat
    [color   = <color_name> | #rrggbb]
    on_press {
        <commands>
    }
    [on_release {
        <commands>
    }]
}
```

Slot types

Slot	Slots	Description
`preset(1..16)`	16	Main button row at bottom of UI
`board(1..48)`	48	Extended 4×12 grid (shown via BOARD toggle)
`custom`	1	Single custom command button
`next`	1	System: calls `step.next`
`prev`	1	System: calls `step.prev`
`panic`	1	System: All Notes Off + reset controllers
`force`	1	System: toggle PC deduplication override

Button types

Type	Behavior
`instant`	Fires `on_press` when pressed; `on_release` when released (optional)
`toggle`	First press fires `on_press` and enters ON state; second press fires `on_release` and returns to OFF
`repeat`	Fires `on_press` immediately and repeatedly every `preset_repeat_every_ms` while held

Commands

```
; Programmatic control
pad.press   <slot>           ; simulate press (toggles toggle pads)
pad.release <slot>           ; simulate release
pad.reset   <slot>           ; reset toggle state to OFF
pad.set     <slot> { caption = "<new label>"  color = blue }
```

```
pad.set     <slot> { state = pressed }  ; set toggle state without executi
pad.force_press <slot>       ; idempotent: press only if OFF (executes on_
pad.force_release <slot>     ; idempotent: release only if ON (executes on
```

State alignment: `pad.set {state=...}` and `pad.force_press/release` are useful after a plugin preset change that resets the UI state. `pad.set` changes state without triggering handlers; `pad.force_press/release` execute the handler only if the current state differs from the target state.

Auto-release anchor

The `releasing:` keyword inside `on_press` instructs azCore to automatically release a `toggle` pad when the specified async operation completes:

```
pad preset(2) {
    type     = toggle
    on_press  { releasing: audio.play background_pad }
    on_release { audio.stop background_pad }
}
```

When `background_pad` finishes playing (reaches end or fades out), the pad is automatically released and `on_release` is called.

Scope and stacking

- Pads declared at **global scope** are the base layer.
- Pads declared inside a **song body** shadow the global pads of the same slot.
- When a song is unloaded, its local pads are removed and the global pads for those slots are restored.

Examples

```
; Simple navigation pad
pad next {
    type     = instant
    on_press { step.next }
}

; Toggle: send MIDI on press, silence on release
pad preset(1) {
    caption  = "Piano on"
    type     = toggle
    color    = blue
    on_press  { midi.send piano { cc,7,100 } }
    on_release { midi.send piano { cc,7,0   } }
}

; Auto-release: pad releases when clip finishes
pad preset(3) {
    caption  = "Storm FX"
    type     = toggle
    on_press  { releasing: audio.play storm }
```

```
    on_release { audio.stop storm }
}
```

trigger

Binds a MIDI or keyboard event to a pad slot, or defines a standalone custom event handler.

Binding mode (connects to a pad slot)

```
trigger <slot> {
    [press   = midi{ device=<id>  <pattern> }]
    [press   = shortcut{ key = "<key>" }]
    [release = midi{ device=<id>  <pattern> }]
    [enabled = on | off]
}
```

Binding mode connects incoming events to the pad's `on_press` / `on_release` handlers. Multiple `press` events can be specified (any of them fires the press).

Standalone mode (custom handler)

```
trigger <custom_id> {
    [fire    = midi{ device=<id>  <pattern> }]
    [fire    = shortcut{ key = "<key>" }]
    [enabled = on | off]
    on_fire {
        <commands>
    }
}
```

Standalone mode defines an independent handler that is not connected to any pad.

Management commands

```
trigger.set    <id> { enabled = on | off }
trigger.delete <id>
```

Event pattern syntax

```
; Match specific note on from device
press = midi{ device=icon  note_on,$C1,any }

; Match any CC value
fire  = midi{ device=backrig  cc,50,any }

; Match specific CC value
press = midi{ device=backrig  cc,50,127 }

; Match any note_off
release = midi{ device=icon  note_off,$C1,any }

; Keyboard shortcut
press = shortcut{ key="PAGEDOWN" }
press = shortcut{ key="CTRL+SHIFT+F1" }
```

```
; Raw scan code
fire  = shortcut{ scan=59 }
```

The `any` keyword matches any value for the velocity or CC value position.

Scope

- Triggers declared at global scope persist across song changes.
- Triggers declared inside a song body are active only while that song is loaded.
- Binding mode and standalone mode cannot be mixed in the same `trigger` block.

Examples

```
; Bind step navigation to MIDI footswitch and keyboard
trigger next {
    press = midi{ device=footswitch  note_on,$C0,any }
    press = shortcut{ key="PAGEDOWN" }
    enabled = on
}

; Standalone: custom panic combination
trigger tr_emergency {
    fire = shortcut{ key="CTRL+ALT+P" }
    enabled = on
    on_fire {
        router.panic
        audio.stop all
        message "EMERGENCY STOP" { color = red  duration = 5s }
    }
}

; Enable/disable at runtime
trigger.set tr_emergency { enabled = off }
```

---

song

Groups all resources and steps for one song in the performance.

Syntax

```
song <id> {
    caption  = "<display name>"
    [version = "<version string>"]
    [update  = "<change note>"]
    [duration = <duration>]

    ; ── Init zone (declarations + commands, run once on load)
    var tempo = 120
    route r_piano { keys1 piano }
    block setup { ... }

    ; ── Steps (navigation targets, executive commands only)
    step default { ... }   ; mandatory
    step verse   { ... }
    step chorus  { ... }

    ; ── Post-zone (declarations only, after last step)
    trigger my_tr { ... }
    pad preset(1) { ... }
}
```

Song zones

Init zone: Everything before the first `step` declaration. Can contain both declarations (routes, blocks, variables, pads, triggers) and executive commands (midi.send, audio.play, wait, etc.). Runs exactly once when the song is loaded.

Steps: Everything from the first `step` to the end of the song body (or start of post-zone). Steps contain only executive commands. The `step default` step is mandatory and executes automatically when the song loads.

Post-zone: Declarations that appear after the last step. Typically used for triggers and pads that should be active for the entire song but reference resources declared in the init zone.

Load commands

```
load song <id>       ; load by ID
load song            ; open song selector UI
close song           ; unload current song
```

Song lifecycle

1. Previous song's local pads/triggers removed; global pads/triggers restored.
2. Previous song's automations (if owned by that song) cancelled.
3. Init zone executes: declarations processed, commands run in order.

4. `step default` executes automatically.
5. Song is now active; steps can be navigated.

Examples

```
song time {
    caption  = "Time"
    duration = 7m30s
    version  = "live 2026"

    var piano_pc = 3
    var organ_pc = 18

    route r_piano { keys1 piano  split = 36:96 }
    route r_organ { keys1 organ }

    step default {
        route.add r_piano
        midi.send piano { pc,$piano_pc  cc,7,100 }
        message "Intro — keys only" { duration = 3s }
    }

    step verse {
        route.add r_organ { exclusive(keys1) }
        midi.send organ { pc,$organ_pc  cc,7,100 }
        cue band_cues "Verse"
    }

    step solo {
        route.add r_piano { exclusive(keys1) }
        cue band_cues "Solo"
    }
}
```

---

step

Declares a navigation section within a song.

Syntax

```
step <id> {
    [caption = "<display name>"]
    <commands>
}
```

Navigation commands

```
step.next          ; go to next step in declaration order
step.prev          ; go to previous step
step.goto <id>     ; jump to specific step
call step <id>     ; execute step as subroutine (no navigation)
```

Notes

- `step default` is mandatory in every song. It executes when the song loads.
- Steps can contain both **declarations** (route, trigger, pad, var, block) and **executive** commands (actions).
- There is no per-step scope; all steps share the same song scope. Entities declared inside a step are song-scoped and visible to all other steps.
- Navigation does not re-run the init zone; only the target step's commands execute.
- `call step` executes the step's commands in place without changing the active step pointer.

Examples

```
step chorus {
    caption = "Chorus"
    route.add r_full { keys1 piano,strings  exclusive(keys1) }
    midi.send piano  { cc,7,127 }
    midi.send strings { cc,11,100 }
    cue lighting "Full bright"
    cue band_cues "Chorus — everybody"
}

step outro {
    caption = "Outro"
    midi.automation fade_out {
        output     = piano
        controller = $EXPRESSION
        from       = 127
        to         = 0
        duration   = 16s
    }
}
```

setlist

Defines an ordered sequence of songs.

Syntax

```
setlist <id> {
    [caption = "<display name>"]
    [transition = <time>]
    song <song_id>
    song <song_id>
    marker "<caption>"
    song <song_id>
    ...
}
```

Properties

Property	Type	Description
`caption`	string	Display name shown in the UI
`transition`	time	Default transition time between songs (optional)

Notes

- The setlist is shown in the song selector UI.
- Switching between setlist mode and “View All” catalog is possible at runtime.
- The last-used setlist is saved to the INI and restored on next boot.
- Navigation does not wrap around: `step.next` on the last song stays on the last song.
- Song durations (from `duration` property) are shown in the selector.
- `marker "<caption>"` inserts a visual separator in the setlist UI (e.g., “ACT 1”, “ENCORE”).

Example

```
setlist concert_2026 {
    caption = "Main Show"
    transition = 2s
    song intro
    marker "ACT 1"
    song time
    song money
    marker "ACT 2"
    song shine_on
    song another_brick
    song comfortably_numb
}
```

---

score

Associates an HTML score document with a display context.

Syntax

```
score <id> {
    caption  = "<display name>"
    media    = "<file.html>"
    [marker  = <anchor_id>]
    [align   = top | center | bottom]
    [factor  = <25..400>]
}
```

Display commands

```
score.show <id>                              ; show score
score.show <id> { marker = verse2 }          ; show at marker
score.show <id> { top = 800 }               ; show at pixel offset
score.show <id> { factor = 150 }            ; show at 150% zoom
score.hide                                   ; hide score area

score.set <id> { caption = "New title"  factor = 120 }

score.scroll <id> { direction = up | down | top | bottom  [speed = <ms>] }
score.zoom   <id> { level = <25..400>  [smooth = <time>] }
```

Properties

Property	Type	Description
`caption`	string	Label shown in UI header
`media`	string	Path to HTML file, relative to `scores_dir`
`marker`	string	HTML anchor to scroll to (`#<marker>` fragment)
`align`	enum	`top`, `center`, `bottom` — initial scroll alignment
`factor`	integer (25–400)	Zoom factor as percentage. 100 = no zoom, 200 = 200%, etc.

Notes

- The score iframe has full browser capabilities: CSS, JavaScript, images, fonts.
- `marker` corresponds to an HTML `id` attribute: `<div id="verse1">...</div>`.
- Scores persist across step navigation and song changes until explicitly hidden.
- `score.hide` shows the off-page display (configured in the INI or as a custom HTML page).

---

cue

Sends a short text message to a named label in the CUE PANEL.

Syntax

```
cue <label> "<text>"
cue <label> "<text>" { duration = <duration> }
cue <label> ""                    ; clear/flash (metronome use)
```

Predefined labels

Label	Color	Use
band_cues	Yellow	Musical direction to band
tech_cues	Blue	Technical instructions to crew
video	Purple	Cue to video operator
cam	Purple	Cue to camera operator
cam_rec	Dark red/pink	Recording status for camera operator (e.g. “REC”, “STOP REC”)
lighting	Green	Cue to lighting director
mixer	Cyan	Cue to FOH/monitor engineer
br2azcore	White	Incoming cue from external source
click_ti, click_ta	Red/Green	Metronome strong/weak beat flash
click_ci, click_cia	—	Metronome compás indicators

Notes

- The default display duration is set in the INI (default_cue_duration_ms, default 3000ms for communication labels, default_cue_click_duration_ms = 250ms for click labels).
- Sending a new cue to the same label replaces the previous message immediately.
- Cues auto-dismiss after the duration expires.
- Empty string "" produces a flash (for metronome click use).

---

message

Displays a centered overlay message on the UI.

Syntax

```
message "<text>"
message "<text>" { duration = <duration> }
message "<text>" { duration = <duration>  color = <color> }
```

Notes

- The message appears as a large centered overlay in the score area.
- If `duration` is omitted, the UI applies its own default display time. Specify `duration` explicitly for reliable behavior.
- Supports variable interpolation: `message "Now: $song_name"`.
- Color names: `red`, `green`, `blue`, `yellow`, `white`, `orange`, or `#rrggbb`.
- A new message replaces any current message.

---

var

Declares a variable.

Syntax

```
var <name> = <value>
[global | local] var <name> = <value>
override var <name> = <value>   ; redeclare at same scope
```

Scope

- Variables declared in the global scope are available everywhere.
- Variables declared inside a song body are local to that song.
- Variables declared inside a step are in the enclosing song scope (there is no per-step scope).
- The `global` keyword forces declaration at global scope even when written inside a song.
- The `local` keyword forces declaration at song scope.
- `override var` allows redeclaring a variable that already exists at the same scope (used to change a default).

Types

Variables are dynamically typed at declaration. Supported types: integer, string, boolean, duration, MIDI message block, shortcut block.

Examples

```
; Global
var master_vol = 100
var piano_init = midi{ pc,3  cc,7,100 }

; Song-local
song time {
    var pc_piano = 18
    var pc_organ = 42

    step default {
        midi.send piano { pc,$pc_piano }
        midi.send organ { pc,$pc_organ  cc,7,$master_vol }
    }
}
```

---

block

Declares a named reusable code section.

Syntax

```
block <id> {
    <commands>
}

call block <id>
include block <id> when { profile = "<profile_name>" }
```

Notes

- Blocks are **flat**: they do not create a new scope. Variables declared inside a block are in the enclosing scope.
- Blocks can be called from anywhere in the same scope (global or song).
- `include block` is a compile-time construct; the block's content is inlined at the call site if the profile condition matches.
- Blocks can call other blocks (nesting is supported); cycle detection prevents infinite recursion.
- Blocks are primarily used to avoid repetition in `step` bodies.

Example

```
block reset_all {
    route.remove r_piano
    route.remove r_organ
    midi.send piano  { cc,7,0 }
    midi.send organ  { cc,7,0 }
}

song time {
    step default { call block reset_all }
    step verse   { call block reset_all  route.add r_organ }
}
```

---

wait

Schedules a deferred code block to execute after a specified duration.

Syntax

```
wait <duration>

wait <duration> {
    <commands>
}
```

Notes

- `wait` is **non-blocking**: execution continues immediately to the next statement. The deferred block executes in the background after the specified duration.
- `wait` without a block: does nothing (no-op).
- `wait` with a block: the block is scheduled and executes after the delay.
- Multiple waits can be pending simultaneously.
- `flush` cancels all pending waits. The deferred blocks are discarded.
- Use waits sparingly in performance-critical paths; prefer trigger-based flow control.

Example

```
step outro {
    ; Fade expression over 8 seconds, then silence
    midi.automation expr_fade {
        output     = piano
        controller = $EXPRESSION
        from       = 127
        to         = 0
        duration   = 8s
    }
    wait 8s {
        midi.send piano { cc,7,0  cc,11,0 }
    }
}
```

---

router

Controls the global MIDI router.

Syntax

```
router.enabled = on | off
router.filter = default_allow
router.filter = default_deny
router.filter = allow(note_on, note_off, cc, pc, pitch_bend, after_touch,
router.filter = deny(note_on, note_off, cc, pc, pitch_bend, after_touch, s
router.panic
```

Filter modes

- `default_allow` — all message types pass through (default mode)
- `default_deny` — all message types are blocked
- `allow(...)` — whitelist specific message types
- `deny(...)` — blacklist specific message types

Filter message types

Token	MIDI message
`note_on`	Note On
`note_off`	Note Off
`cc`	Control Change
`pc`	Program Change
`pitch_bend`	Pitch Bend
`after_touch`	Aftertouch
`sysex`	System Exclusive
`clock`	MIDI Clock
`active_sensing`	Active Sensing

Notes

- The router is **off by default**. Routes only become active when the router is enabled.
- `router.enabled = on` enables forwarding for all active routes.
- `router.filter` restricts which message types are forwarded.
- `router.panic` sends All Notes Off (CC 123) and reset controllers to all bound output devices on all 16 channels. This is the emergency stop for stuck notes.
- `midi.send` and `midi.automation` always bypass the router.

---

profile

Declares a named profile for conditional compilation.

Syntax

```
profile <id> {
    caption = "<label>"
}
```

Notes

- A `profile` declaration only creates a named profile with an optional caption. It does not contain code.
- The active profile is set in `azCore.ini [session] last_profile`.
- The `when` construct references a profile to conditionally execute code based on the active profile.
- Profiles are evaluated at compile time; no runtime overhead.

Example

```
profile live {
    caption = "Live Performance"
}

profile rehearsal {
    caption = "Rehearsal Mode"
}

when { profile = "live" } {
    var click_vol = 0          ; no click in FOH
}

when { profile = "rehearsal" } {
    var click_vol = 80         ; click audible in rehearsal
}

audio.clip click {
    media  = "click.wav"
    output = monitor
    volume = $click_vol
}
```

---

on_song_load / on_song_unload

Lifecycle hooks called at song load and unload.

Syntax

```
on_song_load {
    <commands>
}

on_song_unload {
    <commands>
}
```

Notes

- These hooks are declared at global scope and apply to every song.
- `on_song_unload` is called before the old song's state is torn down.
- `on_song_load` is called before the new song's init zone runs — song-scoped variables declared in the init zone are not yet available inside `on_song_load`.
- Useful for global setup/teardown that should happen on every song transition.

Example

```
on_song_unload {
    audio.stop all
    midi.automation.cancel all
}

on_song_load {
    cue tech_cues "Loaded: $song_name"
}
```

Console commands reference

The console starts automatically when azCore launches. All commands are also available via the UI command interface.

Song and step navigation

Command	Description
`load song <id>`	Load song by ID
`load song`	Open song selector UI
`close song`	Unload current song
`step.next`	Go to next step
`step.prev`	Go to previous step
`step.goto <id>`	Jump to named step
`show current song`	Display active song details
`show current step`	Display active step details

Inspection

Command	Description
`show songs`	All songs in catalog
`show steps`	Steps in current song
`show vars`	All variables with scope
`show local vars`	Song-local variables only
`show router`	Router enabled/filter status
`show routes`	All routes with status
`show active routes`	Only active/release-state routes
`show pads [preset|board|all]`	Pad status with types and states
`show all pads`	All slots including empty
`show triggers`	All triggers
`show active triggers`	Triggers with `enabled = on`
`show scores`	Declared scores and active
`show ledger`	Notes currently held on all devices
`show ledger ghost`	Potentially stuck notes
`show midi ports`	Available MIDI ports
`show audio ports`	Available audio devices

MIDI management

Command	Description
`midi monitor on`	Enable MIDI monitor (IN+OUT)
`midi monitor off`	Disable MIDI monitor
`midi monitor in`	Monitor input only
`midi monitor out`	Monitor output only

Command	Description
`midi monitor in all`	Monitor all inputs including unbound
`midi monitor all raw`	Show raw bytes
`midi bind <id> <index>`	Bind logical device to port number
`midi unbind <id>`	Remove binding
`midi rescan`	Rescan MIDI devices
`audio bind <id> <name>`	Bind audio device (exact name string)
`audio unbind <id>`	Remove audio binding
`audio rescan`	Rescan audio devices

Testing

Command	Description
`load song <id> --test`	Auto-step through song at 1s intervals
`load song --testall`	Test all songs in sequence
`test stop`	Stop test mode

Emergency

Command	Description
`panic`	All Notes Off + reset controllers on all outputs
`stop player`	Stop all audio and MIDI clips
`flush`	Cancel all pending waits and automations

Ledger (ghost note detection)

Command	Description
`show ledger`	All notes currently held
`show ledger ghost`	Notes that may be stuck
`ledger check`	Scan and auto-fix stuck notes

System

Command	Description
`help`	List available commands
`quit`	Exit azCore

Configuration: azCore.ini

The INI file is automatically created on first run and updated at runtime (when devices are bound, when session state changes). It is divided into sections.

[paths]

```
[paths]
project_root  = "./project"   ; root directory for .az files
autoexec      = "autoexec.az" ; main entry point file
media_dir     = "media"       ; subdirectory for audio/MIDI files
scores_dir    = "scores"      ; subdirectory for HTML score files
```

[runtime]

```
[runtime]
automation_tick_ms          = 20     ; MIDI automation
        granularity
grace_time_ms               = 2000   ; Smooth Patch grace period
preset_repeat_every_ms      = 500    ; Repeat pad fire interval
midi_map_c3                 = 60     ; C3 MIDI note number (60 =
        Yamaha)
start_midi_monitor          = off    ; auto-enable MIDI monitor
        on boot
default_cue_duration_ms     = 3000   ; cue label auto-dismiss
        delay
default_cue_click_duration_ms = 250  ; click cue auto-dismiss
        delay
clip_progress_ms            = 250    ; UI progress ring update
        interval
clip_warn_ms                = 3000   ; remaining-time warning
        threshold
clip_default_fadeout_ms     = 10000  ; UI FADE button fade
        duration
audio_lost_check_interval_ms = 500
audio_rescan_interval_ms    = 2000
midi_lost_check_interval_ms = 500
midi_rescan_interval_ms     = 5000
```

[session] (auto-updated)

```
[session]
last_profile = "default"
last_setlist = "concert_2026"
last_song    = "welcome"
```

[ui]

```
[ui]
autostart_ui    = window    ; off / server / window
ui_port         = 7891
ui_window_x     = 0
ui_window_y     = 0
ui_window_w     = 1366
ui_window_h     = 740
ui_maximized    = false
ui_monitor      = 0
ui_resize_lock  = true      ; freeze layout for live use
ui_simulate_press_ms = 200  ; visual flash duration on pad press
```

`autostart_ui` options: - `off` — no UI - `server` — start WebSocket server, open external browser manually - `window` — start WebSocket server and open native window

[midi_bindings]

```
[midi_bindings]
piano       = "IAC Driver Piano"
drums       = "ipMIDI Port 1"
controller  = "Arturia KeyLab 61"
```

Format: `logical_id = "physical port name"`. Port names must match exactly (case-sensitive on macOS). Run `show midi ports` in the console to see available port names.

[audio_bindings]

```
[audio_bindings]
main_out = "Dante Virtual Soundcard"
monitor  = "MacBook Pro Speakers"
```

Format: `logical_id = "OS audio device name"`. Run `show audio ports` for available names.

[console_aspect]

Colors for the console TUI. Accepts ANSI color names, xterm-256 indexed (`indexed:N`), or hex `#rrggbb`:

```
[console_aspect]
col_normal  = "#ffffff"
col_header  = "#00ffff"
col_success = "#00ff00"
col_error   = "#ff6666"
col_warn    = "#ffff00"
col_info    = "#00ffff"
col_song    = "#00ff00"
col_step    = "#ff8700"
```

```
col_prompt  = "#00ffff"
col_ui      = "#ff87ff"
```

[ui_aspect]

CSS-compatible colors for the graphical UI panels, buttons, and indicators. Keys correspond to CSS variables applied at runtime. Refer to the UI documentation for the full list.

Error codes reference

Format

All diagnostics follow this format:

```
LEVEL AZS-CATEGORY-NNN: Short description
  filename.az:line:column
  <source line>
  ^
  at context.az:10:5 (expanded from "call block setup")
```

Error categories

Category	Codes	Description
PARSE	001–006	Syntax errors, unexpected tokens, unterminated blocks
SEM	001–008	Semantic errors: type mismatch, unknown reference, duplicate declaration, scope violation
MIDI	001–002	Unbound device, automation failure
AUD	001	Audio device missing or unavailable
MEDIA	001–002	Audio/MIDI file not found or decode failure
SCORE	001–002	Score not found, marker not found in HTML
ROUTE	001–003	Router disabled warning, non-active route, route not found
PAD	001–003	Slot out of range, pad not found
TRIG	001–003	Mixed binding/standalone modes, trigger not found
BOOT	001–002	Device bind failed at boot (fatal)
RUN	001–003	Runtime reference errors

Severity levels

Level	Meaning
ERROR	Compilation stops; azCore will not start
WARNING	Printed but compilation continues; may indicate a problem
INFO	Informational; no action required

Part 4 — Troubleshooting, Support, and Contacts

Common issues

azCore does not start — compilation error

Symptom: Boot stops with `ERROR AZS-...` and azCore exits.

Cause: A syntax or semantic error in a `.az` file. The error message includes the exact file, line, column, and a description.

Solution: Open the indicated file, go to the indicated line, and fix the issue. Common causes: - Misspelled device or song ID - Missing `step default` in a song - Duplicate variable declaration (use `override var` to redeclare) - Wrong data type (e.g., string where number expected)

MIDI device not working — notes not routing

Symptom: You play keyboard but no MIDI reaches the synth.

Check list: 1. `show midi ports` — is the device listed? 2. `show routes` — is the expected route listed and in ACTIVE state? 3. Is `router.enabled = on` in your `.az` file or set via the REPL? 4. `midi monitor in` — are MIDI events arriving from the input device? 5. `midi monitor out` — are events leaving toward the output device? 6. `show midi ports` — is the output device bound (`midi bind <id> <index>`)?

If the route is active but no notes arrive: check the input device binding and the `split` range. If notes arrive but are not forwarded: check the router is enabled and the route is active. If notes arrive and leave but synth is silent: check the output device binding and channel.

Stuck notes

Symptom: A note is sustaining endlessly after a route change.

Solution: Press the **PANIC** button in the CUE PANEL, or type `panic` in the console. This sends All Notes Off + reset controllers to all output devices.

To investigate: `show ledger ghost` shows which notes are believed to be stuck. `ledger check` attempts automatic correction.

Prevention: Use `route.remove` instead of `route.delete` to trigger Smooth Patch. Ensure the grace period (`grace_time_ms`) is long enough for your playing style.

Audio clip does not play

Symptom: `audio.play` command executes but no sound is heard.

Check list: 1. `show audio ports` — is the audio device listed? 2. Is the device bound? (`audio bind <id> <name>` or check INI) 3. Is the file path correct relative to `media_dir`? 4. Is `volume` > 0 on both the clip and the device? 5. For preloaded clips: did the preload succeed? Check boot log for preload errors.

Score not showing

Symptom: `score.show` executes but the score area shows the off-page display.

Check list: 1. Is the HTML file present at the path in `media` (relative to `scores_dir`)? 2. `show scores` — is the score in the catalog? 3. If using `marker`: does the HTML file contain an element with that `id`? 4. Check the console log for `AZS-SCORE-001` or `AZS-SCORE-002` errors.

UI not accessible remotely

Symptom: Trying to open the UI from an iPad or second computer fails.

Check list: 1. Is `autostart_ui = server` or `window` in the INI? 2. Is the firewall allowing connections on `ui_port` (default 7891)? 3. Are both devices on the same network? 4. Connect to `http://<host-ip>:7891` in the browser (replace with the actual IP).

Song changes are slow / UI is unresponsive

Symptom: There is a noticeable delay when loading a song.

Most likely cause: An audio clip is being decoded on demand. Use `preload = on` for clips that must play immediately after song load, or call `audio.preload <id>` in `autoexec.az` or in `on_song_load` to pre-decode.

License / Trial issues

Symptom: “Trial expired” overlay appears on every startup.

Solution: Enter a valid license key in the About modal (click the `?` or the license area in the header). If you do not have a license key, contact support.

Symptom: “License not valid for this machine” error.

Cause: The license key is bound to a different machine ID.

Solution: Contact support with your machine ID (shown in the About modal) to request a license transfer.

Requesting support

Before contacting support, please gather the following: 1. Your azCore version (shown in the About modal or in the boot log). 2. Your OS and version. 3. The exact error message, including the `AZS-...` code if applicable. 4. The relevant section of your `.az` file (anonymized if needed). 5. The contents of `azCore.ini` `[midi_bindings]` and `[audio_bindings]` if the issue is device-related.

Run `show midi ports` and `show audio ports` in the console and include the output.

Contacts and further resources

Website: `https://azlive.io/azcore`

Documentation: Full technical reference, tutorials, and example projects are available at: `https://azlive.io/docs`

Issue tracker and community: `https://github.com/azcore-project/azcore`

Email support: `support@azlive.io`

License management: `https://azlive.io/activate`

---

azCore User Manual — Version 1.1.0 Last updated: April 2026 © 2026 azCore Project. All rights reserved.